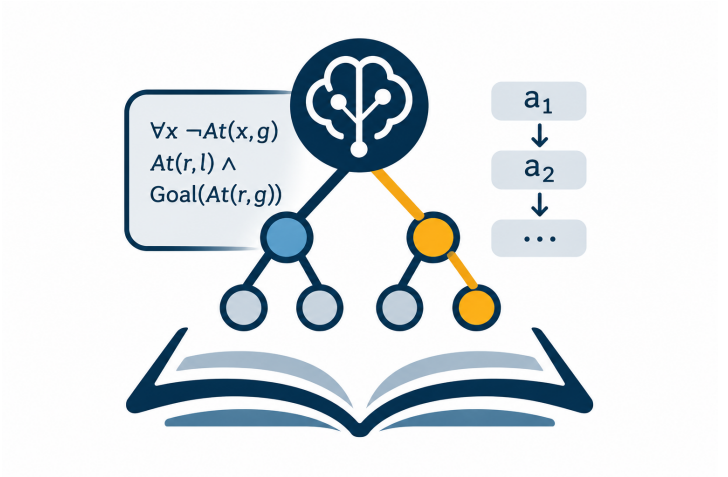
Planificación automática

Planificación y PDDL

Modelar problemas para planificadores automáticos, de PDDL 1.2 a PDDL+



Francisco Martín Rico

v0.1.0

21 de noviembre de 2026

Intelligent Robotics Lab, Universidad Rey Juan Carlos

Copyright
© 2026 Francisco Martín Rico.



Licencia
Este libro se distribuye bajo la licencia *Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional* (CC BY-NC-ND 4.0).

Se permite **compartir** (copiar y redistribuir) el material en cualquier medio o formato, con estas condiciones: **reconocimiento** (se debe dar crédito al autor), **uso no comercial** y **sin obra derivada** (no se puede remezclar, transformar ni crear a partir del material).

Texto completo de la licencia:
https://creativecommons.org/licenses/by-nc-nd/4.0/

Los ejemplos
Cada dominio y cada problema PDDL que aparece en este libro es un fichero de la carpeta `examples` que lo acompaña, y cada plan mostrado lo ha obtenido y validado el planificador Kairos (versión 1.0.6). La orden `make check-examples` los vuelve a ejecutar todos. Las salidas se obtuvieron con las opciones de esa orden, en particular `--threads 1`: con más hilos, Kairos lanza varias búsquedas a la vez y puede escribir otros planes, tan válidos como los que se muestran. Las salidas largas aparecen recortadas, y una línea `...` señala cada corte. Los ficheros PDDL, los nombres de sus objetos y la salida de Kairos se reproducen tal cual, en inglés, que es la lengua habitual de los modelos y de las herramientas de planificación.

Colofón
Este documento se ha compuesto con KOMA-Script y LaTeX, usando la clase kaobook.

«Los planes no valen nada, pero planificar lo es todo.»

— Dwight D. Eisenhower

«Todos los modelos son erróneos, pero algunos son útiles.»

— George E. P. Box

Prólogo

Un planificador es un programa que recibe la descripción de una situación, de lo que se puede hacer en ella y de lo que se quiere conseguir, y responde con una secuencia de acciones que lleva de lo uno a lo otro. El programa es general: el mismo planificador mueve robots en un hospital, programa las observaciones de un satélite o resuelve un rompecabezas. Lo que cambia de un uso a otro es la descripción, y escribir bien esa descripción es una habilidad en sí misma. Este libro enseña esa habilidad.

La descripción se escribe en PDDL, el *Planning Domain Definition Language*. PDDL nació en 1998 como lenguaje común de la primera competición de planificación y no ha dejado de crecer desde entonces: primero tipos y condiciones lógicas (PDDL 1.2), después números y tiempo (PDDL 2.1), hechos derivados y sucesos fijados en el calendario (PDDL 2.2), preferencias y restricciones sobre el plan completo (PDDL 3), variables de estado cuyo valor es un objeto (PDDL 3.1) y, en PDDL+, un mundo que cambia por sí solo, de forma continua, actúe el agente o no. El libro recorre el lenguaje completo en ese orden, desde el modelo más sencillo que un planificador puede resolver hasta los sistemas híbridos de los últimos capítulos.

Qué tiene de distinto este libro

La mayoría de las introducciones a PDDL explican el lenguaje: qué significa cada construcción y cómo se escribe. Este libro también explica el lenguaje, pero su tema es el *modelado*: cómo pasar de una situación descrita con palabras (un robot que debe entregar medicamentos antes de que cierre la farmacia, sin quedarse sin batería) a un modelo PDDL que un planificador resuelva bien, y cómo elegir entre los varios modelos que siempre son posibles.

Con ese fin, el libro construye un método, llamado MODEL por sus cinco pasos: **M**apear la situación, nombrar los **O**bjetos y su estado, describir la **D**inámica, **E**nsayar el modelo con un planificador y hacer una **L**ectura crítica de las alternativas. El método se presenta en el primer capítulo con el poco PDDL que necesita, y cada capítulo posterior le añade algo: cada característica nueva del lenguaje trae preguntas nuevas que hacerle a un problema, formas nuevas de codificarlo y formas nuevas de equivocarse. La última parte del libro pone el método completo a trabajar sobre un problema tomado de la robótica.

De ese propósito se siguen tres decisiones.

- **Un mundo que crece.** Un robot mensajero en una planta de hospital aparece en todos los capítulos. Empieza siendo un robot que coge y deja cosas; gana puertas, carros, una batería, un reloj, horarios de apertura, preferencias y, al final, una batería que se descarga sola. Ver el mismo mundo modelado con cada vez más partes del lenguaje muestra *para qué* sirve cada característica. A su alrededor, cada capítulo trae otros dominios pequeños, para que ninguna idea quede atada a un único ejemplo.
- **Alternativas, comparadas.** Casi todo se puede codificar de varias maneras. Cada capítulo pone dos o tres codificaciones de lo mismo una junto a otra, las ejecuta y dice cuándo conviene cada una.
- **Todo se ejecuta.** Cada dominio y cada problema del libro es un fichero que lo acompaña, y cada plan impreso lo ha obtenido un planificador y ha sido validado. Un modelo es un programa y, como un programa, se aprende ejecutándolo, rompiéndolo y arreglándolo.

El planificador

Los ejemplos usan Kairos, un planificador de código abierto que cubre todo el PDDL de este libro con un único programa: planificación clásica, numérica y temporal, preferencias y restricciones, y PDDL+. También comprueba modelos sin planificar, con mensajes de error que señalan la línea y sugieren la corrección, y valida planes. El Apéndice A explica cómo instalarlo y usarlo. Nada en los modelos depende de Kairos: son PDDL estándar, y el libro avisa de dónde otros planificadores leen una construcción de otra manera o no la admiten.

A quién va dirigido

El libro está escrito para una asignatura universitaria de planificación automática y para cualquiera que tenga que escribir modelos de planificación: profesionales de la robótica, ingenieros de sistemas autónomos, investigadores que usan un planificador como herramienta. Supone que el lector sabe programar y conoce la lógica básica (y, o, no, para todo, existe) y la búsqueda en grafos. No hace falta ningún contacto previo con la planificación.

Una asignatura de un cuatrimestre puede seguir el libro en orden. Las Partes I y II son el núcleo y no deberían saltarse; la Parte III (números) es necesaria para todo lo que viene después; las Partes IV, V y VI pueden impartirse en ese orden o recortarse al final de cualquiera de ellas. Cada capítulo termina con ejercicios para cada una de sus secciones, la mayoría para hacer delante del ordenador, y las soluciones están en el Apéndice C.

Índice general

Índice general vi

Parte I. Planificación y modelos 1

1. Planificar: decidir antes de actuar 2
1.1. Decidir antes de actuar . . . 2
El mundo del mensajero . . . 3
1.2. Un resolutor general más un modelo . . . 3
Los tres ingredientes: estados, acciones, objetivo . . . 4
1.3. Los supuestos clásicos . . . 5
1.4. La planificación y sus alternativas . . . 6
1.5. Dónde se usa la planificación . . . 8
1.6. Dominio, problema y PDDL . . . 9
PDDL y sus versiones . . . 9
1.7. Primer contacto con Kairos . . . 11
El mismo dominio, otro problema . . . 12
1.8. El método MODEL: cinco pasos y cinco criterios . . . 13
Los cinco criterios . . . 14
Alternativas comparadas . . . 15
1.9. Errores comunes . . . 15
1.10. Resumen . . . 16
1.11. Lecturas complementarias . . . 17
1.12. Ejercicios . . . 18

2. Un primer modelo: STRIPS 21
2.1. Listas dentro de listas: la forma de PDDL . . . 21
2.2. Predicados, objetos y hechos . . . 22
Lo que se puede decir: predicados . . . 22
Lo que es cierto: hechos y estados . . . 23
2.3. Acciones: precondiciones y efectos . . . 23
Aplicar una acción . . . 24
2.4. El problema: objetos, estado inicial y objetivo . . . 25
2.5. Modelar el mundo del mensajero . . . 26
Paso M: mapear la situación . . . 26
Paso O: objetos y estado . . . 27
Paso D: dinámica . . . 28
Un problema para el dominio . . . 29
2.6. Ensayar el modelo . . . 30
Comprobar, resolver, validar . . . 30
La batería de pruebas . . . 31
2.7. Dos clásicos: el mundo de bloques y gripper . . . 35
El mundo de bloques . . . 35
Gripper . . . 37
2.8. El método MODEL: la primera pasada completa . . . 38
Lo que añade este capítulo . . . 39
Alternativas comparadas . . . 39
2.9. Errores comunes . . . 41
2.10. Resumen . . . 44

2.11. Lecturas complementarias . . . 45
2.12. Ejercicios . . . 46

3. Qué hace un planificador con tu modelo **49**
3.1. Qué significa un modelo . . . 49
Un problema lo bastante pequeño para dibujarlo . . . 49
Estados . . . 50
Acciones aplicables y sucesores . . . 50
Planes . . . 51
El espacio de estados . . . 52
3.2. Instanciación: de variables a objetos . . . 52
Contar a mano . . . 53
Lo que dice el planificador . . . 53
La planta entera, el puzle y las torres . . . 54
3.3. Por qué explota el espacio . . . 56
Contar estados . . . 56
3.4. Búsqueda hacia delante . . . 57
El bucle . . . 57
Buscar a ciegas . . . 58
3.5. Heurísticas: plantear un problema más sencillo . . . 58
Un número para cada estado . . . 58
Ignorar los borrados . . . 59
La heurística en todo el espacio . . . 60
Cuando la relajación ve poco . . . 61
3.6. Un plan, un plan mejor, el mejor plan . . . 61
Anytime: el primer plan y luego otros mejores . . . 61
Elegir cuánto pedir . . . 63
3.7. No hay plan, o no se ha encontrado . . . 63
Demostrado por la instanciación . . . 63
Demostrado por la búsqueda . . . 64
No encontrado . . . 64
3.8. Qué significa esto para el modelador . . . 65
Los parámetros multiplican . . . 65
Las precondiciones estáticas podan . . . 66
Callejones sin salida . . . 66
Simetría . . . 67
3.9. El método MODEL: tamaño y guía en cifras . . . 68
Alternativas comparadas . . . 68
3.10. Errores comunes . . . 70
3.11. Resumen . . . 71
3.12. Lecturas complementarias . . . 72
3.13. Ejercicios . . . 73

Parte II. Planificación clásica: PDDL 1.2 **76**

4. Tipos, constantes e igualdad **77**
4.1. Lo que cuestan los predicados de tipo . . . 77
4.2. Tipos . . . 78
Lo que ahora detecta el comprobador . . . 80
Lo que gana el planificador . . . 81
4.3. Jerarquías de tipos . . . 81
Un segundo dominio: logistics . . . 84
4.4. Un parámetro de varios tipos: `either` . . . 86
La alternativa: dar nombre a la unión . . . 87

4.5. Constantes . . . 88
4.6. Igualdad . . . 89
Cuándo hace falta la desigualdad . . . 91
4.7. El método MODEL: clases de cosas . . . 92
Alternativas comparadas . . . 92
4.8. Errores comunes . . . 94
4.9. Resumen . . . 96
4.10. Lecturas complementarias . . . 97
4.11. Ejercicios . . . 97

5. Condiciones más ricas **100**
5.1. Condiciones sobre la ausencia: precondiciones negativas . . . 100
Tres estados con dos predicados . . . 102
Ensayar las puertas . . . 102
5.2. Objetivos negativos . . . 103
5.3. Alternativas en una condición: `or` e `imply` . . . 104
5.4. Algún objeto: `exists` . . . 105
Un rompecabezas de llaves y cofres . . . 106
5.5. Todos los objetos: `forall` . . . 107
El mundo de bloques: `clear` es la ausencia de `on` . . . 108
Un solo `move` para todos los pasos . . . 109
Objetivos cuantificados . . . 109
5.6. Qué hace un planificador con las condiciones ricas . . . 110
5.7. El método MODEL: ausencia, alguno, todos . . . 111
Alternativas comparadas . . . 112
5.8. Errores comunes . . . 114
5.9. Resumen . . . 115
5.10. Lecturas complementarias . . . 116
5.11. Ejercicios . . . 116

6. Efectos más ricos **119**
6.1. Efectos que dependen de la situación . . . 119
6.2. Efectos condicionales: `when` . . . 121
Qué significa un efecto condicional . . . 122
A ejecutarlo . . . 122
Una condición en el efecto o en la precondición . . . 123
6.3. Efectos universales: `forall` . . . 123
6.4. Los dos juntos: para todo objeto tal que . . . 124
Lights Out . . . 124
El maletín . . . 126
El carro . . . 127
6.5. Efectos que hay que evitar: el laboratorio estéril . . . 130
Ensayar el modelo . . . 130
Una acción por caso . . . 132
6.6. El requisito `:adl` . . . 133
6.7. Qué hacen los planificadores con los efectos condicionales . . . 133
6.8. El método MODEL: efectos que dependen de la situación . . . 135
Alternativas comparadas . . . 135
Una cuarta vía: no almacenar lo que se puede deducir . . . 137
6.9. Errores comunes . . . 138
6.10. Resumen . . . 140
6.11. Lecturas complementarias . . . 141
6.12. Ejercicios . . . 141

7. Modelar problemas clásicos: patrones y decisiones **145**
7.1. Un problema a partir de una historia: el camarero . . . 145
Paso M: mapear la situación . . . 146
Paso O: objetos y estado . . . 146
Paso D: dinámica . . . 147
Paso E: ensayar el modelo . . . 147
7.2. Patrones para el estado . . . 148
Tabla estática . . . 149
Hechos espejo . . . 150
Invariante de a lo sumo uno . . . 150
Cadena de sucesores . . . 151
Huecos . . . 152
7.3. Patrones para el control . . . 152
Testigo de fase . . . 153
Testigo de turno . . . 153
Cerrojo de recurso . . . 155
7.4. Patrones para las acciones . . . 155
Par inverso y acción sin vuelta atrás . . . 155
Acción fusionada . . . 157
7.5. Antipatrones . . . 158
Borrado olvidado . . . 158
Objetivo en el estado inicial . . . 158
Gemelos desacompasados . . . 159
Parámetro ocioso . . . 159
7.6. Depurar un modelo . . . 160
Sin plan . . . 160
Un plan erróneo . . . 161
7.7. El método MODEL: el método clásico, consolidado . . . 161
Alternativas comparadas . . . 163
7.8. Errores comunes . . . 164
7.9. Resumen . . . 165
7.10. Lecturas complementarias . . . 166
7.11. Ejercicios . . . 166

PARTE III. NÚMEROS: PDDL 2.1 **169**

8. Fluentes numéricos **170**
8.1. Lo que un hecho no puede decir . . . 170
8.2. Funciones: números en el estado . . . 171
8.3. Comparar y cambiar números: la batería . . . 172
Condiciones numéricas . . . 173
Efectos numéricos . . . 173
Ejecutarlo . . . 174
Ensayar el modelo . . . 177
8.4. Números que nunca cambian: distancias, pesos y capacidad . . . 178
Una capacidad de carga . . . 179
Otro dominio: un camión y su combustible . . . 180
8.5. Asignar, escalar y preguntar: jarras, diluciones y existencias . . . 181
Las jarras de agua . . . 182
Escalar: una dilución . . . 183
Contar: existencias en un almacén . . . 184
8.6. Valores que nunca se dieron . . . 185
8.7. Por qué los números dificultan la planificación . . . 187

8.8. El método MODEL: magnitudes . . . 188
Alternativas comparadas . . . 189
8.9. Errores comunes . . . 191
8.10. Resumen . . . 193
8.11. Lecturas complementarias . . . 194
8.12. Ejercicios . . . 194

9. Calidad del plan: métricas y costes de acción **197**
9.1. Cuando no basta con un plan cualquiera . . . 197
9.2. Costes de acción . . . 198
El requisito y sus reglas . . . 200
9.3. Costes a partir de datos: una cocina con precios . . . 200
Acciones que no cuestan nada . . . 202
9.4. El planificador en acción: búsqueda anytime . . . 203
Qué significa `proven optimal` . . . 204
Lo bastante bueno . . . 205
9.5. Métricas sobre cualquier fluente . . . 206
Maximizar . . . 208
9.6. Varios deseos en un solo número . . . 209
Un viaje por carretera: peajes y combustible . . . 210
9.7. Un límite o una métrica . . . 212
9.8. El método MODEL: qué hace mejor a un plan . . . 213
Alternativas comparadas . . . 213
9.9. Errores comunes . . . 215
9.10. Resumen . . . 217
9.11. Lecturas complementarias . . . 217
9.12. Ejercicios . . . 218

PARTE IV. TIEMPO: PDDL 2.1 Y 2.2 **221**

10. Acciones durativas **222**
10.1. Lo que una secuencia no puede decir . . . 222
10.2. Una primera acción durativa . . . 224
El mensajero con duraciones fijas . . . 225
Leer un plan temporal . . . 226
10.3. Qué ocurre en cada instante . . . 227
Por qué 10,001 y no 10 . . . 227
El validador está de acuerdo . . . 228
10.4. Condiciones en el tiempo . . . 228
El intervalo es abierto . . . 229
10.5. Duraciones que dependen de la acción . . . 230
10.6. Trabajar en paralelo . . . 232
10.7. Una cosa cada vez . . . 234
10.8. Otros dominios: una cocina y un ascensor . . . 237
Una cocina: cosas que funcionan solas . . . 237
Un ascensor: esperar a las personas . . . 238
10.9. El método MODEL: tiempo . . . 240
Alternativas comparadas . . . 241
10.10. Errores comunes . . . 243
10.11. Resumen . . . 244
10.12. Lecturas complementarias . . . 245
10.13. Ejercicios . . . 245

11. Coordinación en el tiempo **249**
11.1. Planes que solo existen si las acciones se solapan . . . 249
11.2. La acción envolvente: una puerta sujeta . . . 251
11.3. Temporalmente simple, temporalmente expresivo . . . 254
Qué hace Kairos . . . 254
11.4. Trabajar juntos: dos robots y un cajón . . . 255
Un primer modelo, y por qué es frágil . . . 256
Un modelo con holgura . . . 257
11.5. Duraciones que elige el planificador . . . 258
`?duration` en los efectos . . . 259
Cómo elige un planificador una duración . . . 260
Una cafetera . . . 261
11.6. Cómo ordenan el tiempo los planificadores . . . 261
11.7. El método MODEL: coordinación . . . 263
Alternativas comparadas . . . 263
11.8. Errores comunes . . . 265
11.9. Resumen . . . 267
11.10. Lecturas complementarias . . . 267
11.11. Ejercicios . . . 268

12. Cambio continuo **270**
12.1. Una escalera donde el mundo tiene una rampa . . . 270
12.2. Efectos continuos . . . 272
12.3. Leer un número mientras cambia . . . 274
Agua que viene y va: el invernadero . . . 277
12.4. El generador: cuándo empezar . . . 279
12.5. El horno: cuándo parar . . . 282
12.6. Lo que debe encontrar el planificador . . . 284
12.7. El mundo del mensajero: una batería que se descarga y se carga . . . 286
Ensayar el modelo . . . 287
Dónde cargar, y cuánto tiempo . . . 289
12.8. El método MODEL: cambio continuo . . . 290
Alternativas comparadas . . . 291
12.9. Errores comunes . . . 292
12.10. Resumen . . . 293
12.11. Lecturas complementarias . . . 294
12.12. Ejercicios . . . 294

13. Predicados derivados y un mundo con calendario **298**
13.1. Hechos que se siguen de otros hechos . . . 298
13.2. Predicados derivados . . . 299
Dónde puede aparecer un predicado derivado . . . 300
13.3. Reglas que se llaman a sí mismas . . . 301
Dos recursiones más: torres y circuitos . . . 304
Reglas y negación . . . 306
13.4. Qué hace un planificador con las reglas . . . 306
13.5. Un mundo con calendario: literales iniciales temporizados . . . 307
13.6. Ventanas y plazos . . . 309
Plazos . . . 310
Un satélite y sus ventanas . . . 311
13.7. Números en el calendario . . . 313
13.8. El método MODEL: consecuencias y calendario . . . 314
Alternativas comparadas . . . 315
Un plazo: literal temporizado o restricción . . . 316
13.9. Errores comunes . . . 316

13.10. Resumen . . . 317
13.11. Lecturas complementarias . . . 318
13.12. Ejercicios . . . 318

Parte V. Preferencias y restricciones: PDDL 3 **322**

14. Objetivos blandos y preferencias **323**
14.1. Más deseos que batería . . . 323
14.2. Una preferencia en el objetivo . . . 324
14.3. Lo que vale cada deseo . . . 327
Un deseo sobre el propio robot . . . 328
Objetivos estrictos y blandos juntos . . . 328
14.4. Deseos frente a esfuerzo . . . 329
14.5. Una preferencia, muchas instancias . . . 330
14.6. Deseos sobre el camino: preferencias en las precondiciones . . . 331
14.7. Otros dos mundos . . . 333
El día de un turista . . . 333
Un satélite con más peticiones que tiempo . . . 335
14.8. Qué hace el planificador con una preferencia . . . 336
14.9. El método MODEL: los deseos y su valor . . . 338
Alternativas comparadas . . . 338
14.10. Errores comunes . . . 340
14.11. Resumen . . . 341
14.12. Lecturas complementarias . . . 341
14.13. Ejercicios . . . 342

15. Restricciones sobre todo el plan **345**
15.1. Una regla que el plan incumple . . . 345
15.2. Los estados de un plan: always . . . 347
15.3. Lo que debe ocurrir, y en qué orden . . . 348
sometime y at end . . . 348
at-most-once . . . 349
sometime-before y sometime-after . . . 349
15.4. Reglas con reloj . . . 350
Contar pasos . . . 351
Contar segundos . . . 352
15.5. Los diez operadores, uno junto a otro . . . 353
15.6. Dónde viven las reglas, y reglas que ceden . . . 355
En el dominio o en el problema . . . 355
Restricciones como preferencias . . . 355
15.7. Dos mundos más . . . 356
Un róver y un cráter . . . 356
Un taller con un horno . . . 357
15.8. Qué hace el planificador con una restricción . . . 358
15.9. El método MODEL: reglas sobre el plan completo . . . 359
Alternativas comparadas . . . 360
15.10. Errores comunes . . . 361
15.11. Resumen . . . 363
15.12. Lecturas complementarias . . . 363
15.13. Ejercicios . . . 364

16. Fluentes de objeto **367**
16.1. Un robot en dos habitaciones . . . 367
16.2. Una función cuyo valor es un objeto . . . 368

16.3. Fluentes como términos . . . 370
16.4. Sin valor: indefinido . . . 371
16.5. Dos mundos más . . . 372
Bloques: un fluente en lugar de cuatro predicados . . . 372
Colorear un mapa . . . 373
16.6. Variables de estado: lo que ve el planificador . . . 374
16.7. El método MODEL: un valor o muchos . . . 376
Alternativas comparadas . . . 376
16.8. PDDL 3 de un vistazo . . . 377
16.9. Errores comunes . . . 378
16.10. Resumen . . . 379
16.11. Lecturas complementarias . . . 379
16.12. Ejercicios . . . 379

Parte VI. Sistemas híbridos: PDDL+ **382**

17. Procesos y eventos **383**
17.1. Un mundo que no espera . . . 383
17.2. Procesos: cambio mientras se cumple una condición . . . 384
El primer plan . . . 386
El plan no es único, y el instante tampoco . . . 387
17.3. Eventos: cambio en un instante . . . 387
Un evento debe desactivarse a sí mismo . . . 388
Un problema sin plan . . . 388
17.4. Tres clases de cambio . . . 389
La pelota que rebota . . . 390
El coche . . . 392
17.5. El mundo del mensajero en PDDL+ . . . 394
Un desplazamiento es una acción, un proceso y un evento . . . 394
La batería: un proceso y un evento . . . 395
Ensayar el modelo . . . 395
La prueba de la trampa: un robot que avanza por inercia . . . 396
Una muestra que se calienta . . . 397
17.6. Cómo planifica Kairos con PDDL+ . . . 399
Búsqueda en pasos de tiempo . . . 399
Validación en tiempo continuo . . . 400
Lo que se le puede escapar . . . 400
17.7. El método MODEL: quién causa cada cambio . . . 401
Alternativas comparadas . . . 402
17.8. Errores comunes . . . 403
17.9. Resumen . . . 404
17.10. Lecturas complementarias . . . 405
17.11. Ejercicios . . . 405

18. Modelar sistemas híbridos **408**
18.1. Reglas que se cumplen en todo momento: restricciones globales . . . 408
18.2. Cambio que no es lineal . . . 410
Un coche con resistencia del aire . . . 411
18.3. Procesos que interactúan . . . 412
Un dron que aterriza . . . 414
18.4. Eventos en cascada y eventos que no paran . . . 416
Eventos como dispositivos: el termostato . . . 417
Eventos que no paran . . . 418

18.5. Procesos con acciones durativas y literales temporizados . . . 420
Esperar no sale gratis . . . 421
18.6. Un sistema, tres modelos . . . 423
Lo que cada nivel no puede decir . . . 424
18.7. Límites: pasos y tolerancias . . . 425
Un evento no rescata a la búsqueda . . . 427
Valores exactos . . . 428
18.8. El método MODEL: probar y elegir modelos híbridos . . . 428
Alternativas comparadas . . . 429
18.9. Errores comunes . . . 430
18.10. Resumen . . . 431
18.11. Lecturas complementarias . . . 431
18.12. Ejercicios . . . 432

Parte VII. El método en acción **435**

19. El método MODEL, completo **436**
19.1. El método en una página . . . 436
19.2. Las cinco listas de comprobación . . . 438
19.3. Qué característica para qué necesidad . . . 441
Lo que cuesta cada característica . . . 443
19.4. El catálogo de patrones . . . 444
19.5. El catálogo de antipatrones . . . 446
19.6. Depurar, más allá de lo clásico . . . 448
Primero, leer el veredicto . . . 448
Sin plan, con demostración . . . 449
Sin plan, sin demostración . . . 449
Un plan erróneo, y uno pobre . . . 450
19.7. Hacer crecer un modelo por versiones . . . 451
Versión 1: clásica . . . 452
Versión 2: números . . . 454
Versión 3: tiempo . . . 456
Versión 4: deseos . . . 458
Versión 5: procesos . . . 459
Cinco versiones, una al lado de otra . . . 462
19.8. Documentar un modelo . . . 462
19.9. El método MODEL: el método completo . . . 464
Alternativas comparadas . . . 464
19.10. Errores comunes . . . 466
19.11. Resumen . . . 467
19.12. Lecturas complementarias . . . 468
19.13. Ejercicios . . . 468

20. Caso de estudio: una misión robótica, de la historia a la ejecución **472**
20.1. La historia . . . 472
20.2. Paso M: mapear la situación . . . 473
Un plan para el modelado . . . 474
20.3. Versión 1: quién lleva qué . . . 475
Pasos O y D . . . 475
Paso E: la batería de pruebas . . . 477
Paso L: tres maneras de escribir la regla del laboratorio . . . 478
Lo que la versión 1 no puede decir . . . 479
20.4. Versión 2: tiempo y batería . . . 480
Pasos O y D: lo que cambia . . . 480

Paso E: la ronda en el tiempo . . . 481
Paso L: cuánto dura una carga . . . 483
Lo que la versión 2 no puede decir . . . 485
20.5. Versión 3: un calendario y deseos . . . 485
Pasos O y D: lo que cambia . . . 485
Paso E, primera parte: las pruebas que pasan . . . 486
Paso E, segunda parte: la ronda, y un límite . . . 487
Paso L: tres maneras de planificar los deseos . . . 488
Lo que la versión 3 no puede decir . . . 491
20.6. Versión 4: la batería es cosa del mundo . . . 491
Pasos O y D: lo que cambia . . . 491
Paso E: un robot y una persiana . . . 492
Paso L: ¿necesita la ronda la versión 4? . . . 495
20.7. Del plan a la ejecución . . . 496
Qué es un plan para un robot . . . 496
Qué puede salir mal . . . 496
Planificar de nuevo . . . 497
20.8. PlanSys2 . . . 498
Cuatro nodos . . . 498
Un plan se convierte en un árbol de comportamiento . . . 499
Ejecutantes de acciones . . . 500
La aplicación . . . 500
Lo que esto pide al modelo . . . 501
20.9. Lo que PDDL deja fuera . . . 502
20.10. El método MODEL: una historia, cuatro modelos . . . 502
Alternativas comparadas . . . 503
20.11. Errores comunes . . . 504
20.12. Resumen . . . 505
20.13. Lecturas complementarias . . . 506
20.14. Ejercicios . . . 507

Apéndices **509**

A. Kairos **510**
A.1. Instalación . . . 510
Con pip . . . 510
Desde el código fuente . . . 510
En un espacio de trabajo de colcon . . . 510
A.2. Las órdenes . . . 511
A.3. Planificar . . . 511
Opciones . . . 512
Motores y búsquedas . . . 512
A.4. Formatos de plan . . . 513
A.5. Estado y códigos de salida . . . 514
A.6. Comprobar un modelo . . . 514
A.7. Validar un plan . . . 515
A.8. Mirar por dentro . . . 516
A.9. Desde Python . . . 517

B. Referencia rápida de PDDL **518**
B.1. Requisitos . . . 518
B.2. Plantillas de sintaxis . . . 519
Clásico: PDDL 1.2 . . . 519
Numérico: PDDL 2.1, nivel 2 . . . 521

Temporal: PDDL 2.1, niveles 3 y 4, y PDDL 2.2 522
Preferencias, restricciones y fluentes de objeto: PDDL 3 524
Híbrido: PDDL+ 525
B.3. Qué admite cada planificador 527

C. Soluciones a los ejercicios **529**
C.1. Soluciones del Capítulo 1 529
C.2. Soluciones del Capítulo 2 533
C.3. Soluciones del Capítulo 3 540
C.4. Soluciones del Capítulo 4 547
C.5. Soluciones del Capítulo 5 553
C.6. Soluciones del Capítulo 6 558
C.7. Soluciones del Capítulo 7 564
C.8. Soluciones del Capítulo 8 569
C.9. Soluciones del Capítulo 9 574
C.10. Soluciones del Capítulo 10 577
C.11. Soluciones del Capítulo 11 583
C.12. Soluciones del Capítulo 12 588
C.13. Soluciones del Capítulo 13 594
C.14. Soluciones del Capítulo 14 598
C.15. Soluciones del Capítulo 15 602
C.16. Soluciones del Capítulo 16 605
C.17. Soluciones del Capítulo 17 609
C.18. Soluciones del Capítulo 18 615
C.19. Soluciones del Capítulo 19 622
C.20. Soluciones del Capítulo 20 628

Bibliografía **636**

Índice alfabético **638**

Cajas y recuadros

A lo largo del libro aparecen cuatro tipos de recuadros que interrumpen el texto principal. El primero lleva el hilo conductor del libro; los otros tres pueden saltarse en una primera lectura.

Caja de método

Recuadro verde que señala un paso del método MODEL, el método de modelado que este libro construye capítulo a capítulo: un paso aplicado al ejemplo del momento, o una pregunta nueva que el capítulo añade al método.

Caja de concepto

Recuadro gris para una digresión autocontenida: una aclaración terminológica, una mirada a cómo tratan los planificadores una característica, o una comparación que amplía el tema sin romper su hilo.

Caja de historia

Recuadro azul que cuenta un episodio real de la historia de la planificación automática: quién hizo qué, cuándo y qué resultó de ello.

Caja de error típico

Recuadro rojo que muestra un error frecuente al modelar: qué aspecto tiene en el PDDL, qué síntoma produce en el planificador y cómo corregirlo.

Parte I. Planificación y modelos

Planificar: decidir antes de actuar | 1.

Un robot mensajero espera en su base de carga, en una planta de hospital. Una enfermera de la sala A pide una caja de medicamentos que está en una estantería de la farmacia. Nadie le dice al robot qué puertas cruzar ni en qué orden hacer las cosas: solo se le dice lo que se quiere. Antes de que giren sus ruedas, el robot tiene que deducir que primero debe ir a la farmacia, coger la caja y solo entonces dirigirse a la sala A, pasando por el vestíbulo y el pasillo, porque no hay otro camino. Deducir eso es *planificar*, y un programa que lo hace para cualquier planta, cualquier robot y cualquier petición es un *planificador*.

Este capítulo explica qué es planificar y qué necesita de nosotros un planificador para funcionar: una descripción del mundo, de lo que se puede hacer en él y de lo que se quiere. Esa descripción se llama modelo, y escribir buenos modelos es el tema de todo el libro. El capítulo presenta también los tres compañeros del resto del texto: PDDL, el lenguaje en el que se escriben los modelos; Kairos, el planificador que los ejecuta; y el método MODEL, cinco pasos para ir de una situación contada con palabras a un modelo que un planificador resuelva bien.

Empezamos con la idea de decidir antes de actuar y con el mundo del mensajero, el ejemplo que crece a lo largo de todos los capítulos. Después vemos cómo se reparten el trabajo un resolutor general y un modelo, qué supuestos hace la forma más sencilla de planificación y qué ofrece la planificación frente a programar el comportamiento a mano, buscar en un espacio de estados programado a mano o aprender. Tras un breve recorrido por los lugares donde se usan planificadores, conocemos PDDL y sus versiones, ejecutamos Kairos por primera vez sobre un modelo que tratamos como una caja negra y cerramos con el método MODEL. Aún no se enseña nada de la sintaxis de PDDL: eso empieza en el Capítulo 2.

1.1 Decidir antes de actuar . . 2
El mundo del mensajero . 3
1.2 Un resolutor general más un modelo 3
Los tres ingredientes: estados, acciones, objetivo 4
1.3 Los supuestos clásicos . . 5
1.4 La planificación y sus alternativas 6
1.5 Dónde se usa la planificación 8
1.6 Dominio, problema y PDDL 9
PDDL y sus versiones . . . 9
1.7 Primer contacto con Kairos 11
El mismo dominio, otro problema 12
1.8 El método MODEL: cinco pasos y cinco criterios . . . 13
Los cinco criterios 14
Alternativas comparadas . 15
1.9 Errores comunes 15
1.10 Resumen 16
1.11 Lecturas complementarias 17
1.12 Ejercicios 18

1.1. Decidir antes de actuar

Algunas decisiones no exigen pensar en el futuro. Un robot que ve un obstáculo a medio metro se detiene: la reacción correcta depende solo de lo que percibe ahora. Otras decisiones son distintas, porque lo que conviene hacer ahora depende de lo que vendrá después. El robot mensajero que está en la base no gana nada acercándose a la sala A, aunque sea allí donde deben acabar los medicamentos: primero tiene que *alejarse* de la sala, hacia la farmacia. El primer paso solo tiene sentido como parte de una secuencia completa.

Planificar es el razonamiento que produce una secuencia así: dada una situación, las cosas que se pueden hacer y un objetivo, encontrar qué acciones llevar a cabo, y en qué orden, para alcanzar el objetivo. El resultado es un **plan**. Para la entrega de los medicamentos, un plan es: ir al vestíbulo, ir a la farmacia, coger la caja, volver al vestíbulo, ir al pasillo, ir a la sala A, dejar la caja. La planificación ocurre *antes* de actuar, sobre una descripción del mundo y no sobre el mundo mismo. El robot imagina las consecuencias de sus acciones y puede descartar mil secuencias malas sin chocar con una sola pared.

Planificar es encontrar, antes de actuar, una secuencia de acciones que lleve de la situación actual a un objetivo. Se trabaja sobre una descripción del mundo, no sobre el mundo.

Tres rasgos distinguen un problema de planificación de otros problemas de computación. Primero, la respuesta es una forma de actuar, no un número ni una clasificación. Segundo, las acciones interactúan: una acción hace posible otra (el robot solo puede coger la caja cuando está en la farmacia) o la hace imposible (si ya lleva una caja, no puede coger una segunda). Tercero, hay demasiadas secuencias para probarlas todas a ciegas, así que un planificador necesita buscar con cierta inteligencia. Cómo lo hace es el tema del Capítulo 3; en este capítulo el planificador es una caja negra.

El mundo del mensajero

Un mismo ejemplo recorre todo el libro, y es hora de presentarlo. El **mundo del mensajero** es una planta de hospital con ocho lugares: un vestíbulo, un pasillo, una farmacia, un laboratorio, dos salas (A y B), una unidad de cuidados intensivos (UCI) y una base de carga. Los lugares están unidos como muestra la Figura 1.1: el vestíbulo da a la farmacia, a la base y al pasillo; el pasillo da al laboratorio y a las dos salas; y a la UCI solo se llega a través de la sala B. Cada conexión se puede cruzar en los dos sentidos.

En esta planta viven uno o dos robots mensajeros, `r1` y `r2`. Su trabajo es transportar artículos: cajas de medicamentos (`meds1`, `meds2`), una muestra de sangre (`sample1`), una bandeja de comida (`meal1`). En su primera versión, un robot puede hacer tres cosas: moverse de un lugar a otro vecino, coger un artículo que esté donde está el robot y dejar el artículo que lleva. Lleva un artículo cada vez.

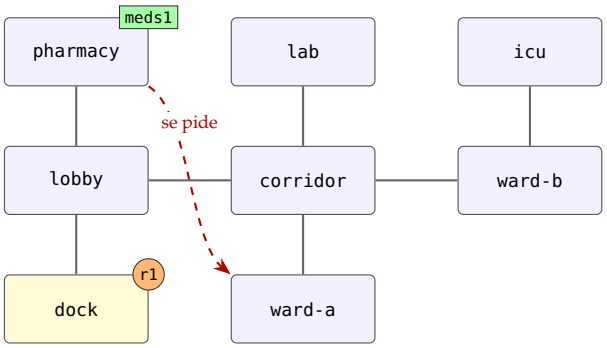


Figura 1.1.: El mundo del mensajero: ocho lugares y las conexiones entre ellos. El robot `r1` descansa en la base, los medicamentos `meds1` están en la farmacia y alguien los quiere en la sala A (flecha discontinua). La flecha es el deseo, no el camino: el camino tiene que ir por las conexiones grises.

El mundo es pequeño a propósito. Cabe en la cabeza, de modo que siempre podemos comprobar a mano lo que responde el planificador. Y tiene sitio para crecer: los capítulos siguientes le añaden puertas y tarjetas de acceso, un carro, una batería que se gasta, distancias y duraciones, horarios de apertura, entregas deseadas pero no obligatorias y, al final, una batería que se descarga sola, actúe el robot o no. Cada añadido necesitará una pieza nueva del lenguaje, y así es como el libro recorre PDDL.

1.2. Un resolutor general más un modelo

¿Cómo podríamos hacer que el robot mensajero entregue los medicamentos? Una forma es escribir un programa que contenga la respuesta:

una función `deliver_medicines_to_ward_a()` con los siete pasos dentro. Funciona hasta que algo cambia: hoy los medicamentos están en el laboratorio, el pasillo está cerrado, hay dos peticiones. Cada cambio necesita un programador.

La planificación sigue otro camino, llamado **autonomía basada en modelos**. El trabajo se divide en dos partes (Figura 1.2):

- Un **modelo**: una descripción del mundo en un lenguaje formal. Dice qué cosas existen, qué puede ser cierto de ellas, qué puede hacer el agente y qué se quiere. No contiene ninguna instrucción sobre *cómo* alcanzar el objetivo.
- Un **planificador**: un programa general que lee cualquier modelo escrito en ese lenguaje y busca un plan. No sabe nada de hospitales, robots ni medicamentos. Sabe buscar.

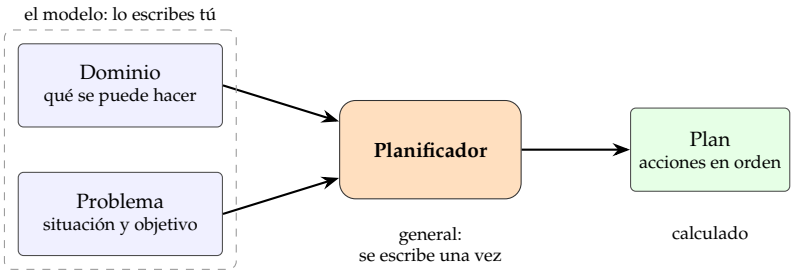


Figura 1.2.: Autonomía basada en modelos. El modelo, en dos ficheros, es la única parte que habla del hospital. El planificador es el mismo programa para todos los modelos. Si cambias el modelo, el mismo planificador da el plan para la nueva situación.

La división compensa en cuanto cambia la situación. Si los medicamentos están en el laboratorio, cambia una línea del modelo y el planificador encuentra el nuevo plan. Si el hospital abre un ala nueva, el modelo gana algunos lugares y conexiones. Si llega una clase de robot nueva, la descripción de lo que puede hacer entra en el modelo. En ninguno de estos casos escribe nadie a mano una nueva secuencia de pasos.

El planificador es general y solo sabe buscar. Todo lo que tiene que ver con la aplicación está en el modelo. Para cambiar el comportamiento, se cambia el modelo.

La división también traslada la dificultad. El planificador lo escriben una vez los especialistas, y nosotros lo damos por hecho. El modelo se escribe para cada aplicación, y lo escribimos nosotros. Un planificador no puede ser mejor que el modelo que recibe: si el modelo dice que el robot puede llevar diez cajas a la vez, el plan apilará diez cajas sobre un robot que tiene una sola bandeja, y el planificador habrá hecho su trabajo a la perfección. Este libro trata de esa mitad del trabajo.

Los tres ingredientes: estados, acciones, objetivo

Sea cual sea el lenguaje, un modelo de planificación tiene tres ingredientes.

Un **estado** es una descripción del mundo en un momento dado, con todo lo que importa para decidir y nada más. En el mundo del mensajero, un estado dice dónde está cada robot, dónde está cada artículo y si cada robot lleva algo. No dice de qué color son las paredes, cuánta carga tiene la batería (todavía no) ni qué tiempo hace. El estado del que parte todo es el **estado inicial**.

Una **acción** es algo que el agente puede hacer y que cambia el estado. Una acción tiene dos partes: la condición bajo la que se puede hacer (su **precondición**) y el cambio que produce (sus **efectos**). El robot puede moverse del vestíbulo al pasillo si está en el vestíbulo y los dos lugares

están conectados; después del movimiento está en el pasillo y ya no está en el vestíbulo.

Un **objetivo** es una condición sobre los estados: lo que debe ser cierto al final. «Los medicamentos están en la sala A» es un objetivo. No describe un estado, sino muchos: no dice nada de dónde acaba el robot, y sirve cualquier estado en el que los medicamentos estén en la sala A.

Con estos tres ingredientes, la tarea del planificador tiene un enunciado exacto. Un plan es una secuencia de acciones tal que la primera se puede hacer en el estado inicial, cada una de las siguientes se puede hacer en el estado que deja la anterior y el estado que deja la última satisface el objetivo. La Figura 1.3 dibuja esa cadena para la entrega de los medicamentos.

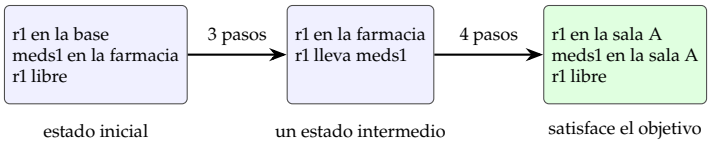


Figura 1.3.: Un plan es una cadena de estados unidos por acciones. Cada caja enumera lo que dice un estado del mundo del mensajero. El último estado satisface el objetivo «`meds1` en la sala A»; que el robot también esté allí es una consecuencia, no algo que pidiera el objetivo.

Quien conozca la búsqueda en grafos reconocerá el dibujo: los estados son nodos, las acciones son aristas y un plan es un camino desde el estado inicial hasta cualquier estado que satisfaga el objetivo. Es correcto, con una diferencia que hace de la planificación un campo propio: nadie dibuja el grafo. El modelo lo describe de forma compacta (un puñado de descripciones de acciones que valen para cualquier robot, artículo y lugar) y el planificador construye, sobre la marcha, la pequeña parte que necesita. Un grafo con más nodos que átomos hay en el universo se puede describir en una página de modelo.

1.3. Los supuestos clásicos

La forma más sencilla de planificación, y aquella por la que empezamos, es la **planificación clásica**. Simplifica el mundo con cinco supuestos. Todos son falsos en un hospital real, y todos hacen la planificación mucho más fácil. Buena parte de la historia de los lenguajes de planificación, y de este libro, consiste en levantarlos uno a uno.

Finito. Hay un número finito de cosas, y un número finito de hechos que pueden ser verdaderos o falsos sobre ellas. Un robot está en el vestíbulo o no lo está; nunca está «a 2,37 metros dentro del vestíbulo». Por tanto, el número de estados es finito, aunque enorme.

Determinista. Una acción aplicada en un estado tiene exactamente un resultado. Cuando el robot coge la caja, la tiene. Nunca se le resbala.

Totalmente conocido. El planificador conoce el estado inicial por completo. No hay cajas cuyo sitio se desconozca, y no hace falta percibir nada mientras se ejecuta el plan.

Estático. El mundo solo cambia cuando el agente actúa. Nadie más mueve los medicamentos, las puertas no se cierran según un horario y las baterías no se descargan solas.

Acciones instantáneas. Una acción no tiene duración. Ocurre en un paso, una acción tras otra, y no tiene sentido preguntar cuánto tarda el plan ni si dos acciones se solapan.

La planificación clásica supone un mundo finito, determinista, totalmente conocido y estático, con acciones instantáneas. Las partes del libro levantan estos supuestos uno a uno.

Con estos supuestos, un plan es una simple secuencia de acciones y su éxito está garantizado: si el modelo es correcto, ejecutar el plan en el mundo lleva al objetivo, sin necesidad de mirar el mundo por el camino.

La Tabla 1.1 muestra qué parte del libro levanta cada supuesto. Dos de ellos, el determinismo y el conocimiento completo, nos acompañan hasta el final: PDDL, en todas las versiones que cubre este libro, describe un mundo cuyo comportamiento es conocido y predecible. Eso limita menos de lo que parece. Un robot cuyo plan falla (la caja se resbala, una puerta está cerrada con llave) puede observar la nueva situación y volver a planificar desde ahí, y el Capítulo 20 muestra cómo se ejecutan y reparan los planes de esa manera.

Tabla 1.1.: Los cinco supuestos clásicos y la parte del libro que levanta cada uno. Las Partes I y II se mantienen dentro de los cinco.

Supuesto	Qué lo levanta	Dónde
Finito	Números: niveles de batería, cargas, distancias	Parte III
Acciones instantáneas	Acciones con duración, que pueden solaparse	Parte IV
Estático	Hechos que cambian en momentos conocidos; después, procesos y eventos que el mundo ejecuta por sí solo	Parte IV (Capítulo 13), Parte VI
Determinista	No se levanta en PDDL: ejecutar, observar y volver a planificar	Capítulo 20
Totalmente conocido	No se levanta en PDDL	Capítulo 20

Un sexto supuesto se esconde en la palabra «objetivo». En la planificación clásica, un objetivo es una condición sobre el último estado, y todos los planes que la alcanzan valen lo mismo. Las peticiones reales son más ricas: la ruta más corta, la menor energía, «si es posible, trae también la comida», «nunca lleves residuos por la UCI». La Parte III añade una medida de calidad a los planes, y la Parte V añade objetivos que son deseos y condiciones sobre todo el plan y no solo sobre su final.

Un modelo equivocado que es útil

Los supuestos clásicos no afirman que los hospitales sean finitos, predecibles e inmóviles. Son una decisión sobre qué dejar fuera. Un modelo del mundo del mensajero en el que las habitaciones son puntos y moverse no lleva tiempo es falso, y aun así le dice al robot en qué orden visitar las habitaciones, que es lo que le pedimos. La pregunta que hay que hacerle a un modelo nunca es «¿es verdadero?», sino «¿es lo bastante bueno para las decisiones que necesitamos de él?». El paso M del método MODEL, al final de este capítulo, hace esa pregunta de forma explícita.

1.4. La planificación y sus alternativas

Planificar es una de varias formas de hacer que una máquina elija sus acciones. Ver las demás una junto a otra muestra cuándo un planificador es la herramienta adecuada.

Programar el comportamiento a mano. El ingeniero decide qué hacer en cada situación y lo escribe: un guion, una máquina de estados, un árbol de comportamiento. Para un conjunto pequeño y estable de situaciones es la mejor opción: es rápido, predecible y fácil de certificar. Su coste aparece con la variedad. Un mensajero con un robot y una sola clase de petición necesita un guion corto. Con dos robots, cuatro artículos, puertas cerradas y baterías bajas, las situaciones se multiplican, y el programa escrito a mano tiene que prever todas las combinaciones. Las que nadie previó son las que fallan de noche.

Búsqueda en un espacio de estados programado a mano. El ingeniero escribe un programa en un lenguaje general: una estructura de datos para el estado, una función que genera los sucesores, una comprobación del objetivo, quizá una heurística, y ejecuta un algoritmo de búsqueda como A*. Esto ya separa el problema de la búsqueda y resuelve cualquier instancia. Pero todo está escrito para un solo problema: la función de sucesores del mensajero no sirve para un almacén, la heurística hay que inventarla cada vez y un cambio en las reglas obliga a cambiar código. Un planificador es esta misma idea llevada un paso más allá: el espacio de estados no se programa, sino que se *declara* en un lenguaje de modelado, de modo que el algoritmo de búsqueda y sus heurísticas, que el planificador obtiene por sí solo a partir del modelo, se reutilizan en todos los problemas.

Aprendizaje. El comportamiento no se escribe en absoluto: se aprende a partir de ejemplos o por ensayo y error. El aprendizaje brilla donde no se puede escribir ningún modelo: reconocer una caja en una imagen, agarrar un objeto blando. Necesita datos o muchos intentos, lo que aprende queda atado a las situaciones que ha visto y rara vez explica o garantiza sus decisiones. Para un problema como el del mensajero, en el que las reglas son conocidas y exactas, un modelo enuncia en una página lo que el aprendizaje necesitaría miles de intentos para aproximar.

Enfoque	Tú aportas	Una situación nueva exige	Punto débil
Comportamiento a mano	Las decisiones	Código nuevo, si no se previó	La variedad
Búsqueda a mano	Estados, sucesores, heurística, en código	Datos nuevos; código nuevo si cambian las reglas	La reutilización
Aprendizaje	Ejemplos o una recompensa	Más datos, si se aleja de lo visto	Las garantías
Planificación	Un modelo	Un fichero de problema nuevo	El modelo tiene que existir

Tabla 1.2.: Cuatro formas de obtener un comportamiento. La planificación pide un modelo y a cambio da flexibilidad: las situaciones nuevas son datos nuevos, no código nuevo.

La planificación encaja en problemas con reglas claras, muchas situaciones posibles y objetivos que cambian. A cambio, pide un modelo.

En un sistema real, los cuatro enfoques no compiten. En un robot ocupan niveles distintos: la percepción aprendida encuentra la caja, un controlador escrito a mano mantiene firme el brazo, un planificador de trayectorias programado para la geometría traza la ruta por la habitación, y un planificador de la clase que estudia este libro decide qué caja ir a buscar primero y adónde llevarla. Este último nivel, donde las decisiones tratan de *qué* hacer y en qué orden, se suele llamar **planificación de tareas**.

Los límites de la planificación nacen del mismo sitio que su fuerza. Necesita un modelo: donde las reglas no se pueden enunciar (cómo

agarrar un vaso mojado), no hay nada con lo que planificar. El plan es tan bueno como el modelo. Y planificar es difícil en el sentido técnico: incluso con los supuestos clásicos la búsqueda puede tardar mucho, y un modelo descuidado la alarga todavía más. Los criterios del método MODEL existen para no perder de vista ninguno de los dos peligros.

1.5. Dónde se usa la planificación

El espacio. Una nave lejos de la Tierra no puede esperar instrucciones: una señal de radio tarda minutos en llegar a Marte, y horas a los planetas exteriores. En mayo de 1999, el experimento Remote Agent dejó que un planificador a bordo tomara el control de la sonda Deep Space 1 de la NASA: a partir de objetivos enviados desde tierra, el software planificó las actividades de la nave y las ejecutó por sí solo. Los planificadores también trabajan en la parte de tierra de las misiones espaciales. Las actividades diarias de los vehículos marcianos Spirit y Opportunity, que aterrizaron en 2004, las preparaban los operadores con ayuda de una herramienta de planificación que mantenía los planes dentro de los límites de tiempo, energía y datos de los vehículos.

Logística. Mover mercancías con una flota de vehículos, cargar y descargar contenedores, asignar grúas y camiones, secuenciar las operaciones de una fábrica: todos ellos son problemas de planificación, y grandes. El dominio llamado *logistics*, con paquetes, camiones, aviones y ciudades, es una prueba habitual para los planificadores desde la primera competición, y lo modelaremos en el Capítulo 4.

Robótica. Un robot de servicio recibe objetivos (trae esto, ordena aquello, guía a un visitante) que cambian continuamente, en un entorno que también cambia. Escribir a mano cada comportamiento no escala, así que un planificador de tareas decide la secuencia de acciones y el resto del software del robot ejecuta cada una. El primer planificador de esta línea se construyó para un robot: STRIPS, escrito en el Stanford Research Institute para un robot móvil llamado Shakey Fikes y Nilsson, 1971. Hoy, entornos como PlanSys2 Martín et al., 2021 conectan planificadores PDDL con robots que funcionan con ROS 2: el modelo dice lo que puede hacer el robot, el planificador decide y cada acción del plan va ligada al fragmento de código que la realiza. Volveremos a ello en el Capítulo 20.

La planificación se usa donde las reglas son conocidas, las situaciones son muchas y los objetivos cambian: naves espaciales, logística, robots de servicio.

Los planificadores se usan también en lugares donde no hay ningún robot: para configurar y reparar sistemas informáticos, componer los pasos de un proceso de negocio, controlar ascensores, generar la secuencia de operaciones de una planta química o encontrar los movimientos de un rompecabezas. Lo que comparten estos usos es la forma del problema: reglas conocidas, muchas situaciones posibles y un objetivo que cambia de un día para otro.

STRIPS y un robot llamado Shakey

En 1971, Richard Fikes y Nils Nilsson, del Stanford Research Institute, publicaron STRIPS, el *Stanford Research Institute Problem Solver* Fikes y Nilsson, 1971. Era el planificador de Shakey, un robot móvil que empujaba cajas entre habitaciones. STRIPS describía cada acción por los hechos que necesita, los hechos que hace verdaderos y los hechos que hace falsos. El planificador en sí pronto quedó superado, pero

esa forma de describir acciones sobrevivió: más de cincuenta años después sigue siendo el núcleo de PDDL, y el Capítulo 2 la enseña con el mismo nombre.

1.6. Dominio, problema y PDDL

La Figura 1.2 mostraba el modelo en dos ficheros. Esa división es la primera idea de modelado del libro, así que merece una mirada atenta.

El **dominio** describe una clase de mundo: qué se puede decir de él y qué se puede hacer en él. El dominio del mundo del mensajero dice que los robots están en lugares, que los artículos están en lugares o los llevan robots, y que un robot puede moverse, coger y dejar, con la condición y el efecto de cada acción. No menciona ningún robot, habitación o artículo concreto.

El **problema** describe una situación de esa clase de mundo y una petición: qué robots, artículos y lugares existen, cómo están conectados los lugares, dónde está cada cosa al principio y qué se quiere. «El robot `r1` está en la base, `meds1` está en la farmacia, queremos `meds1` en la sala A» es un problema.

Un dominio sirve para muchos problemas. El hospital escribe el dominio del mensajero una vez; cada petición de cada día es un problema nuevo, que en la práctica genera un programa a partir de la base de datos del hospital. Hasta el mapa pertenece al problema: el mismo dominio sirve para otra planta o para otro hospital.

El dominio contiene las reglas de una clase de mundo: qué puede ser cierto y qué se puede hacer. El problema contiene una situación y un objetivo. Un dominio, muchos problemas.

Para dar una idea del lenguaje, el Listado 1.1 muestra cómo describe el dominio del mensajero la acción de moverse. Todavía no hace falta entender la notación, solo fijarse en su forma: un nombre, unos parámetros, una condición y un efecto, escritos con muchos paréntesis. Las palabras que empiezan por un signo de interrogación son variables: la misma descripción sirve para cualquier robot y cualquier par de lugares conectados.

```
(:action move
  :parameters (?r ?from ?to)
  :precondition (and (robot ?r) (location ?from) (location ?to)
                     (robot-at ?r ?from) (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

Listado 1.1: La acción `move` del dominio del mensajero, mostrada solo para dar una idea del lenguaje: el Capítulo 2 explica cada símbolo.

PDDL y sus versiones

El lenguaje de ese listado es **PDDL**, el *Planning Domain Definition Language*. Hasta 1998, casi cada planificador leía su propio lenguaje de entrada, así que comparar dos planificadores obligaba a reescribir todos los problemas de prueba. PDDL lo crearon Drew McDermott y un comité de investigadores McDermott et al., 1998 como lenguaje común de la primera competición de planificación, celebrada ese año, y desde entonces es el lenguaje estándar del campo. Con un lenguaje estándar, un modelo escrito una vez se le puede dar a cualquier planificador, y un planificador nuevo se puede probar en el acto sobre miles de modelos ya existentes.

PDDL ha crecido por versiones. Cada una añadió a la anterior los medios para describir una clase de mundo más rica, y las versiones se corresponden con las partes de este libro (Tabla 1.3 y Figura 1.4).

Versión	Año	Qué añade	Capítulos
PDDL 1.2	1998	Acciones con precondiciones y efectos; tipos; condiciones más ricas (negación, *o*, *para todo*, *existe*) y efectos que dependen de la situación	2 a 7
PDDL 2.1	2002	Números; una medida de la calidad del plan; acciones con duración; cambio continuo	8 a 12
PDDL 2.2	2004	Hechos que se derivan de otros mediante reglas; hechos que cambian en momentos conocidos	13
PDDL 3.0	2006	Preferencias (objetivos que son deseos); restricciones sobre todo el plan	14, 15
PDDL 3.1	2008	Variables de estado cuyo valor es un objeto; costes de acción	16, 9
PDDL+	2006	Procesos y eventos: un mundo que cambia por sí solo	17, 18

Tabla 1.3.: La familia de versiones de PDDL. Cada versión contiene a las anteriores; PDDL+ es una rama que sale de PDDL 2.1. El año es el de la competición o la publicación que introdujo la versión.



Figura 1.4.: Cronología de PDDL. La línea principal añade, versión a versión, lo que los supuestos clásicos dejan fuera. PDDL+ se separa de PDDL 2.1 hacia mundos que cambian de forma continua y por sí solos.

PDDL 1.2 es el lenguaje de la planificación clásica: todo en él respeta los cinco supuestos. PDDL 2.1 Fox y Long, 2003 levantó dos de golpe, con los números (el mundo deja de ser finito) y con acciones que duran y se solapan. PDDL 2.2 Edelkamp y Hoffmann, 2004 y PDDL 3.0 A. Gerevini y Long, 2005 enriquecieron el mundo y las peticiones: hechos que se siguen de otros hechos, hechos que cambian en momentos fijos, deseos, reglas para todo el plan. PDDL 3.1 Kovacs, 2011 añadió una forma más compacta de escribir el estado. PDDL+ Fox y Long, 2006 levantó por completo el supuesto estático: en él, el mundo tiene su propia dinámica, que el agente solo puede iniciar, detener o esperar.

Ningún planificador está obligado a admitirlo todo. Cada modelo declara qué características del lenguaje usa (sus **requisitos**), y cada planificador documenta cuáles acepta. La mayoría de los planificadores cubren una parte de la familia: muchos son puramente clásicos, algunos son numéricos, otros temporales y unos pocos manejan PDDL+. Esto importa al modelar, y es la razón por la que la *portabilidad* es uno de los cinco criterios con los que juzgaremos un modelo.

La Competición Internacional de Planificación

La primera competición de planificación la organizó Drew McDermott en la conferencia AIPS de 1998 McDermott, 2000, y PDDL se escribió para que todos los participantes pudieran leer los mismos problemas. La competición, conocida como IPC, se ha vuelto a celebrar en 2000, 2002, 2004, 2006, 2008, 2011, 2014, 2018 y 2023. Varias ediciones dieron forma al lenguaje: la de 2002 introdujo PDDL 2.1; la de 2004, PDDL 2.2; la de 2006, PDDL 3.0; y la de 2008, PDDL 3.1. Las

competiciones dejaron además una gran colección pública de dominios y problemas, con la que se siguen probando los planificadores hoy en día.

1.7. Primer contacto con Kairos

El planificador de este libro es Kairos Martín Rico et al., 2026, un planificador de código abierto que lee todas las versiones de PDDL de la lista anterior. Lo usamos desde la línea de órdenes. El Apéndice A explica las formas de instalarlo; la más corta es el paquete de Python:

```
$ pip install kairos-planner
$ kairos --version
kairos 1.0.6
```

La carpeta `examples/ch01` del libro contiene el modelo del mensajero en dos ficheros: `courier-domain.pddl`, el dominio, y `courier-p01.pddl`, el problema de la Figura 1.1. En este capítulo no los abrimos. Se los damos al planificador, primero el dominio y después el problema:

```
$ kairos courier-domain.pddl courier-p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Eso es un plan, y ha llegado en unos pocos milisegundos. Merece la pena aprender a leerlo ya, porque leeremos cientos.

Cada línea entre paréntesis es un **paso** del plan: el nombre de una acción seguido de los objetos a los que se aplica. La línea `(move r1 dock lobby)` dice: el robot `r1` se mueve de la base al vestíbulo. El orden de los objetos es el orden de los parámetros de la acción del Listado 1.1: `?r`, `?from`, `?to`. Los pasos se ejecutan de arriba abajo. La primera línea, que empieza por punto y coma, es un comentario que escribe el planificador: este plan tiene coste 7, y en un modelo que no dice nada de costes el coste de un plan es su número de pasos.

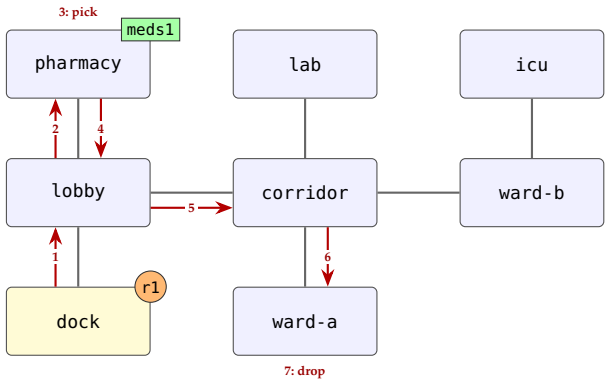


Figura 1.5.: El plan que encuentra Kairos, dibujado sobre el mapa. Sigue los números: los pasos 1 y 2 alejan al robot de la sala A, porque primero hay que coger los medicamentos (paso 3). Los pasos 4 a 6 los llevan a la sala, donde el paso 7 los deja.

La Figura 1.5 traza el plan sobre el mapa. Comprobarlo a mano lleva un minuto: cada movimiento une dos lugares conectados y empieza donde acabó el anterior, el robot está en la farmacia cuando coge la caja y la lleva cuando la deja en la sala A. Nadie le habló al planificador de farmacias; encontró el rodeo por la farmacia porque el modelo dice que un robot solo coge un artículo donde está el artículo.

El planificador también puede comprobar un plan por nosotros. Kairos tiene una orden para ello, `validate`, que recibe el dominio, el problema y un fichero con un plan. El fichero `courier-p01.plan` contiene los siete pasos anteriores:

```
$ kairos validate courier-domain.pddl courier-p01.pddl \
    courier-p01.plan
plan valid, makespan 7
```

Validar un plan significa simularlo sobre el modelo, paso a paso, y comprobar que cada paso se puede hacer cuando le llega el turno y que el objetivo se cumple al final. La palabra *makespan* designa el tiempo que tarda el plan; con acciones instantáneas, Kairos cuenta una unidad por paso. A partir de ahora, todos los planes impresos en el libro han pasado esta comprobación.

El mismo dominio, otro problema

El fichero `courier-p02.pddl` describe otro día en la misma planta. El robot está en la sala A, una muestra de sangre espera en la UCI y el laboratorio la necesita. El fichero de dominio es el mismo:

```
$ kairos courier-domain.pddl courier-p02.pddl
; cost = 8
(move r1 ward-a corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(pick r1 sample1 icu)
(move r1 icu ward-b)
(move r1 ward-b corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

Otros objetos, otro punto de partida, otro objetivo, y un plan nuevo sin tocar el planificador ni el dominio. Es la promesa de la Figura 1.2, cumplida.

Un tercer problema, `courier-p03.pddl`, vuelve a pedir los medicamentos en la sala A, pero un día en que la sala está cerrada por limpieza: el fichero del problema no incluye la conexión entre el pasillo y la sala.

```
$ kairos courier-domain.pddl courier-p03.pddl
note: the goal needs atoms that no action can reach
(item-at meds1 ward-a) can never be reached:
  drop adds it, but needs (robot-at ?r - object ward-a), which never
      holds
    move adds it, but needs (connected ?from - object ward-a), which
      never holds
      no action adds it, and the initial state does not have it [
     orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

No hay plan, y Kairos lo dice con la palabra **unsolvable** («irresoluble»). Es una respuesta, y valiosa: el planificador no se ha rendido, ha establecido que ninguna secuencia de acciones, por larga que sea, entrega los medicamentos. La nota que precede al veredicto incluso explica por qué, con el vocabulario del modelo: para que los medicamentos estén en la sala A alguien tiene que dejarlos allí, para eso un robot tiene que estar en la sala A, para eso tiene que moverse hasta allí, y nada está conectado con la sala A. Leer explicaciones así se volverá rutina en el Capítulo 2.

Un planificador da una de dos respuestas útiles: un plan, que se puede validar, o la afirmación de que no existe ningún plan. Las dos salen del modelo, no del mundo.

1.8. El método MODEL: cinco pasos y cinco criterios

Ejecutar el planificador era la parte fácil. Alguien escribió el fichero de dominio y, al hacerlo, tomó decisiones: que la planta son ocho lugares y no una cuadrícula de metros cuadrados; que «llevar» es algo que hay que registrar y «cansado» no; que coger exige que el robot y el artículo estén en el mismo sitio. Cada decisión podría haberse tomado de otra manera. Escribir un modelo no es traducir una historia a paréntesis. *Modelar es elegir*, y este libro enseña un método para elegir bien.

El método se llama MODEL por las iniciales de sus cinco pasos (Figura 1.6). Cada paso responde a una pregunta y deja algo por escrito.

M: Mapear la situación. ¿Cuál es el problema, quién actúa, qué cuenta como resuelto, qué se deja fuera? El paso produce la **ficha del modelo**, unas pocas líneas con cinco apartados: la *Historia* contada con palabras sencillas, los *Agentes* que actúan, qué significa *Éxito*, el *Alcance* (qué entra en el modelo y qué queda fuera) y los *Supuestos* que se hacen sobre el mundo. En este paso no se escribe nada de PDDL. La mayoría de los malos modelos fallan aquí: resuelven otro problema.

O: Objetos y estado. ¿Qué cosas existen, y qué tiene que saber de ellas el planificador para decidir? El paso produce la **tabla de estado**: cada propiedad que registrará el modelo, de qué trata, si puede cambiar y cómo se escribe.

D: Dinámica. ¿Qué se puede hacer o qué puede ocurrir, cuándo, y qué cambia? El paso produce la **tabla de acciones**: cada acción, sobre qué actúa, cuándo se puede hacer y qué cambia.

E: Ensayar el modelo. ¿Hace el planificador con el modelo lo que el mundo permitiría, y nada más? Un modelo es un programa y tiene errores. El paso produce la **batería de pruebas**: un problema trivial, uno típico, uno que no debe tener plan y otro que tienta al planificador a hacer trampa. Cada plan se lee y se valida.

L: Lectura crítica. ¿Qué otra codificación serviría, y cuál es mejor aquí? El paso produce la **comparativa**: dos o tres alternativas juzgadas con cinco criterios, y una decisión con su motivo.

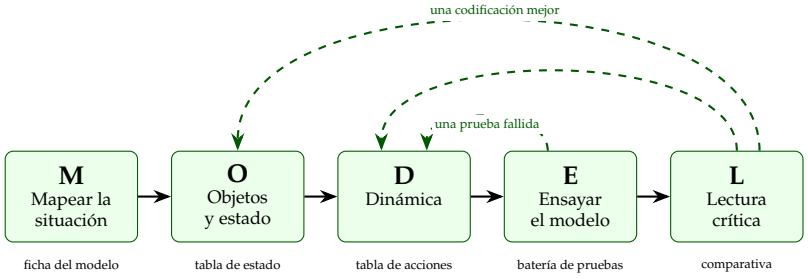


Figura 1.6.: El método MODEL. Flechas continuas: los cinco pasos en orden, cada uno con lo que produce. Flechas discontinuas: el método es un bucle. Una prueba fallida devuelve a la dinámica; una alternativa mejor encontrada en el paso L devuelve al estado o a la dinámica.

Los tres primeros pasos van del mundo al modelo: decidir cuál es el problema, después qué existe y después qué ocurre. Los dos últimos van del modelo de vuelta al mundo: comprobar que el modelo se comporta como el mundo y preguntarse si otro modelo se comportaría mejor. El método es un bucle. Una prueba que falla en el paso E nos devuelve al paso D, y la comparativa del paso L nos devuelve a menudo al paso O con una idea mejor de cómo representar el estado.

MODEL: Mapear la situación, Objetos y estado, Dinámica, Ensayar el modelo, Lectura crítica. Cada paso responde a una pregunta y deja un resultado por escrito.

El paso M en el mundo del mensajero: una primera ficha del modelo

Esta es la ficha del modelo que hay detrás de los ficheros que hemos ejecutado en este capítulo.

Historia	Un robot lleva artículos entre las habitaciones de una planta de hospital, a petición.
Agentes	Un robot mensajero. Es el único que actúa.
Éxito	Cada artículo pedido está en la habitación donde se pidió.
Alcance	Dentro: las habitaciones y sus conexiones; dónde están el robot y los artículos; qué lleva el robot. Fuera: distancias, tiempo, batería, puertas, personas, cómo agarra el brazo.
Supuestos	Los cinco clásicos. El robot lleva un artículo cada vez.

Cada línea del Alcance es una decisión. «Fuera: batería» significa que este modelo mandará sin reparos al robot a hacer un recorrido que la agote. Si eso importa, la ficha tiene que cambiar, y el modelo con ella: el Capítulo 8 hace exactamente eso.

Los cinco criterios

El paso L necesita una forma de decir que un modelo es mejor que otro. El método usa cinco criterios, siempre los mismos y siempre en este orden.

1. **Fidelidad**. El modelo no tiene ningún plan imposible: todo plan del modelo se puede llevar a cabo en el mundo. Y no tiene ninguna solución perdida: toda forma razonable de resolver el problema es un plan del modelo. Un modelo que deja al robot llevar diez cajas falla en la primera mitad; un modelo que prohíbe al robot cruzar dos veces el vestíbulo falla en la segunda.
2. **Tamaño**. Lo grande que se vuelve el modelo para el planificador a medida que crece el problema: cuántos hechos y acciones concretos resultan cuando las descripciones del dominio se aplican a todos los objetos del problema.

3. **Guía**. Cuánto facilita el modelo la búsqueda: si tiene callejones sin salida, muchas formas equivalentes de hacer lo mismo o rodeos largos, y cuánto del problema pueden ver las estimaciones del planificador.
4. **Claridad**. Lo fácil que es leer el modelo, cambiarlo y explicárselo a otra persona, incluido tú mismo dentro de seis meses.
5. **Portabilidad**. Qué planificadores aceptan las características del lenguaje que usa el modelo.

La fidelidad va primero porque un modelo sin ella da respuestas equivocadas muy deprisa. Los demás se compensan unos con otros: un modelo compacto puede ser más difícil de leer, y uno fiel puede necesitar una característica que admiten pocos planificadores. Ninguna codificación gana en los cinco en todos los casos, y por eso el último paso termina con una decisión *y su motivo*.

Alternativas comparadas

A partir de ahora, cada capítulo termina su sección del método con una tabla como la Tabla 1.4: dos o tres codificaciones de lo mismo, escritas, ejecutadas y juzgadas con los cinco criterios. Todavía no sabemos escribir PDDL, así que esta primera comparativa se hace con palabras, sobre una decisión que la ficha del modelo ya tomó: cómo describir *dónde* están las cosas. La codificación A es la de este capítulo: la planta son ocho lugares. La codificación B divide la planta en celdas cuadradas de un metro; para una planta de 40 por 25 metros eso son mil celdas, y el robot se mueve de una celda a otra vecina.

Criterio	A: ocho lugares	B: mil celdas
Fidelidad	Los planes dicen qué habitaciones cruzar, no cómo; el camino dentro de una habitación queda para la navegación del robot	Los planes son rutas exactas; dejan de valer en cuanto se mueve una silla
Tamaño	8 lugares, 14 movimientos posibles	1000 celdas, unos 4000 movimientos posibles
Guía	Una entrega son 7 pasos	La misma entrega son decenas de pasos, con muchísimas rutas equivalentes
Claridad	Los lugares tienen los nombres que usa la gente	La celda (17, 4) no le dice nada a una enfermera
Portabilidad	Clásica	Clásica

Tabla 1.4.: Una primera comparativa, con palabras: dos formas de describir dónde están las cosas en el mundo del mensajero.

Veredicto. Para decidir qué artículo ir a buscar y en qué orden visitar las habitaciones, la codificación A es mejor en todos los criterios en los que difieren. B responde a otra pregunta, cómo cruzar una habitación, y esa pregunta ya tiene mejores herramientas: el software de navegación del robot. Elige A cuando el planificador decide *qué* hacer, y deja el nivel de detalle de B al componente que sabe de geometría. La lección general es la de la ficha del modelo: el nivel de detalle adecuado es el más bajo que todavía distingue las decisiones que necesitas.

1.9. Errores comunes

Los errores de un primer contacto con un planificador tienen menos que ver con PDDL que con lo que cabe esperar de la herramienta.

Pasar los ficheros en el orden equivocado

Kairos espera primero el dominio y después el problema. Con los dos ficheros intercambiados, lee el problema donde espera un dominio y se detiene en la segunda línea:

```
$ kairos courier-p01.pddl courier-domain.pddl
courier-p01.pddl:2:10: error: expected 'domain', found name '
     problem' [parse.unexpected_token]
 2 | (define (problem courier-p01)
   |          ^
```

El mensaje tiene la forma de todos los diagnósticos de Kairos: fichero, línea y columna, qué está mal, un código entre corchetes y la línea con una marca debajo del lugar del error. Lee primero el primer error.

Tomar «unsolvable» por un fallo del planificador

Cuando la respuesta es `kairos: unsolvable (exhausted)`, como con `courier-p03.pddl`, el planificador no se ha estropeado, y ejecutarlo otra vez o durante más tiempo no cambia nada: el modelo no tiene plan. O la situación de verdad no tiene solución (la sala está cerrada), o el modelo está mal (alguien olvidó escribir una conexión). La nota que Kairos imprime antes del veredicto dice qué hecho no se puede alcanzar; empieza por ahí.

Esperar que el planificador sepa lo que el modelo no dice

El plan de `courier-p01.pddl` termina con el robot parado en la sala A. Una persona lo mandaría de vuelta a su base; el planificador no lo hace, porque el objetivo solo pide los medicamentos en la sala A. El planificador no tiene sentido común ni sabe nada de hospitales: lo que no está en el modelo no existe. Si el robot tiene que acabar en la base, el objetivo tiene que decirlo.

Tratar un plan como un programa

Un plan resuelve un problema: es una secuencia ajustada a un estado inicial y a un objetivo. El plan del primer problema no sirve para el segundo, y el validador lo rechaza; aquí se detiene en el primer paso que nombra un objeto que el segundo problema no tiene:

```
$ kairos validate courier-domain.pddl courier-p02.pddl \
    courier-p01.plan
courier-p01.plan:4: error: step 3 (pick r1 meds1 pharmacy) at
     2.000: unknown object 'meds1' [validate.unknown_object]
 4 | (pick r1 meds1 pharmacy)
plan invalid
```

Lo que se reutiliza de un día para otro es el dominio, nunca el plan. Una situación nueva significa volver a ejecutar el planificador.

1.10. Resumen

Planificar es decidir, antes de actuar, qué acciones llevar a cabo y en qué orden para alcanzar un objetivo. Un planificador es un programa general que hace esto para cualquier modelo que se le dé; el modelo es la parte

que escribimos nosotros, y su calidad decide la calidad de los planes. Las ideas que hay que quedarse de este capítulo:

- Un modelo tiene tres ingredientes: **estados** (entre ellos, el inicial), **acciones** con precondiciones y efectos, y un **objetivo**. Un plan es una secuencia de acciones que lleva del estado inicial a un estado que satisface el objetivo.
- La **planificación clásica** supone un mundo finito, determinista, totalmente conocido y estático, con acciones instantáneas. Las Partes III a VI del libro levantan el primero, los dos últimos y los límites de lo que puede ser un objetivo.
- La planificación encaja en problemas con reglas claras y situaciones cambiantes. El comportamiento escrito a mano, la búsqueda programada a mano y el aprendizaje encajan en otros, y los sistemas reales los combinan.
- Un modelo viene en dos ficheros: el **dominio** (una clase de mundo) y el **problema** (una situación y un objetivo). Un dominio sirve para muchos problemas.
- **PDDL** es el lenguaje estándar para los modelos, nacido en 1998 con la competición de planificación. Sus versiones (1.2, 2.1, 2.2, 3.0, 3.1 y PDDL+) son la ruta de este libro.
- **Kairos** recibe un dominio y un problema y responde con un plan o con «unsolvable». También valida planes.
- Modelar es elegir. El **método MODEL** organiza las elecciones en cinco pasos (Mapear la situación, Objetos y estado, Dinámica, Ensayar el modelo, Lectura crítica) y juzga las alternativas con cinco criterios (fidelidad, tamaño, guía, claridad, portabilidad).

El próximo capítulo abre los dos ficheros que aquí hemos tratado como una caja negra, enseña el PDDL que contienen y aplica por primera vez los cinco pasos del método al mundo del mensajero.

1.11. Lecturas complementarias

La visión de la planificación como un resolutor general más un modelo, y su comparación con la programación y con el aprendizaje, sigue el primer capítulo de Geffner y Bonet Geffner y Bonet, 2013, un libro breve que después explica cómo buscan los planificadores. El libro de texto de referencia del campo es el de Ghallab, Nau y Traverso Ghallab et al., 2004; su primer capítulo discute uno a uno los supuestos clásicos (allí llamados supuestos restrictivos). El segundo libro de los mismos autores Ghallab et al., 2016 trata de lo que este capítulo dejó para el Capítulo 20: cómo se conecta la planificación con la actuación. Russell y Norvig Russell y Norvig, 2020 sitúan la planificación dentro de la inteligencia artificial en su conjunto, junto a la búsqueda y el aprendizaje.

Sobre el lenguaje: el manual original de PDDL McDermott et al., 1998 y el relato de McDermott sobre la primera competición McDermott, 2000 cuentan por qué PDDL se hizo como se hizo. Haslum, Lipovetzky, Magazzeni y Muise Haslum et al., 2019 ofrecen una referencia compacta de todo el lenguaje. El artículo con el que empezó todo, de Fikes y Nilsson Fikes y Nilsson, 1971, todavía se lee bien hoy.

1.12. Ejercicios

Ejercicios de la Sección 1.1 — Decidir antes de actuar

Ejercicio 1.1.1. Para cada tarea, di si resolverla exige planificar tal como se define en este capítulo (una secuencia de acciones que interactúan, encontrada antes de actuar) y por qué: (a) un termostato que enciende la calefacción por debajo de 19 grados; (b) sacar un coche de un aparcamiento abarrotado en el que antes hay que mover otros coches; (c) ordenar una lista de números; (d) hacer una maleta de modo que lo que se va a necesitar primero quede arriba; (e) frenar cuando frena el coche de delante.

Ejercicio 1.1.2. Con el mapa de la Figura 1.1 y solo papel: el robot está en la base, `meds1` está en la farmacia y `sample1` está en la UCI. Los medicamentos tienen que ir a la sala B y la muestra al laboratorio. Escribe un plan, como una lista de pasos (moverse, coger, dejar), y cuenta sus pasos. Después escribe el plan que hace las dos entregas en el otro orden, vuelve a contar y explica por qué un orden es mucho mejor que el otro.

Ejercicios de la Sección 1.2 — Un resolutor general más un modelo

Ejercicio 1.2.1. Describe con palabras los tres ingredientes (qué dice un estado, dos o tres acciones con su condición y su cambio, y un objetivo) para: (a) preparar una taza de té; (b) un ascensor que atiende las llamadas de un edificio; (c) el 8-puzle, con ocho fichas deslizantes en un marco de 3 por 3.

Ejercicio 1.2.2. El hospital del mundo del mensajero hace cinco cambios. Para cada uno, di qué tiene que cambiar: el planificador, la descripción de lo que puede hacer el robot o la descripción de la situación: (a) se abre una nueva sala C junto al pasillo; (b) la petición de hoy es una comida para la UCI; (c) los robots reciben una segunda bandeja y pueden llevar dos artículos; (d) se compra un segundo robot; (e) el hospital quiere que los planes se calculen el doble de rápido.

Ejercicio 1.2.3. El objetivo «`meds1` está en la sala A» lo satisfacen muchos estados. Con un robot, un artículo y los ocho lugares del mapa, ¿cuántos estados distintos lo satisfacen? (Un estado dice dónde está el robot y dónde está el artículo, o que el robot lo lleva).

Ejercicios de la Sección 1.3 — Los supuestos clásicos

Ejercicio 1.3.1. Cada uno de estos hechos sobre un hospital real rompe uno de los cinco supuestos clásicos. Di cuál, y qué parte del libro se ocupa de él según la Tabla 1.1: (a) cruzar el pasillo lleva 40 segundos; (b) la farmacia cierra a las 15:00; (c) una vez de cada cincuenta el robot no consigue agarrar una caja; (d) el nivel de batería es un número entre 0 y 100; (e) nadie sabe en qué estantería se dejó la muestra; (f) dos robots trabajan a la vez.

Ejercicio 1.3.2. La caja de concepto de la sección afirma que un modelo equivocado puede ser útil. Da un ejemplo de una decisión para la que el modelo de ocho lugares del mundo del mensajero es lo bastante bueno aunque ignore las distancias, y una decisión para la que ignorar las distancias da una respuesta claramente mala.

Ejercicios de la Sección 1.4 — La planificación y sus alternativas

Ejercicio 1.4.1. Escribe, en pseudocódigo, un programa hecho a mano para el robot mensajero que atienda peticiones de la forma «lleva el artículo *i* del lugar *a* al lugar *b*» sobre el mapa de la Figura 1.1. Puedes suponer que existe una función `route(a, b)` que devuelve la lista de lugares entre dos lugares. Después enumera lo que tendrías que cambiar en tu programa si (a) una petición puede nombrar dos artículos para el mismo destino y el robot tiene dos bandejas; (b) algunas conexiones están cerradas algunos días. Compáralo con lo que cambiaría en un modelo.

Ejercicio 1.4.2. Para cada sistema, elige el enfoque de la Tabla 1.2 que usarías primero y justifícalo en una frase: (a) una máquina expendedora; (b) un brazo robótico que debe reconocer tomates maduros; (c) un almacén donde cuarenta robots atienden pedidos que llegan durante todo el día; (d) el resolutor de un rompecabezas fijo de un videojuego.

Ejercicios de la Sección 1.6 — Dominio, problema y PDDL

Ejercicio 1.6.1. Di si cada afirmación pertenece al dominio o al problema del mundo del mensajero: (a) el vestíbulo está conectado con el pasillo; (b) un robot solo puede coger un artículo si los dos están en el mismo lugar; (c) hay dos robots; (d) `meal1` tiene que acabar en la sala B; (e) después de dejar algo, el robot no lleva nada; (f) el robot empieza en la base.

Ejercicio 1.6.2. Con la Tabla 1.3, indica la primera versión de PDDL que puede expresar cada requisito: (a) el robot tiene una batería que pierde 2 unidades por metro; (b) el laboratorio solo acepta muestras entre las 8:00 y las 14:00; (c) estaría bien, aunque no es obligatorio, que el robot acabe en la base; (d) moverse lleva tiempo y dos robots se mueven a la vez; (e) una muestra fuera de la nevera se calienta sola y se estropea a 8 grados; (f) el robot nunca debe entrar en la UCI mientras lleva residuos.

Ejercicios de la Sección 1.7 — Primer contacto con Kairos

Ejercicio 1.7.1. Delante del ordenador. Ejecuta Kairos sobre los ficheros `courier-domain.pddl` y `courier-p01.pddl` y comprueba que obtienes el plan del capítulo. Copia el problema en un fichero nuevo y, sin leer ninguna explicación de la sintaxis, cámbialo para que los medicamentos tengan que acabar en la UCI. Antes de ejecutarlo, predice con el mapa el número de pasos del plan.

Ejercicio 1.7.2. Delante del ordenador. En una copia de `courier-p01.pddl`, borra el texto `(free r1)` y ejecuta el planificador. Lee la respuesta de Kairos y explica, a partir de ella, qué le decía al planificador ese fragmento de texto.

Ejercicio 1.7.3. Delante del ordenador. Escribe a mano, en un fichero, un plan para `courier-p02.pddl` en el formato que imprime Kairos, y valídalo. Después intercambia dos líneas de tu fichero para que el robot coja la muestra antes de llegar a la UCI, vuelve a validar y explica cada parte del mensaje.

Ejercicios de la Sección 1.8 — El método MODEL: cinco pasos y cinco criterios

Ejercicio 1.8.1. Escribe una ficha del modelo (Historia, Agentes, Éxito, Alcance, Supuestos) para un robot aspirador que debe limpiar las habitaciones de un piso y volver a su base. Da al menos tres cosas que dejes fuera del alcance y, para una de ellas, describe una situación en la que dejarla fuera produce un mal plan.

Ejercicio 1.8.2. Asocia cada actividad con el paso del método al que pertenece: (a) decidir que el modelo no registrará la batería; (b) escribir un problema que no debería tener plan y comprobar que el planificador está de acuerdo; (c) enumerar lo que cambia al moverse; (d) decidir registrar qué artículo lleva un robot; (e) reescribir el modelo sin la acción de coger para ver si los planes se acortan; (f) leer un plan línea a línea.

Ejercicio 1.8.3. Un compañero propone una tercera codificación para la Tabla 1.4: la planta tiene solo dos lugares, «lado de la farmacia» (vestíbulo, farmacia, base) y «lado de las salas» (todo lo demás). Júzgala con los cinco criterios y da un veredicto: ¿hay alguna petición para la que sea lo bastante buena, y alguna para la que pierda fidelidad?

Un primer modelo: STRIPS 2.

En el Capítulo 1 entregamos dos ficheros a un planificador y recibimos un plan que llevaba una caja de medicamentos de la farmacia a la sala A. Podíamos ejecutar el modelo, pero no cambiarlo. Supongamos que ahora el hospital nos dice que el robot tiene que volver a su base de carga después de cada entrega, o que hoy la UCI está cerrada, o que llega un segundo tipo de robot capaz de llevar dos artículos. Cada una de estas cosas es un cambio pequeño en el modelo, y una caja negra no se puede cambiar. Para modelar hay que escribir.

Este capítulo enseña el PDDL necesario para escribir un modelo completo partiendo de un fichero vacío: cómo se organiza el lenguaje, cómo se dice qué puede ser cierto en un mundo (predicados), cómo se dice qué se puede hacer en él (acciones, con sus precondiciones y efectos) y cómo se describen una situación y un objetivo (objetos, estado inicial, objetivo). El nivel del lenguaje que usamos es el más antiguo y sencillo, llamado STRIPS: las condiciones son listas de hechos que deben ser ciertos, y los efectos son listas de hechos que pasan a ser ciertos o falsos. Todo lo que añaden los capítulos posteriores se construye sobre él.

Las cuatro primeras secciones enseñan la sintaxis con el dominio más pequeño que se nos ha ocurrido: unas cuantas lámparas con sus interruptores. Con la sintaxis aprendida, aplicamos por primera vez el método MODEL completo al mundo del mensajero: la ficha del modelo, la tabla de estado y la tabla de acciones nos llevan al fichero de dominio, que después ponemos a prueba con cuatro problemas pensados para cazar sus errores. Dos dominios clásicos, el mundo de bloques y gripper, muestran las mismas ideas lejos del hospital. La sección del método recoge lo que enseña esta primera pasada y compara tres codificaciones del mundo del mensajero, y el capítulo termina, como terminarán todos, con los errores que comete todo el mundo.

2.1 Listas dentro de listas: la forma de PDDL 21
2.2 Predicados, objetos y hechos 22
Lo que se puede decir: predicados 22
Lo que es cierto: hechos y estados 23
2.3 Acciones: precondiciones y efectos 23
Aplicar una acción 24
2.4 El problema: objetos, estado inicial y objetivo . 25
2.5 Modelar el mundo del mensajero 26
Paso M: mapear la situación 26
Paso O: objetos y estado . . 27
Paso D: dinámica 28
Un problema para el dominio 29
2.6 Ensayar el modelo 30
Comprobar, resolver, validar 30
La batería de pruebas . . . 31
2.7 Dos clásicos: el mundo de bloques y gripper 35
El mundo de bloques . . . 35
Gripper 37
2.8 El método MODEL: la primera pasada completa 38
Lo que añade este capítulo 39
Alternativas comparadas . 39
2.9 Errores comunes 41
2.10 Resumen 44
2.11 Lecturas complementarias 45
2.12 Ejercicios 46

2.1. Listas dentro de listas: la forma de PDDL

Un fichero PDDL parece extraño a primera vista por sus paréntesis. La extrañeza desaparece con una sola regla: en PDDL todo es una lista entre paréntesis, y el primer elemento de la lista dice qué es la lista. Los elementos se separan con espacios, y un elemento puede ser una palabra u otra lista.

```
(on desk)
(and (on desk) (off hall))
(:goal (and (on desk) (off hall)))
```

La primera línea es una lista de dos palabras. La leeremos como «`on` se cumple para `desk`»: la lámpara del escritorio está encendida. La segunda es una lista de tres elementos, la palabra `and` y dos listas: se cumplen las dos cosas. La tercera envuelve la segunda en una lista que empieza por `:goal`: esto es lo que se quiere conseguir. Las listas escritas así se llaman **expresiones S**, una notación que PDDL hereda del lenguaje de programación Lisp. La Figura 2.1 dibuja la tercera como un árbol, que es como la ve un planificador.

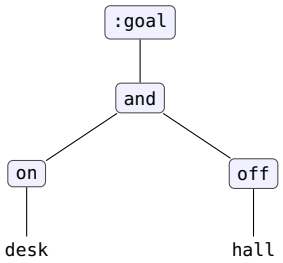


Figura 2.1.: La expresión `(:goal (and (on desk) (off hall)))` como árbol. Cada par de paréntesis es un nodo, que toma el nombre de la primera palabra de su interior; lo que sigue a esa primera palabra cuelga debajo.

Cuatro reglas más completan la forma.

- **La disposición es libre.** Espacios, tabuladores y saltos de línea cuentan todos como un único separador. Sangramos para mostrar qué lista está dentro de cuál; el planificador no mira la sangría, cuenta paréntesis.
- Los **comentarios** van desde un punto y coma hasta el final de la línea.
- Los **nombres** se forman con letras, dígitos, guiones y guiones bajos, y empiezan por una letra: `ward-a`, `meds1`, `robot-at`. PDDL no distingue mayúsculas de minúsculas; este libro lo escribe todo en minúsculas, con guiones entre las palabras.
- Las **palabras clave** del lenguaje que introducen una sección empiezan por dos puntos (`:goal`, `:action`), y las **variables** empiezan por un signo de interrogación (`?l`).

PDDL se escribe como listas entre paréntesis. La primera palabra de una lista dice qué es la lista; el resto son sus partes, que a su vez pueden ser listas.

Un modelo son dos ficheros, como vimos: un dominio y un problema. Cada uno es una única lista grande que empieza por la palabra `define`:

```
(define (domain switch)
  ...)                        ; one file

(define (problem switch-p01)
  (:domain switch)
  ...)                        ; another file
```

El dominio tiene un nombre, aquí `switch`. El problema también tiene nombre, y dice con `(:domain switch)` a qué dominio pertenece. Los puntos ocupan el lugar de las secciones que irán rellenando las tres secciones siguientes del capítulo: predicados y acciones en el dominio; objetos, estado inicial y objetivo en el problema.

2.2. Predicados, objetos y hechos

Necesitamos un primer mundo, lo más pequeño posible. En una casa hay unas cuantas lámparas: una en el recibidor, otra en el escritorio y otra en el porche. Cada lámpara está encendida o apagada, y tiene un interruptor. Eso es todo.

Lo que se puede decir: predicados

Las cosas que existen en un problema son sus **objetos**: aquí las tres lámparas, `hall`, `desk` y `porch`. Un **predicado** es una propiedad que puede tener un objeto, o una relación en la que pueden estar varios objetos. El dominio declara sus predicados en la sección `:predicates`, cada uno con un nombre y una lista de variables, una por cada objeto del que habla el predicado:

```
(:predicates
  (on ?l)       ; lamp ?l is lit
  (off ?l))     ; lamp ?l is dark
```

Esto declara dos predicados de un parámetro. La variable `?l` representa «cualquier objeto»: la declaración dice solo que `on` se dice de una cosa. Su nombre es libre, y elegir uno bueno (`?l` para una lámpara) es, junto con el comentario, la documentación del predicado. Un predicado puede tener varios parámetros, como `(connected ?from ?to)` en el mundo del mensajero, o ninguno.

Lo que es cierto: hechos y estados

Un predicado aplicado a objetos es un **hecho**[1]: `(on desk)` es un hecho, y también lo es `(off hall)`. Un hecho es cierto o falso en cada momento. Con tres lámparas y dos predicados se pueden escribir seis hechos: `(on hall)`, `(on desk)`, `(on porch)`, `(off hall)`, `(off desk)`, `(off porch)`.

1: Otros textos dicen *átomo* o *átomo instanciado* (en los mensajes de Kairos, *atom* o *ground atom*) para lo que aquí llamamos hecho. *Instanciado* (*ground*) significa «con objetos, sin variables».

En el Capítulo 1 un estado era «una descripción del mundo en un momento dado». Ahora podemos ser exactos: un **estado** es un conjunto de hechos, los que son ciertos en ese momento. El estado en el que solo está encendida la lámpara del recibidor es el conjunto

$$\{\texttt{(on hall)}, \texttt{(off desk)}, \texttt{(off porch)}\}.$$

Todo lo que no está en el conjunto es falso. Esta convención tiene nombre, la **hipótesis del mundo cerrado**: lo que no se afirma que es cierto es falso. No hay un tercer valor. Un planificador nunca piensa «no sé si la lámpara del porche está encendida»: si `(on porch)` no está en el estado, la lámpara del porche no está encendida.

Un hecho es un predicado aplicado a objetos. Un estado es el conjunto de los hechos que son ciertos; cualquier otro hecho es falso (hipótesis del mundo cerrado).

Hay dos cosas de esta definición que merecen una segunda mirada.

La primera: el planificador no da ningún significado a los nombres. Para nosotros, `on` y `off` son opuestos. Para el planificador son dos predicados sin relación entre sí, y el conjunto {`(on hall)`, `(off hall)`} es un estado perfectamente válido en el que la lámpara del recibidor está encendida y apagada a la vez. Nada en el lenguaje lo prohíbe. Es el modelo, mediante su estado inicial y sus acciones, el que nunca debe producir un estado así. Mantener los hechos de un modelo coherentes entre sí es buena parte del oficio, y nos lo encontraremos en todos los capítulos.

La segunda: un lector razonable preguntará para qué necesitamos `off`, si una lámpara que no está encendida está apagada. La respuesta es un límite de STRIPS, el nivel de PDDL de este capítulo: una condición puede pedir que ciertos hechos sean ciertos, pero no puede pedir que un hecho sea falso. Para encender una lámpara queremos la condición «la lámpara está apagada», y la única forma de pedirla es un hecho que sea cierto exactamente cuando la lámpara está apagada. El Capítulo 5 levanta este límite con las precondiciones negativas y compara las dos formas.

2.3. Acciones: precondiciones y efectos

Las lámparas tienen interruptores. En PDDL, lo que se puede hacer es una **acción**, que se declara en el dominio con cuatro partes: un nombre, sus `:parameters`, su `:precondition` y su `:effect`. El Listado 2.1 es el dominio completo.

```
1  ;; The smallest domain of the book: lamps that are on or off.
2  (define (domain switch)
3    (:requirements :strips)
4
5    (:predicates
6      (on ?l)       ; lamp ?l is lit
7      (off ?l))     ; lamp ?l is dark
8
9    (:action switch-on
10     :parameters (?l)
11     :precondition (off ?l)
12     :effect (and (on ?l)
13                  (not (off ?l))))
14
15   (:action switch-off
16     :parameters (?l)
17     :precondition (on ?l)
18     :effect (and (off ?l)
19                  (not (on ?l)))))
```

Listado 2.1: El dominio de los interruptores, completo: dos predicados y dos acciones.

La línea 3 es nueva. La sección `:requirements` enumera las características del lenguaje que usa el fichero, para que un planificador sepa enseguida si es capaz de leerlo. `:strips` es el nivel básico: todos los planificadores lo admiten. Cada capítulo posterior añadirá requisitos a esta lista, y siempre será lo primero que escribamos.

Tomemos la acción `switch-on`, en las líneas 9 a 13.

`:parameters` es la lista de variables de la acción: aquello sobre lo que actúa. `switch-on` actúa sobre una cosa, `?l`. La acción es un patrón: describe a la vez cómo se enciende la lámpara del recibidor, la del escritorio y cualquier lámpara de cualquier problema.

`:precondition` es lo que debe ser cierto para que la acción sea posible: aquí, que `?l` esté apagada. Si hay más de un hecho, se unen con `and`. En STRIPS una precondición es siempre esto: un hecho, o una lista de hechos que deben ser todos ciertos.

`:effect` es lo que cambia la acción. Un hecho escrito tal cual pasa a ser cierto: es un **efecto de adición**. Un hecho envuelto en `not` pasa a ser falso: es un **efecto de borrado**. Después de `switch-on`, `(on ?l)` es cierto y `(off ?l)` es falso.

Las variables de la precondición y del efecto tienen que ser parámetros de la acción. Al sustituir cada parámetro por un objeto se obtiene una **acción instanciada**, que es de lo que está hecho un plan: `(switch-on desk)` es `switch-on` con `desk` en el lugar de `?l`. Con tres lámparas, las dos acciones del dominio dan seis acciones instanciadas.

Aplicar una acción

Lo que una acción le hace a un estado se define con dos reglas, y conviene sabérselas de memoria.

1. Una acción instanciada es **aplicable** en un estado si todos los hechos de su precondición están en el estado.
2. **Aplicarla** da un estado nuevo: se toma el anterior, se quitan los hechos de los efectos de borrado y se ponen los hechos de los efectos de adición. Todos los demás hechos se quedan como estaban.

La Figura 2.2 aplica `(switch-on desk)` al estado en el que solo está encendida la lámpara del recibidor. La precondición `(off desk)` está en el estado, así que la acción es aplicable. El efecto de borrado elimina

`(off desk)`, el efecto de adición pone `(on desk)`, y los dos hechos sobre las otras lámparas pasan sin cambios.

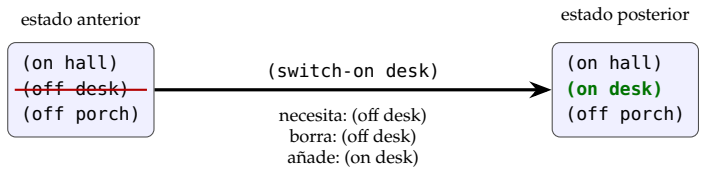


Figura 2.2.: Aplicar una acción. La precondición se comprueba en el estado anterior; el estado posterior es el mismo conjunto sin el hecho borrado (tachado) y con el añadido (en verde). Los hechos que la acción no menciona se copian.

La última frase de la regla 2 es la razón de que las acciones de STRIPS sean cortas. Una acción dice solo lo que cambia. Para encender la lámpara del escritorio no hace falta decir que la del recibidor sigue encendida, que la del porche sigue apagada o que el robot que está tres habitaciones más allá sigue donde estaba. A cambio, una acción tiene que decir *todo* lo que cambia: si `switch-on` olvidara su efecto de borrado, la lámpara del escritorio acabaría encendida y apagada a la vez. A esta doble obligación la llamaremos regla del marco, y volveremos a ella en la sección del método.

Una acción es aplicable cuando los hechos de su precondición están en el estado. Aplicarla quita los efectos de borrado y añade los efectos de adición. Nada más cambia.

STRIPS, el planificador y el nivel de PDDL

STRIPS fue el planificador del robot Shakey Fikes y Nilsson, 1971, y su idea duradera fue la forma de describir una acción que acabamos de ver: una precondición, una lista de hechos que añadir y una lista de hechos que borrar. En PDDL el nombre sobrevive en el requisito `:strips`, que significa: las precondiciones y los objetivos son conjunciones de hechos positivos, y los efectos son conjunciones de efectos de adición y de borrado. Es el subconjunto útil más pequeño del lenguaje, y basta para escribir todos los modelos de este capítulo.

2.4. El problema: objetos, estado inicial y objetivo

El dominio dice cómo son las lámparas. Un problema dice qué lámparas hay, cómo están ahora y cómo las queremos. El Listado 2.2 es un problema del dominio de los interruptores.

```
;; Three lamps: leave the desk lamp lit and the hall lamp dark.
(define (problem switch-p01)
  (:domain switch)
  (:objects hall desk porch)
  (:init (on hall) (off desk) (off porch))
  (:goal (and (on desk) (off hall))))
```

Listado 2.2: Un problema del dominio de los interruptores: tres lámparas, una de ellas encendida.

Tras su nombre y su dominio, un problema tiene tres secciones.

`:objects` enumera los nombres de los objetos. Existen a partir de esta línea; no existe nada más.

`:init` es el estado inicial: la lista de los hechos que son ciertos al principio. Por la hipótesis del mundo cerrado, todo hecho que no está en la lista es falso. El estado inicial tiene que ser completo. Si olvidamos `(off porch)`, la lámpara del porche no está ni encendida ni apagada, y ninguna acción del dominio la tocará nunca.

`:goal` es una condición que debe cumplir el último estado: en STRIPS, un hecho o un `and` de hechos que deben ser ciertos. A diferencia del estado inicial, el objetivo es parcial. No dice nada de la lámpara del porche, así que la lámpara del porche puede acabar de cualquier manera.

El estado inicial es completo: lo que no enumera es falso. El objetivo es parcial: lo que no menciona queda libre.

Antes de planificar, le preguntamos a Kairos si los dos ficheros están bien escritos. La orden `check` lee un dominio, o un dominio y un problema, e informa de todos los errores que encuentra, sin buscar un plan:

```
$ kairos check switch-domain.pddl switch-p01.pddl
valid, plannable
```

Valid significa que los ficheros son PDDL correcto y concuerdan entre sí: todo predicado usado está declarado, con el número correcto de argumentos, y todo objeto mencionado existe. *Plannable* significa que Kairos admite todas las características que usan los ficheros. Ahora, el planificador:

```
$ kairos switch-domain.pddl switch-p01.pddl
; cost = 2
(switch-off hall)
(switch-on desk)
```

Un plan es una secuencia de acciones instanciadas, y ya podemos decir exactamente cuándo es correcto. Partimos del estado inicial. `(switch-off hall)` necesita `(on hall)`, que está; al aplicarla se obtiene {`(off hall)`, `(off desk)`, `(off porch)`}. En ese estado, `(switch-on desk)` necesita `(off desk)`, que está; al aplicarla se obtiene {`(off hall)`, `(on desk)`, `(off porch)`}. Los dos hechos del objetivo están en este último estado. Un **plan** es una secuencia de acciones instanciadas en la que cada una es aplicable en el estado que dejan las anteriores, y cuyo estado final contiene el objetivo. Aquí los dos pasos podrían ir en el otro orden, y eso sería otro plan, tan bueno como este.

2.5. Modelar el mundo del mensajero

Ya tenemos toda la sintaxis que necesita un modelo STRIPS. Pero escribir un buen modelo exige más que sintaxis, y aquí es donde el método MODEL se pone a trabajar. Vamos a seguir sus cinco pasos sobre el mundo del mensajero, despacio, porque es la primera vez. Los tres primeros pasos construyen el dominio; la sección siguiente lo pone a prueba.

Paso M: mapear la situación

La ficha del modelo del mundo del mensajero se escribió al final del Capítulo 1. La repetimos aquí, con la precisión que ahora nos podemos permitir.

Paso M: la ficha del modelo del mundo del mensajero

Historia	Un robot lleva artículos entre las habitaciones de una planta de hospital, según se le pide.
Agentes	Los robots mensajeros (uno, en este capítulo). Solo ellos actúan.
Éxito	Cada artículo pedido está en la habitación donde se pidió. Algunas peticiones dicen también dónde debe acabar el robot.
Alcance	Dentro: las habitaciones y cuáles están conectadas; dónde está cada robot y cada artículo; qué artículo lleva un robot. Fuera: distancias, tiempo, batería, puertas, personas, cómo agarra el brazo, en qué punto exacto de una habitación están las cosas.
Supuestos	Los clásicos. Un robot lleva como mucho un artículo. Los artículos no se mueven solos. Toda conexión se puede recorrer en los dos sentidos, salvo que el problema diga otra cosa.

La ficha decide de qué pueden hablar los dos pasos siguientes. «Un robot lleva como mucho un artículo» se convertirá en un predicado y dos condiciones. «Fuera: distancias» significa que ningún predicado hablará de metros, y que el planificador contará todos los movimientos como iguales.

Paso O: objetos y estado

El paso O pregunta qué cosas existen y qué necesita saber de ellas el planificador para decidir. Las cosas de la ficha son de tres clases: robots, lugares y artículos. Para cada clase nos preguntamos qué se puede decir de ella que importe para una entrega, y para cada respuesta, si puede cambiar.

Las propiedades estáticas nunca cambian: son datos del problema. Las dinámicas las cambia alguna acción, y hay que revisar cada acción frente a cada una de ellas.

Una propiedad que ninguna acción cambia nunca es **estática**: el mapa es estático, porque moverse, coger y dejar no construyen pasillos. Una propiedad que alguna acción cambia es **dinámica**: dónde está el robot. La diferencia nos importa a nosotros, porque las propiedades estáticas son datos que escribimos una vez en el estado inicial, y le importa al planificador, que puede tratar los hechos estáticos de una forma mucho más barata, como mostrará el Capítulo 3. Las respuestas van a la *tabla de estado* (Tabla 2.1).

Tabla 2.1.: Paso O: la tabla de estado del mundo del mensajero. Cada fila se convierte en un predicado del dominio.

Propiedad	Sobre	Clase	Codificada como
Es un robot / un lugar / un artículo	un objeto	estática	`(robot ?x)` etc.
Se puede ir de un lugar a otro	dos lugares	estática	`(connected ?from ?to)`
Dónde está un robot	un robot, un lugar	dinámica	`(robot-at ?r ?l)`
Dónde está un artículo	un artículo, un lugar	dinámica	`(item-at ?i ?l)`
Qué artículo lleva un robot	un robot, un artículo	dinámica	`(carrying ?r ?i)`
El robot no lleva nada	un robot	dinámica	`(free ?r)`

Tres filas merecen un comentario.

La primera fila existe porque STRIPS, tal como lo usamos en este capítulo, no tiene noción de clases de objetos. Para el planificador, `r1`, `lobby` y

`meds1` son tres nombres, y nada le impide intentar mover el vestíbulo o coger el robot. Distinguimos las clases con tres predicados de un parámetro, `(robot ?x)`, `(location ?x)` y `(item ?x)`, llamados **predicados de tipo**. El problema indica qué objeto es de qué clase, y cada acción pide las clases que necesita. El Capítulo 4 los sustituye por tipos de verdad.

La segunda fila, `connected`, tiene dirección: `(connected lobby corridor)` dice que se puede ir del vestíbulo al pasillo, y nada sobre la vuelta. El planificador no sabe que los pasillos se recorren en los dos sentidos. Una conexión de doble sentido son dos hechos, y el problema tiene que enumerar los dos.

La última fila, `free`, es redundante para nosotros: un robot está libre exactamente cuando no lleva ningún artículo. La necesitamos por la misma razón que `off` en las lámparas. La acción de coger tiene que pedir que el robot no lleve nada, y STRIPS solo puede pedir hechos que sean ciertos, así que «no lleva nada» tiene que ser un hecho. El precio es que ahora dos predicados cuentan historias que se solapan, y las acciones deben mantenerlos acompasados: siempre que aparece `carrying`, `free` tiene que desaparecer, y al revés.

Paso D: dinámica

El paso D pregunta qué se puede hacer, cuándo y qué cambia. La ficha da tres verbos: un robot se mueve, coge y deja. Para cada uno rellenamos una fila de la *tabla de acciones* (Tabla 2.2), primero con palabras.

Acción	Parámetros	Se aplica cuando	Cambia
`move`	robot, origen, destino	el robot está en *origen*; *origen* está conectado con *destino*	el robot está en *destino*; ya no está en *origen*
`pick`	robot, artículo, lugar	el robot y el artículo están los dos en el lugar; el robot está libre	el robot lleva el artículo; el artículo ya no está en el lugar; el robot ya no está libre
`drop`	robot, artículo, lugar	el robot está en el lugar y lleva el artículo	el artículo está en el lugar; el robot está libre; ya no lleva el artículo

Tabla 2.2.: Paso D: la tabla de acciones del mundo del mensajero. La columna «Cambia» enumera, para cada acción, lo que pasa a ser cierto y lo que deja de serlo.

La columna de cambios es donde se equivocan los modelos, así que la rellenamos con una rutina fija: para cada fila dinámica de la tabla de estado, nos preguntamos si la acción la hace cierta o falsa para algún objeto. Tomemos `pick`. Dónde está el robot: no cambia. Dónde está el artículo: deja de estar en el lugar, porque ahora va encima del robot. Qué lleva el robot: el artículo. Libre: ya no. Cambian tres de las cuatro propiedades dinámicas, y la tabla muestra las tres.

Los parámetros salen de la misma rutina: una acción necesita como parámetros todos los objetos que mencionan su condición y sus cambios. Por eso `pick` tiene el lugar como parámetro, aunque «el robot coge el artículo» no lo nombre: la condición habla del sitio donde están los dos, y el efecto de borrado `(not (item-at ?i ?l))` tiene que decir qué hecho quitar.

Escribir la tabla en PDDL es ya casi mecánico. El Listado 2.3 es el resultado: el fichero del Capítulo 1, que por fin podemos leer.

```
;; The courier world, first version: plain STRIPS, no types.
;; A robot carries items between the rooms of a hospital floor.
(define (domain courier)
  (:requirements :strips)

```

```
  (:predicates
    (robot ?x)                ; ?x is a robot
    (location ?x)             ; ?x is a room of the floor
    (item ?x)                 ; ?x is something to carry
    (connected ?from ?to)     ; the robot can go from ?from to ?to
    (robot-at ?r ?l)          ; robot ?r is in ?l
    (item-at ?i ?l)           ; item ?i lies in ?l
    (carrying ?r ?i)          ; robot ?r carries item ?i
    (free ?r))                ; robot ?r carries nothing

  (:action move
    :parameters (?r ?from ?to)
    :precondition (and (robot ?r) (location ?from) (location ?to)
                       (robot-at ?r ?from) (connected ?from ?to))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to)))

  (:action pick
    :parameters (?r ?i ?l)
    :precondition (and (robot ?r) (item ?i) (location ?l)
                       (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
    :effect (and (not (item-at ?i ?l)) (not (free ?r))
                 (carrying ?r ?i)))

  (:action drop
    :parameters (?r ?i ?l)
    :precondition (and (robot ?r) (item ?i) (location ?l)
                       (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l) (free ?r))))
```

Listado 2.3: El dominio del mensajero en STRIPS: la tabla de estado se ha convertido en los predicados y la tabla de acciones, en las acciones.

Cada acción empieza su precondición con los predicados de tipo de sus parámetros, y sigue con las condiciones de la tabla de acciones. Cada efecto tiene los borrados y las adiciones de la columna «Cambia». Compara `pick` y `drop`: una es la imagen especular de la otra, y lo que una añade la otra lo borra. Esa simetría es buena señal en un modelo, y su ausencia merece una segunda mirada.

Un problema para el dominio

El Listado 2.4 es el problema del Capítulo 1: los medicamentos van de la farmacia a la sala A.

```
;; Take the medicines from the pharmacy to ward A.
(define (problem courier-p01)
  (:domain courier)
  (:objects r1 meds1
            lobby corridor pharmacy lab ward-a ward-b icu dock)
  (:init
    (robot r1) (item meds1)
    (location lobby) (location corridor) (location pharmacy)
    (location lab) (location ward-a) (location ward-b)
    (location icu) (location dock)
    ; the map: every connection in both directions
    (connected lobby corridor) (connected corridor lobby)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (connected lobby dock) (connected dock lobby)
    (connected corridor lab) (connected lab corridor)
    (connected corridor ward-a) (connected ward-a corridor)
    (connected corridor ward-b) (connected ward-b corridor)
    (connected ward-b icu) (connected icu ward-b)
    ; where things are
    (robot-at r1 dock) (free r1)
    (item-at meds1 pharmacy))
  (:goal (item-at meds1 ward-a)))
```

Listado 2.4: El primer problema del mensajero. La mayor parte del estado inicial son datos estáticos: las clases de los objetos y el mapa.

El problema tiene diez objetos, y su estado inicial, 27 hechos. De ellos, 24 son estáticos: diez dicen la clase de cada objeto y catorce dibujan el mapa, cada una de las siete conexiones en sus dos sentidos. Solo los tres últimos, en las líneas 20 y 21, describen cosas que van a cambiar. Esta proporción es típica. Buena parte de un problema es la descripción de un mundo fijo, y la parte dinámica, la que cambia un plan, suele ser pequeña (Figura 2.3).

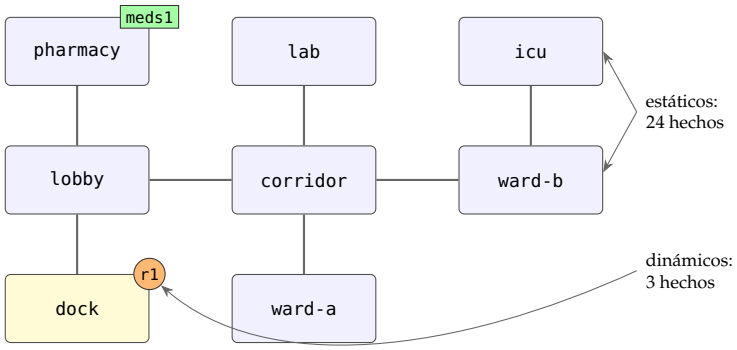


Figura 2.3.: Lo estático y lo dinámico en `courier-p01.pddl`. Las habitaciones y las líneas que las unen nunca cambian: son los 24 hechos estáticos. El robot y el artículo, dibujados encima, son lo que describen los tres hechos dinámicos y lo que mueve un plan.

El objetivo es un único hecho, `(item-at meds1 ward-a)`. No dice nada del robot, así que se acepta cualquier lugar final para el robot.

2.6. Ensayar el modelo

Un modelo es un programa, y un programa que solo se ha escrito probablemente está mal. El paso E ejecuta el modelo con una pregunta en mente: ¿hace el planificador con él lo que el mundo permitiría, y nada más? La rutina tiene tres órdenes y cuatro pruebas.

Comprobar, resolver, validar

Primero, `check`. Encuentra los errores de forma antes de que se conviertan en misterios:

```
$ kairos check courier-domain.pddl courier-p01.pddl
valid, plannable
```

Segundo, resolver. El plan es el que ya conocemos:

```
$ kairos courier-domain.pddl courier-p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Tercero, leer el plan. Leer significa seguirlo sobre el estado, como hicimos con las lámparas, y preguntarse en cada paso si el robot real podría darlo.

La Tabla 2.3 sigue los hechos dinámicos; los estáticos no cambian y no hace falta escribirlos.

Después de	Hechos dinámicos del estado
(inicio)	`(robot-at r1 dock) (free r1) (item-at meds1 pharmacy)`
`(move r1 dock lobby)`	`(robot-at r1 lobby) (free r1) (item-at meds1 pharmacy)`
`(move r1 lobby pharmacy)`	`(robot-at r1 pharmacy) (free r1) (item-at meds1 pharmacy)`
`(pick r1 meds1 pharmacy)`	`(robot-at r1 pharmacy) (carrying r1 meds1)`
`(move r1 pharmacy lobby)`	`(robot-at r1 lobby) (carrying r1 meds1)`
`(move r1 lobby corridor)`	`(robot-at r1 corridor) (carrying r1 meds1)`
`(move r1 corridor ward-a)`	`(robot-at r1 ward-a) (carrying r1 meds1)`
`(drop r1 meds1 ward-a)`	`(robot-at r1 ward-a) (free r1) (item-at meds1 ward-a)`

Tabla 2.3.: El plan de `courier-p01.pddl`, leído sobre el estado. Mientras el robot lleva los medicamentos, estos no están en ninguna habitación: `pick` borró su hecho `item-at` y `drop` añade uno nuevo.

La traza muestra una decisión del modelo que el listado ocultaba: un artículo que va encima del robot no está en ninguna parte. Ningún hecho `item-at` lo menciona entre la acción de cogerlo y la de dejarlo. Es una elección (otro modelo podría mantener el artículo «dentro» de la habitación que cruza el robot), y una buena, porque así nada puede coger el artículo del suelo mientras el robot lo sostiene.

La orden `validate` hace esa misma lectura por nosotros, sobre cualquier plan guardado en un fichero: uno que haya escrito un planificador o uno que hayamos tecleado nosotros. El fichero `courier-p01.plan` contiene los siete pasos de arriba, y `courier-p01-shortcut.plan` es un plan con un error: va de la farmacia directamente al pasillo.

```
$ kairos validate courier-domain.pddl courier-p01.pddl \
    courier-p01.plan
plan valid, makespan 7
$ kairos validate courier-domain.pddl courier-p01.pddl \
    courier-p01-shortcut.plan
courier-p01-shortcut.plan:6: error: step 4 (move r1 pharmacy
     corridor) at 3.000: precondition false: (connected pharmacy
     corridor) is false [validate.precondition_failed]
 6 | (move r1 pharmacy corridor)
plan invalid
```

El mensaje nombra el paso, el hecho de su precondición que es falso y la línea del fichero del plan. (El número que sigue a *at* es el instante del paso: Kairos asigna a los pasos de un plan secuencial los instantes 0, 1, 2, etc., y llama al total *makespan*. El tiempo empieza a ser interesante en el Capítulo 10.) Validar es la forma de hacerle al modelo una pregunta precisa: «¿permitirías esto?». Enseguida lo usaremos para preguntar por un plan que *no* debería permitirse.

La rutina del paso E: comprobar los ficheros, resolver, leer el plan sobre el estado y validar. Un plan válido para el modelo puede seguir siendo incorrecto para el mundo: eso es lo que buscan las pruebas.

La batería de pruebas

Un problema que da un plan sensato demuestra poco. El método MODEL pide cuatro clases de pruebas, cada una dirigida a una clase distinta de error.

Prueba 1: un problema trivial. Un problema tan pequeño que conocemos su plan antes de ejecutarlo. En `courier-p02.pddl` el robot empieza en la farmacia, junto a los medicamentos, y tiene que llevarlos a la habitación de al lado, el vestíbulo.

```
    ; where things are
    (robot-at r1 pharmacy) (free r1)
    (item-at meds1 pharmacy))
  (:goal (item-at meds1 lobby)))
```

```
$ kairos courier-domain.pddl courier-p02.pddl
; cost = 3
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(drop r1 meds1 lobby)
```

Tres pasos, uno de cada acción, como esperábamos. Una prueba trivial caza los errores gruesos: una acción que nunca se aplica, un efecto sobre el objeto equivocado. Cuando falla, la causa es fácil de encontrar, porque intervienen muy pocas cosas.

Prueba 2: un problema típico. Un problema como aquellos para los que está pensado el modelo. `courier-p01.pddl` es uno. Uno más rico es `courier-p03.pddl`, con dos entregas y un robot que tiene que volver a su base de carga:

```
    ; where things are
    (robot-at r1 dock) (free r1)
    (item-at meds1 pharmacy) (item-at sample1 icu))
  (:goal (and (item-at meds1 ward-a)
              (item-at sample1 lab)
              (robot-at r1 dock))))
```

```
$ kairos courier-domain.pddl courier-p03.pddl
; cost = 18
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(pick r1 sample1 icu)
(move r1 icu ward-b)
(move r1 ward-b corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
(move r1 lab corridor)
(move r1 corridor lobby)
(move r1 lobby dock)
```

El plan entrega los medicamentos (pasos 1 a 7), recoge la muestra y la lleva al laboratorio (pasos 8 a 15) y vuelve a la base de carga (pasos 16 a 18). El objetivo tiene tres hechos unidos por `and`, y el tercero, `(robot-at r1 dock)`, es el que hace volver al robot: en el primer problema nadie lo pedía, y el robot se quedó en la sala A. Al leer un plan típico buscamos pasos que una persona no daría. Aquí no hay ninguno.

Prueba 3: un problema que no debe tener plan. Un modelo puede estar mal por permitir demasiado, y una prueba que espera un plan nunca lo nota. Así que escribimos un problema que es imposible en el mundo, y exigimos al planificador que lo diga. En `courier-p04.pddl` la UCI está cerrada: el estado inicial no tiene los dos hechos `connected` que la unen con la sala B, y el objetivo quiere allí los medicamentos.

```
    (connected corridor ward-b) (connected ward-b corridor)
    ; no (connected ward-b icu) and no (connected icu ward-b)
    ; where things are
    (robot-at r1 dock) (free r1)
    (item-at meds1 pharmacy))
  (:goal (item-at meds1 icu)))
```

```
$ kairos courier-domain.pddl courier-p04.pddl
note: the goal needs atoms that no action can reach
(item-at meds1 icu) can never be reached:
  drop adds it, but needs (robot-at ?r - object icu), which never
     holds
    move adds it, but needs (connected ?from - object icu), which
     never holds
      no action adds it, and the initial state does not have it [
     orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

El veredicto es *unsolvable*, irresoluble: no existe ningún plan. La nota lo explica retrocediendo desde el objetivo, y con lo que ya sabemos podemos leerla palabra por palabra. Un *atom* es un hecho. `(item-at meds1 icu)` solo lo añade `drop`, cuya precondición incluye `(robot-at ?r icu)` para algún robot `?r`. (El texto `- object` tras una variable es su tipo, una notación del Capítulo 4; aquí significa «cualquier objeto».) Un hecho `(robot-at ?r icu)` solo lo añade `move`, que necesita `(connected ?from icu)` para algún lugar, y ninguna acción añade un hecho `connected`: es estático, y el estado inicial no tiene ninguno que acabe en la UCI. Es la explicación que habríamos dado nosotros.

Prueba 4: un problema que tienta al planificador a hacer trampa. La prueba más útil, y la que es propia del modelado. Un planificador busca el camino más corto hacia el objetivo que el modelo le permite. Si el modelo tiene un agujero, el planificador lo encontrará y se colará por él. Así que escribimos un problema en el que saltarse una regla del mundo saldría a cuenta, y comprobamos que el plan no se la salta. La regla que hay que probar en el mundo del mensajero es «un artículo cada vez». En `courier-p05.pddl`, dos cajas de medicamentos esperan en la farmacia y las dos van a la sala A:

```
    ; where things are
    (robot-at r1 dock) (free r1)
    (item-at meds1 pharmacy) (item-at meds2 pharmacy))
  (:goal (and (item-at meds1 ward-a) (item-at meds2 ward-a))))
```

Un robot tramposo hace un solo viaje con las dos cajas: 2 movimientos hasta la farmacia, 2 recogidas, 3 movimientos hasta la sala y 2 descargas, 9 pasos. Un robot honrado hace dos viajes: 15 pasos (Figura 2.4). La longitud del plan nos dice cuál de los dos permite el modelo.

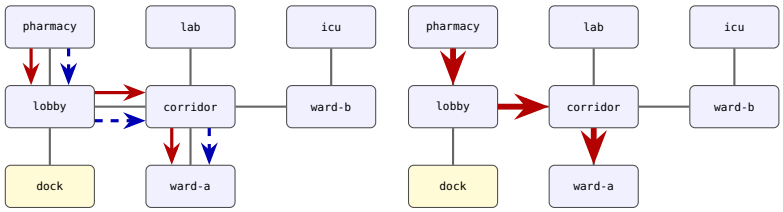
```
$ kairos courier-domain.pddl courier-p05.pddl
; cost = 15
```

```
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds2 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds2 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Quince pasos y dos viajes: el modelo aguanta. También se lo podemos preguntar directamente. El fichero `courier-p05-cheat.plan` es el plan tramposo de 9 pasos, tecleado a mano:

```
$ kairos validate courier-domain.pddl courier-p05.pddl \
    courier-p05-cheat.plan
courier-p05-cheat.plan:5: error: step 4 (pick r1 meds2 pharmacy) at
     3.000: precondition false: (free r1) is false [validate.
     precondition_failed]
 5 | (pick r1 meds2 pharmacy)
plan invalid
```

La segunda recogida se rechaza porque `(free r1)` es falso: la primera lo borró. El predicado que parecía redundante en la tabla de estado es lo que mantiene honrado al robot.



honrado: dos viajes, 15 pasos

tramposo: un viaje, 9 pasos

Figura 2.4.: La prueba de la trampa. Izquierda: los viajes con carga de un plan honrado, uno por caja (primero en trazo continuo y después en discontinuo). Derecha: el único viaje con carga de un plan que lleva las dos cajas. Un modelo que permite el plan de la derecha tiene un agujero, y el planificador lo encontrará porque el plan es más corto.

Una variante pide directamente el estado prohibido. En `courier-p06.pddl` el objetivo es `(and (carrying r1 meds1) (carrying r1 meds2))`, y la respuesta correcta es, otra vez, que no existe ningún plan:

```
$ kairos courier-domain.pddl courier-p06.pddl
note: no plan: every state reachable from the initial one was
     searched (1220 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

La razón es de otra clase que en la prueba 3. Allí, un hecho del objetivo nunca podía llegar a ser cierto. Aquí cada hecho del objetivo puede llegar a ser cierto, pero los dos nunca se cumplen a la vez; Kairos lo descubre visitando todos los estados a los que puede llegar el robot.

Paso E: la batería de pruebas del mundo del mensajero

Prueba	Fichero	Esperado y obtenido
Trivial	`courier-p02`	3 pasos: pick, move, drop
Típica	`courier-p01, p03`	7 y 18 pasos; todos los pasos, sensatos
Sin plan	`courier-p04`	irresoluble: no se puede llegar a la UCI
Trampa	`courier-p05, p06`	15 pasos (no 9); irresoluble

La batería de pruebas se queda con el modelo. Cada vez que cambia el dominio, en este capítulo y en los siguientes, las cuatro pruebas se vuelven a ejecutar: `tools/check_examples.sh` lo hace con todos los ejemplos del libro.

2.7. Dos clásicos: el mundo de bloques y gripper

Las ideas del capítulo no dependen de los hospitales. Dos dominios que conoce todo estudiante de planificación las muestran de nuevo, cada uno con una lección propia.

El mundo de bloques

Sobre una mesa hay unos bloques. Un bloque puede estar sobre la mesa o encima de otro bloque, y una mano los mueve de uno en uno: solo puede tomar un bloque si no tiene nada encima. Un problema da los bloques en una disposición y pide otra (Figura 2.5).

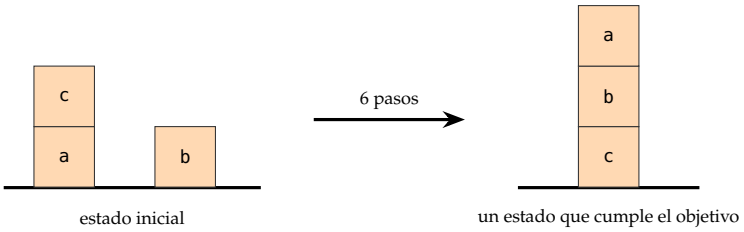


Figura 2.5.: El problema `blocks-p01.pddl`. El objetivo pide `a` sobre `b` y `b` sobre `c`. El bloque `c` empieza encima de `a` y tiene que acabar abajo del todo, así que lo primero que hay que hacer no tiene nada que ver con ninguna de las dos partes del objetivo: poner `c` sobre la mesa.

El Listado 2.5 es el modelo STRIPS habitual, con cuatro acciones: dos para la mesa (`pick-up`, `put-down`) y dos para las torres (`unstack`, `stack`).

```
;; Blocks world with four actions: a hand moves blocks, one at a
;; time, between the table and the top of other blocks.
(define (domain blocks)
  (:requirements :strips)

  (:predicates
    (on ?x ?y)       ; block ?x is on block ?y
    (on-table ?x)    ; block ?x is on the table
    (clear ?x)       ; nothing is on block ?x
    (holding ?x)     ; the hand holds block ?x
    (hand-empty))    ; the hand holds nothing

  (:action pick-up              ; take a block from the table
    :parameters (?x)
    :precondition (and (on-table ?x) (clear ?x) (hand-empty))
    :effect (and (holding ?x)
                 (not (on-table ?x)) (not (clear ?x))
                 (not (hand-empty))))
```

```
19
20  (:action put-down             ; leave the block on the table
21    :parameters (?x)
22    :precondition (holding ?x)
23    :effect (and (on-table ?x) (clear ?x) (hand-empty)
24                 (not (holding ?x))))
25
26  (:action unstack              ; take ?x from the top of ?y
27    :parameters (?x ?y)
28    :precondition (and (on ?x ?y) (clear ?x) (hand-empty))
29    :effect (and (holding ?x) (clear ?y)
30                 (not (on ?x ?y)) (not (clear ?x))
31                 (not (hand-empty))))
32
33  (:action stack                ; leave ?x on top of ?y
34    :parameters (?x ?y)
35    :precondition (and (holding ?x) (clear ?y))
36    :effect (and (on ?x ?y) (clear ?x) (hand-empty)
37                 (not (holding ?x)) (not (clear ?y)))))
```

Listado 2.5: El mundo de bloques en STRIPS con cuatro acciones.

Hay tres cosas en las que fijarse. El predicado `(hand-empty)` no tiene parámetros: hay una sola mano, así que el hecho habla del mundo y no de ningún objeto. El predicado `(clear ?x)` es otro hecho sobre una ausencia, como `free` y `off`: no hay nada sobre `?x`. Y el dominio no tiene predicados de tipo, porque todos los objetos de un problema son bloques: cuando solo hay una clase de cosa, no hay nada que distinguir.

```
;; Three blocks: c is on a, b is alone. Build the tower a-b-c.
(define (problem blocks-p01)
  (:domain blocks)
  (:objects a b c)
  (:init (on-table a) (on-table b) (on c a)
         (clear b) (clear c) (hand-empty))
  (:goal (and (on a b) (on b c))))
```

Listado 2.6: Un problema de bloques con tres bloques.

```
$ kairos blocks-domain.pddl blocks-p01.pddl
; cost = 6
(unstack c a)
(put-down c)
(pick-up b)
(stack b c)
(pick-up a)
(stack a b)
```

Sigue `clear` a lo largo del plan, porque es la parte delicada del modelo. Al principio, `b` y `c` están despejados. `(unstack c a)` despeja `a` y deja `c` sin despejar (la mano lo sostiene, y un bloque sostenido no puede recibir otro). `(put-down c)` vuelve a despejar `c`. `(stack b c)` le quita esa condición para siempre. Cada acción que pone un bloque en algún sitio o lo quita de algún sitio tiene que actualizar `clear` para el bloque de debajo, y las cuatro lo hacen. El mundo de bloques es pequeño, pero lleva más contabilidad por acción que el mundo del mensajero: por eso es uno de los favoritos para mostrar lo que provoca un efecto olvidado.

La anomalía de Sussman

El problema de la Figura 2.5 tiene nombre. Es la *anomalía de Sussman*, por Gerald Sussman, que la estudió en el MIT a principios de los años setenta. Los planificadores de la época resolvían un objetivo de dos partes resolviendo primero una parte y después la otra. Aquí fallan los dos órdenes: si se construye primero `a` sobre `b`, `b` ya no puede ir sobre `c` sin deshacer el trabajo, y si se pone primero `b` sobre

`c`, `a` queda aún más enterrado bajo `c`. Las únicas soluciones intercalan las dos partes. Un rompecabezas de tres bloques demostró que, en general, las partes de un objetivo no se pueden planificar una a una.

El dominio nos da también una tercera clase de problema irresoluble. El fichero `blocks-p02.pddl` mantiene los mismos bloques y pide `(and (on a b) (on b a))`: cada bloque encima del otro.

```
$ kairos blocks-domain.pddl blocks-p02.pddl
note: no plan: every state reachable from the initial one was
     searched (40 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

Un modelo STRIPS no tiene dónde escribir las leyes de su mundo. Se cumplen porque ninguna acción, aplicada a un estado sensato, produce un estado que las rompa.

Nada en el dominio dice «un bloque no puede estar sobre un bloque que está sobre él». El modelo no necesita decirlo: las acciones están escritas de forma que nunca se produce un estado así. Así es como un modelo STRIPS expresa las leyes de su mundo: mediante lo que sus acciones pueden y no pueden hacer.

Gripper

Un robot con dos pinzas lleva bolas de una habitación a otra. El dominio procede de la primera competición de planificación McDermott, 2000. Se parece al mundo del mensajero a propósito: lo usamos para ver el mismo problema modelado con otras decisiones.

```
;; Gripper: a robot with two grippers carries balls between rooms.
(define (domain gripper)
  (:requirements :strips)

  (:predicates
    (room ?x) (ball ?x) (gripper ?x)   ; the kinds of things
    (at-robby ?r)                      ; the robot is in room ?r
    (at ?b ?r)                         ; ball ?b is in room ?r
    (empty ?g)                         ; gripper ?g holds nothing
    (holds ?g ?b))                     ; gripper ?g holds ball ?b

  (:action move
    :parameters (?from ?to)
    :precondition (and (room ?from) (room ?to) (at-robby ?from))
    :effect (and (at-robby ?to)
                 (not (at-robby ?from))))

  (:action pick
    :parameters (?b ?r ?g)
    :precondition (and (ball ?b) (room ?r) (gripper ?g)
                       (at ?b ?r) (at-robby ?r) (empty ?g))
    :effect (and (holds ?g ?b)
                 (not (at ?b ?r)) (not (empty ?g))))

  (:action drop
    :parameters (?b ?r ?g)
    :precondition (and (ball ?b) (room ?r) (gripper ?g)
                       (holds ?g ?b) (at-robby ?r))
    :effect (and (at ?b ?r) (empty ?g)
                 (not (holds ?g ?b)))))
```

Listado 2.7: El dominio gripper en STRIPS.

Compáralo con el Listado 2.3.

- Hay un solo robot, así que el robot no es un objeto: `(at-robby ?r)` solo tiene como parámetro la habitación, mientras que el

mundo del mensajero tiene `(robot-at ?r ?l)`. Un modelo con menos parámetros es más pequeño, y no puede pasar a tener dos robots sin reescribirse.
- Las manos son objetos, `left` y `right`. «Llevar como mucho dos» se expresa con dos pinzas que sostienen como mucho una bola cada una: `(empty ?g)` cumple para una pinza la función que cumple `(free ?r)` para un robot mensajero. Una capacidad se puede modelar dando objetos a los sitios donde se sostienen las cosas.
- `move` no tiene `connected` en su precondición: el robot va de cualquier habitación a cualquier otra. No hay mapa, y por tanto no hay en el problema ninguna relación estática aparte de los predicados de tipo.

El problema `gripper-p01.pddl` tiene dos habitaciones y cuatro bolas que deben ir de `room-a` a `room-b`.

```
$ kairos gripper-domain.pddl gripper-p01.pddl
; cost = 13
(pick ball1 room-a right)
(move room-a room-b)
(drop ball1 room-b right)
(move room-b room-a)
(pick ball2 room-a left)
...
(drop ball3 room-b left)
; better plan 2
; cost = 11
(pick ball1 room-a left)
(pick ball2 room-a right)
(move room-a room-b)
(drop ball2 room-b right)
(drop ball1 room-b left)
(move room-b room-a)
(pick ball3 room-a right)
(pick ball4 room-a left)
(move room-a room-b)
(drop ball3 room-b right)
(drop ball4 room-b left)
```

Han salido dos planes. Kairos imprime un primer plan en cuanto lo encuentra, sigue buscando e imprime cada plan que mejora el anterior; el último plan impreso es el mejor que ha encontrado. El primer plan, de 13 pasos, lleva las dos primeras bolas de una en una. El segundo, de 11 pasos, llena siempre las dos pinzas. El Capítulo 3 explica por qué un planificador trabaja así; con la opción `--no-anytime` se detiene en el primer plan. Por ahora, la regla es leer el último plan.

El plan de 11 pasos usa las dos pinzas en todos los viajes, y nadie le ha dicho al planificador que lo haga. En el mundo del mensajero, la prueba de la trampa comprobaba que el robot no puede llevar dos artículos. Aquí llevar dos es legal, el modelo lo dice con sus dos pinzas, y el planificador lo aprovecha. La prueba de la trampa de gripper es la misma idea un paso más allá: llevar tres bolas en un viaje tiene que ser imposible.

2.8. El método MODEL: la primera pasada completa

Hemos recorrido una vez los pasos M, O, D y E sobre el mundo del mensajero. Esta sección resume lo que esa pasada ha añadido al método

y después da el último paso, L.

Lo que añade este capítulo

Los artefactos existen. La ficha del modelo, la tabla de estado, la tabla de acciones y la batería de pruebas ya no son nombres del Capítulo 1: son la caja de método del paso M, la Tabla 2.1, la Tabla 2.2 y la caja de método del paso E. Los capítulos posteriores no los repetirán enteros. Dirán qué cambia en ellos.

Estático frente a dinámico. El paso O pregunta ahora, de cada propiedad: ¿puede cambiarla alguna acción? Si no, es estática: se escribe una vez en el estado inicial, aparece en precondiciones y nunca en efectos. Si sí, es dinámica, y el paso D le debe una respuesta para cada acción. Una comprobación rápida de un dominio terminado es enumerar los predicados que aparecen en algún efecto: esos son los dinámicos, y la lista debe coincidir con la columna «Clase» de la tabla de estado. En el dominio del mensajero son `robot-at`, `item-at`, `carrying` y `free`, como estaba previsto.

La regla del marco. Los efectos de una acción deben *decir todo lo que cambia, y nada de lo que no cambia*. La segunda mitad es un regalo del lenguaje: lo que un efecto no menciona se queda como estaba. La primera mitad es obligación del modelador, y es fácil fallar en los cambios que no son el propósito de la acción. El propósito de `pick` es que el robot lleve el artículo. Que el artículo ya no esté en la estantería y que el robot ya no esté libre son consecuencias que el mundo da gratis y el modelo no. La rutina del paso D (recorrer cada propiedad dinámica para cada acción) existe para cazarlas.

La regla del marco: los efectos de una acción dicen todo lo que cambia y nada de lo que no cambia.

Preguntas que este capítulo añade al método

- **M** ¿Puedo escribir las cinco líneas de la ficha del modelo antes de escribir nada de PDDL? ¿Qué reglas del mundo («un artículo cada vez») enuncia la ficha?
- **O** ¿Esta propiedad es estática o dinámica? ¿Trata de una ausencia (`free`, `clear`, `off`) y por tanto necesita un hecho propio? ¿Se solapan dos predicados, y quién los mantiene acompasados?
- **D** Para cada acción y cada propiedad dinámica: ¿la acción la hace cierta, la hace falsa o la deja igual? ¿Cada hecho que pasa a ser cierto sustituye a uno que hay que borrar?
- **E** ¿Pasan `check`, la resolución y `validate` con un problema trivial, uno típico, uno sin plan y uno que tienta al planificador a hacer trampa?
- **L** ¿Qué otra codificación serviría? (Siguiente subsección.)

Alternativas comparadas

El paso L pregunta qué otra codificación serviría. Comparamos el dominio del mensajero, la codificación A, con otras dos que podría haber escrito un modelador razonable. Las tres son ficheros de `examples/ch02`, y las tres se han ejecutado sobre la prueba de la trampa, el problema de las dos cajas para la sala A.

Codificación B: sin coger ni dejar. Si lo único que importa es dónde acaban los artículos, se puede prescindir del agarre. El fichero `courier-carry-domain.pddl` no tiene `carrying` ni `free`. Además de `move`, tiene una acción que mueve juntos el robot y un artículo:

```
(:action carry
  :parameters (?r ?i ?from ?to)
  :precondition (and (robot ?r) (item ?i)
                     (location ?from) (location ?to)
                     (robot-at ?r ?from) (item-at ?i ?from)
                     (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (item-at ?i ?from)) (item-at ?i ?to))))
```

La regla «un artículo cada vez» ya no necesita ningún predicado: cada `carry` lleva consigo un solo artículo, por construcción.

```
$ kairos courier-carry-domain.pddl courier-carry-p05.pddl
; cost = 11
(move r1 dock lobby)
(move r1 lobby pharmacy)
(carry r1 meds2 pharmacy lobby)
(carry r1 meds2 lobby corridor)
(move r1 corridor lobby)
(move r1 lobby pharmacy)
(carry r1 meds1 pharmacy lobby)
(carry r1 meds1 lobby corridor)
(carry r1 meds2 corridor ward-a)
(move r1 ward-a corridor)
(carry r1 meds1 corridor ward-a)
```

Once pasos para los dos viajes. El plan es honrado y trae una sorpresa: deja la primera caja en el pasillo, vuelve a por la segunda y termina las dos desde allí. En este modelo, dejar un artículo y volver a cogerlo no cuesta nada, porque esas acciones no existen, así que llevar las cajas por relevos a través del pasillo es tan bueno como cualquier otra forma.

Codificación C: un solo predicado para toda posición. Un robot en una habitación y un artículo en una habitación son, los dos, «algo en algún sitio». `courier-at-domain.pddl` es la codificación A con los predicados `robot-at` e `item-at` fusionados en uno solo, `(at ?x ?l)`. Su plan para la prueba de la trampa es, paso a paso, el de la codificación A. La diferencia aparece cuando se quitan los predicados de tipo. En la codificación A son casi una formalidad: solo un robot puede aparecer en un hecho `robot-at`, así que `move` difícilmente podría aplicarse a otra cosa. En la codificación C son los que sostienen todo el peso. El fichero `courier-at-notypes-domain.pddl` es la codificación C sin ellos, y este es su plan para la entrega de `courier-p01`:

```
$ kairos courier-at-notypes-domain.pddl \
    courier-at-notypes-p01.pddl
; cost = 3
(move meds1 pharmacy lobby)
(move meds1 lobby corridor)
(move meds1 corridor ward-a)
```

Los medicamentos van andando solos hasta la sala. Para `move`, `(at meds1 pharmacy)` es exactamente igual que `(at r1 dock)`, y nada dice que `meds1` no sea un robot.

La Tabla 2.4 juzga las tres codificaciones según los cinco criterios. Los tamaños son los de la prueba de la trampa; el número de acciones instanciadas es el que muestra Kairos con la opción `-v`, y el Capítulo 3 enseña a calcularlo a mano.

Tabla 2.4.: Paso L: tres codificaciones STRIPS del mundo del mensajero, comparadas sobre la prueba de la trampa (`courier-p05` y sus versiones para B y C).

Criterio	A: `pick`, `drop`	B: `carry`	C: un `at`
Fidelidad	Un artículo cada vez, impuesto por `free`. Los planes dicen cuándo agarrar y soltar	Un artículo cada vez, por construcción. El agarre es invisible: el ejecutor tiene que deducirlo; ningún hecho dice que un robot sostiene un artículo	Como A, pero solo mientras cada acción conserve sus predicados de tipo
Tamaño	8 predicados, 3 acciones; 46 acciones instanciadas	6 predicados, 2 acciones; 42 acciones instanciadas; crece con artículos por conexiones	7 predicados, 3 acciones; 46 acciones instanciadas
Guía	Plan de 15 pasos	Plan de 11 pasos; muchos relevos equivalentes	Como A
Claridad	Se lee como la historia	El fichero más corto; los planes hay que interpretarlos	Un nombre menos; cuesta más ver las clases de cosas
Portabilidad	`:strips`	`:strips`	`:strips`

Una palabra sobre la fila Tamaño. En las codificaciones A y C, `pick` y `drop` dan cada una una acción instanciada por cada robot, artículo y lugar. En B, `carry` da una por cada robot, artículo y conexión. En nuestro mapa, con 8 lugares y 14 conexiones contando los dos sentidos, los totales son parecidos. En un mapa en el que cada habitación tiene muchos vecinos, B crece más deprisa.

Veredicto. Elige B cuando la única pregunta es qué artículos van adónde y en cuántos viajes, y nada en el mundo depende de sostenerlos: es el modelo más pequeño y sus planes son los más cortos. Elige C cuando muchas clases de cosas comparten la propiedad de estar en algún sitio y las acciones las tratan igual; con los tipos de verdad del Capítulo 4, su debilidad desaparece. Para el mundo del mensajero de este libro nos quedamos con A. Sus planes dicen todo lo que tiene que hacer el robot, agarre incluido, y el hecho `(carrying ?r ?i)` es uno que necesitarán los capítulos posteriores: una muestra que se contamina mientras se transporta, una carga que pesa sobre la batería, un carro que se empuja. B habría que tirarla a la primera de ellas.

2.9. Errores comunes

Cada error de esta sección es un fichero de `examples/ch02` que está mal a propósito, junto con lo que hace Kairos cuando se lo encuentra.

Un efecto de borrado olvidado

El error más frecuente al modelar en STRIPS, y además silencioso. En `courier-nodelete-domain.pddl` la acción `move` añade la nueva posición y se olvida de borrar la antigua:

```
16  (:action move
17    :parameters (?r ?from ?to)
```

```
    :precondition (and (robot ?r) (location ?from) (location ?to)
                       (robot-at ?r ?from) (connected ?from ?to))
    :effect (robot-at ?r ?to))
```

`check` no encuentra nada mal, porque no hay nada mal en la forma. La prueba de la trampa lo delata:

```
$ kairos courier-nodelete-domain.pddl courier-nodelete-p05.pddl
; cost = 8
(move r1 dock lobby)
(move r1 lobby pharmacy)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(pick r1 meds2 pharmacy)
(drop r1 meds2 ward-a)
(pick r1 meds1 pharmacy)
(drop r1 meds1 ward-a)
```

Ocho pasos donde hacen falta 15. Tras sus movimientos, el robot está a la vez en la base de carga, el vestíbulo, la farmacia, el pasillo y la sala A: deja una copia de sí mismo en cada habitación. Después coge en la farmacia y deja en la sala sin moverse. *Síntoma*: planes demasiado cortos, en los que algo está en dos sitios. *Remedio*: para cada efecto de adición de una propiedad dinámica que tiene un solo valor cada vez, escribe el borrado del valor antiguo.

Una precondición que falta

En `courier-nofree-domain.pddl` la acción `pick` no pide `(free ?r)`. Otra vez `check` queda satisfecho, y otra vez la prueba de la trampa no:

```
$ kairos courier-nofree-domain.pddl courier-nofree-p05.pddl
; cost = 9
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds2 pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds2 ward-a)
(drop r1 meds1 ward-a)
```

Este es el plan de 9 pasos del que avisaba la Figura 2.4. En los problemas típicos `courier-p01` y `courier-p03`, este dominio erróneo da los mismos planes que el correcto: solo una prueba construida para tentar al planificador saca a la luz el agujero. *Síntoma*: un plan que hace algo que la ficha del modelo prohíbe. *Remedio*: busca el paso que no debería haber sido posible y añade a su acción la precondición que impone el mundo.

Un predicado mal escrito

En `courier-typo-domain.pddl` la precondición de `move` dice `robot-in` donde debería decir `robot-at`.

```
$ kairos check courier-typo-domain.pddl
courier-typo-domain.pddl:18:24: warning: 'robot-in' is not
     declared: never true [dialect.undeclared_predicate]
 18 |                       (robot-in ?r ?from) (connected ?
     from ?to))
    |                       ^
valid, plannable
```

Kairos avisa y sigue adelante: para él, un predicado no declarado es un hecho que nunca es cierto, así que `move` nunca es aplicable y todas las entregas son irresolubles. Otros planificadores pueden tratar el mismo fichero como un error y detenerse. *Síntoma*: un aviso de `check`, y problemas sin plan cuya explicación menciona una acción que «nunca» se aplica. *Remedio*: lee los avisos de `check`, no solo su veredicto.

Paréntesis desequilibrados y variables sueltas

Un paréntesis que falta al final de `move` (`courier-paren-domain.pddl`) se detecta donde empieza la sección siguiente, no donde falta el paréntesis:

```
$ kairos check courier-paren-domain.pddl
courier-paren-domain.pddl:22:3: error: expected action keyword,
     found '(' [parse.unexpected_token]
 22 |   (:action pick
    |   ^
courier-paren-domain.pddl:22:3: note: a new section starts here:
      the previous one was probably not closed [parse.resuming]
 22 |   (:action pick
    |   ^
invalid
```

Mira *por encima* de la línea del mensaje. Un editor que resalte los paréntesis emparejados evita la mayoría de estos errores. Una variable que se usa y no es un parámetro (`courier-var-domain.pddl` escribe `?loc` en lugar de `?l` en un efecto) se señala en su sitio:

```
$ kairos check courier-var-domain.pddl
courier-var-domain.pddl:27:35: error: in action 'pick': variable
      ?loc is not a parameter or quantified variable [semantic.
     unbound_variable]
 27 |     :effect (and (not (item-at ?i ?loc)) (not (free ?r))
    |                                   ^
invalid
```

Un problema que no encaja con su dominio

`courier-bad-p01.pddl` tiene tres errores en su estado inicial: un lugar que no está en `:objects`, un hecho con menos argumentos de la cuenta y un predicado que no existe.

```
$ kairos check courier-domain.pddl courier-bad-p01.pddl
courier-bad-p01.pddl:11:5: warning: in the initial state: 'robot
     -at' takes 2 argument(s), 1 given; the fact is ignored [
     dialect.init_arity]
 11 |     (robot-at r1) (free r1)
    |     ^
courier-bad-p01.pddl:10:22: error: in the initial state: unknown
      object or constant 'ward-c' (did you mean 'ward-a'?) [
     semantic.unknown_object]
 10 |     (connected lobby ward-c)
    |                      ^
courier-bad-p01.pddl:12:5: warning: in the initial state:
     unknown predicate 'item-in'; the fact is ignored (did you
     mean 'item-at'?) [semantic.unknown_predicate]
 12 |     (item-in meds1 pharmacy))
    |     ^
invalid
```

El objeto desconocido es un error. Los otros dos son avisos, y su texto dice lo que hace Kairos: *the fact is ignored*, el hecho se ignora. Un hecho ignorado del estado inicial es, por la hipótesis del mundo cerrado, un hecho falso: el robot no está en ninguna parte, y los medicamentos tampoco. *Síntoma*: un problema irresoluble que parece resoluble. *Remedio*: ejecuta `check` con el dominio *y* el problema, y no dejes ningún aviso sin leer.

Un objetivo que ya es cierto

Cuando el objetivo se cumple en el estado inicial, el plan está vacío. `courier-p07.pddl` pide los medicamentos en la farmacia, donde ya están:

```
$ kairos courier-domain.pddl courier-p07.pddl
; cost = 0
```

Un plan de coste 0 sin pasos es una respuesta correcta, y suele ser señal de un objetivo copiado de la línea equivocada o de una prueba que no prueba nada. *Remedio*: para cada prueba, conoce la longitud esperada del plan antes de ejecutarla.

2.10. Resumen

Un modelo STRIPS dice con predicados qué puede ser cierto y con acciones qué se puede hacer; un problema da los objetos, el estado inicial y el objetivo. La Tabla 2.5 recoge el PDDL del capítulo.

Construcción	Significado
`(define (domain d) ...)`	Un fichero de dominio, llamado `d`
`(:requirements :strips)`	Las características del lenguaje que usa el fichero
`(:predicates (p ?x ?y) ...)`	Lo que se puede decir: cada predicado con sus parámetros
`(:action a ...)`	Algo que se puede hacer
`:parameters (?x ?y)`	Las variables de la acción
`:precondition (and ...)`	Hechos que deben ser ciertos para que la acción se aplique
`:effect (and f (not g))`	Hechos que pasan a ser ciertos (adición) y falsos (borrado)
`(define (problem p) ...)`	Un fichero de problema, llamado `p`
`(:domain d)`	El dominio al que pertenece el problema
`(:objects a b c)`	Los objetos que existen
`(:init ...)`	Los hechos ciertos al principio; todos los demás son falsos
`(:goal (and ...))`	Hechos que deben ser ciertos al final; el resto queda libre

Tabla 2.5.: El PDDL de este capítulo: todo un modelo STRIPS.

Las ideas que hay detrás de la sintaxis:

- Un **hecho** es un predicado aplicado a objetos; un **estado** es el conjunto de los hechos que son ciertos. Lo que no está en el estado es falso: la **hipótesis del mundo cerrado**.
- Una acción es **aplicable** cuando los hechos de su precondición están en el estado. Aplicarla quita sus efectos de borrado y añade sus efectos de adición; todo lo demás se queda igual.
- Un **plan** es una secuencia de acciones instanciadas, cada una aplicable tras las anteriores, que acaba en un estado que contiene el objetivo.
- Las condiciones de STRIPS solo piden hechos ciertos. Una ausencia necesita su propio hecho (`free`, `off`, `clear`), y las acciones tienen que mantenerlo acompasado con el resto.
- Los hechos **estáticos** son datos: aparecen en el estado inicial y en las precondiciones, nunca en los efectos. Los hechos **dinámicos** son lo que cambian los planes.
- La **regla del marco**: un efecto dice todo lo que cambia y nada de lo que no cambia.
- El paso E es una rutina: `kairos check`, resolver, leer, `kairos validate`, sobre una **batería de pruebas** con un problema trivial, uno típico, uno sin plan y uno que tienta al planificador a hacer trampa.

Hemos usado el planificador sin preguntarnos cómo encuentra sus planes ni por qué a veces imprime dos. El capítulo siguiente abre esa caja: lo que un planificador hace con un modelo decide qué modelos le resultan fáciles, y por eso forma parte del modelado.

2.11. Lecturas complementarias

La descripción de las acciones mediante precondiciones, listas de adición y listas de borrado procede del artículo de STRIPS de Fikes y Nilsson Fikes y Nilsson, 1971. La sintaxis es la del primer manual de PDDL McDermott et al., 1998, donde `:strips` abre la lista de requisitos. Kovacs Kovacs, 2011 da la gramática completa del lenguaje actual en unas pocas páginas, útiles cuando surge una duda de sintaxis. Haslum et al. Haslum et al., 2019 dedican su segundo capítulo al mismo subconjunto del lenguaje, con una definición formal de estados, acciones y planes que complementa la

definición informal que damos aquí, y Ghallab, Nau y Traverso Ghallab et al., 2004 lo presentan como la «representación clásica», junto a otras dos equivalentes. Gripper y los demás dominios de la primera competición los describe McDermott McDermott, 2000. La validación de planes como herramienta en sí misma empezó con VAL Howey et al., 2004, el validador de las competiciones de planificación, cuyo formato de planes lee y escribe Kairos.

2.12. Ejercicios

Ejercicios de la Sección 2.1 — Listas dentro de listas: la forma de PDDL

Ejercicio 2.1.1. Cada una de estas líneas pretende ser un fragmento de PDDL y tiene un error de forma. Encuéntralo y escribe la línea correctamente: (a) `(and (on desk) (off hall);` (b) `(on, desk);` (c) `(:goal and (on desk) (off hall));` (d) `(on desk) ; (off hall))`. Después dibuja el árbol de la línea (a) ya corregida, al estilo de la Figura 2.1.

Ejercicios de la Sección 2.2 — Predicados, objetos y hechos

Ejercicio 2.2.1. El dominio de los interruptores con tres lámparas tiene seis hechos posibles. (a) ¿Cuántos conjuntos de hechos distintos hay? (b) ¿Cuántos de ellos describen una situación con sentido, con cada lámpara encendida o apagada? (c) Da un conjunto que no tenga sentido y di qué línea de un fichero de problema podría producirlo.

Ejercicio 2.2.2. Un estado de un problema del mensajero es {`(robot-at r1 lab)`, `(carrying r1 sample1)`, `(item-at meds1 lobby)`} más los hechos estáticos de `courier-p01.pddl`. Según la hipótesis del mundo cerrado, di si cada hecho es cierto, falso o desconocido: (a) `(free r1)`; (b) `(item-at sample1 lab)`; (c) `(connected lab corridor)`; (d) `(connected lab icu)`; (e) `(robot-at r1 corridor)`.

Ejercicios de la Sección 2.3 — Acciones: precondiciones y efectos

Ejercicio 2.3.1. A mano, con el dominio del mensajero del Listado 2.3. Los hechos dinámicos de un estado son `(robot-at r1 lobby)`, `(free r1)`, `(item-at meds1 lobby)` y `(item-at sample1 lab)`; los estáticos son los de `courier-p01.pddl`, con `sample1` declarado como artículo. (a) ¿Cuáles de estas acciones instanciadas son aplicables: `(move r1 lobby lab)`, `(pick r1 meds1 lobby)`, `(pick r1 sample1 lab)`, `(drop r1 meds1 lobby)`, `(move r1 lobby dock)`? Para cada una que no lo sea, nombra el hecho de la precondición que falla. (b) Escribe el estado que resulta de aplicar `(pick r1 meds1 lobby)` y después `(move r1 lobby corridor)`.

Ejercicio 2.3.2. Un estudiante escribe `pick` con el efecto `(and (not (free ?r)) (carrying ?r ?i))`. Aplica a mano esta acción al estado inicial de `courier-p02.pddl` y completa después la entrega con un movimiento y la acción de dejar. ¿Qué tiene de extraño el estado final? ¿Qué prueba de la batería lo notaría, y cómo?

Ejercicios de la Sección 2.4 — El problema: objetos, estado inicial y objetivo

Ejercicio 2.4.1. Delante del ordenador. Escribe un problema del dominio de los interruptores con el estado inicial de `switch-p01.pddl` en el que las tres lámparas deban acabar encendidas. Predice el plan y después ejecútalo.

Ejercicio 2.4.2. Delante del ordenador. Quita `(off porch)` del estado inicial de `switch-p01.pddl` y cambia el objetivo por `(on porch)`. Predice la respuesta del planificador, ejecútalo y explica la nota que imprime Kairos.

Ejercicio 2.4.3. Las lámparas dependen de un interruptor general: una lámpara solo se puede encender mientras hay corriente. Amplía el dominio de los interruptores con lo necesario (puede haber corriente o no haberla, y una acción la da), y escribe un problema en el que al principio no hay corriente y dos lámparas deben acabar encendidas. Pista: un predicado puede no tener parámetros, y una acción también: `:parameters ()`.

Ejercicios de la Sección 2.5 — Modelar el mundo del mensajero

Ejercicio 2.5.1. Delante del ordenador. Escribe un problema para el dominio del mensajero: el robot empieza en el pasillo; `sample1` está en el laboratorio y debe ir a la UCI; `meal1` está en el vestíbulo y debe ir a la sala B. Antes de ejecutarlo, busca a mano el plan más corto que puedas y cuenta sus pasos. Compáralo con el último plan que imprime Kairos.

Ejercicio 2.5.2. La puerta de la sala A se cambia por una que solo se abre hacia dentro: el robot puede entrar en la sala y no puede salir. ¿Qué fichero cambia, el dominio o el problema, y qué hechos? Haz el cambio en una copia de `courier-p01.pddl`, pide además que el robot acabe en la base de carga y explica la respuesta del planificador.

Ejercicio 2.5.3. Añade al dominio del mensajero una acción `hand-over`: un robot que lleva un artículo se lo da a un robot libre que está en el mismo lugar. Rellena primero su fila de la tabla de acciones, recorriendo las cuatro propiedades dinámicas, y escribe después el PDDL. Pruébala con un problema en el que `r1` sostiene `meds1` en la farmacia, `r2` está libre en el vestíbulo y el objetivo es `(carrying r2 meds1)`.

Ejercicios de la Sección 2.6 — Ensayar el modelo

Ejercicio 2.6.1. Delante del ordenador. Escribe en un fichero un plan para `courier-p02.pddl` que sea válido y tenga más pasos que el que encuentra Kairos, y valídalo. Después escribe el plan `(move r1 pharmacy lobby)`, `(drop r1 meds1 lobby)`, predice qué dirá el validador y en qué paso, y compruébalo.

Ejercicio 2.6.2. Diseña una batería de pruebas para el dominio de los interruptores: un problema trivial, uno típico, uno que no debe tener plan y uno que tienta al planificador a hacer trampa. Para cada uno, indica el resultado esperado antes de ejecutarlo. (Para los dos últimos, piensa en qué es imposible para una lámpara.)

Ejercicios de la Sección 2.7 — Dos clásicos: el mundo de bloques y gripper

Ejercicio 2.7.1. Delante del ordenador. Escribe un problema de bloques con cuatro bloques: `d` sobre `c`, `c` sobre `b`, `b` sobre `a` y `a` sobre la mesa. El objetivo es la misma torre del revés. Predice el número de pasos y ejecútalo.

Ejercicio 2.7.2. En una copia del dominio de bloques, quita el efecto de borrado `(not (clear ?y))` de `stack`. (a) Predice un objetivo que pase a ser alcanzable y que ningún juego de bloques real pueda cumplir. (b) Escríbelo como problema y ejecútalo con el dominio erróneo y con el correcto. (c) ¿De cuál de las cuatro clases de pruebas es tu problema?

Ejercicio 2.7.3. Modela a partir de esta historia, siguiendo los pasos M a E: *un ferri lleva coches entre puertos. Puede navegar de cualquier puerto a cualquier otro. Lleva un coche cada vez: un coche embarca en el ferri en el puerto donde están los dos, y desembarca en el puerto donde está el ferri.* Escribe la ficha del modelo en tres líneas, la tabla de estado, la tabla de acciones, el dominio y un problema en el que dos coches que están en puertos distintos deben intercambiar sus puertos.

Ejercicios de la Sección 2.8 — El método MODEL: la primera pasada completa

Ejercicio 2.8.1. Para el dominio de bloques y para el dominio gripper, enumera los predicados estáticos y los dinámicos, con la comprobación de la sección del método (¿aparece el predicado en algún efecto?). Después, para `unstack`, recorre cada predicado dinámico del dominio de bloques y di si la acción lo hace cierto, lo hace falso o lo deja igual, y para qué objetos.

Ejercicio 2.8.2. El paso L en el mundo de bloques. Una codificación alternativa no tiene mano: tres acciones mueven un bloque despejado en un solo paso, de un bloque a otro bloque, de un bloque a la mesa y de la mesa a un bloque. Escríbela, resuelve con ella el problema de la Figura 2.5 y compárala con el dominio de cuatro acciones según los cinco criterios. Usa `kairos -v` para ver el número de acciones instanciadas de cada una.

Ejercicios de la Sección 2.9 — Errores comunes

Ejercicio 2.9.1. Encuentra el fallo. Un dominio del mensajero es idéntico al del Listado 2.3 salvo en el efecto de `drop`, que es `(and (not (carrying ?r ?i)) (item-at ?i ?l))`. (a) Sin ejecutar nada, di cuáles de las pruebas, de `courier-p01` a `courier-p05`, siguen dando la respuesta correcta y cuáles no. (b) ¿Qué responde el planificador en las que fallan? (c) Este error y la precondición que falta de la caja de error típico tienen que ver, los dos, con `free`. ¿Por qué uno produce planes demasiado cortos y el otro ningún plan?

Qué hace un planificador con tu modelo

3.

En el capítulo anterior escribimos un modelo del mundo del mensajero, se lo dimos a Kairos y recibimos un plan en unos pocos milisegundos. El planificador era una caja negra, y para un robot y una caja de medicamentos una caja negra basta. Deja de bastar muy pronto. Demos al mismo dominio cuatro entregas y dos robots: el primer plan sigue llegando en milisegundos, pero después el planificador trabaja durante tres segundos, sobre más de medio millón de estados, antes de decir que no existe un plan mejor. Quitemos una precondición por error, y el número de acciones que maneja el planificador pasa de 30 a 80 mientras los planes se vuelven imposibles de llevar a cabo. Un modelador que no sabe qué ocurre dentro de la caja no puede explicarse por qué, ni elegir entre dos modelos que parecen igual de correctos.

Este capítulo abre la caja. Primero precisa qué *significa* un modelo: qué estados existen, cuándo se puede aplicar una acción, qué produce y qué cuenta como plan. Después sigue al modelo a través del planificador: cómo las acciones con variables se convierten en acciones sobre objetos concretos, cuántos estados crea eso, cómo una búsqueda encuentra un camino entre ellos y cómo una heurística, calculada a partir del propio modelo, le dice a la búsqueda dónde mirar. Aquí no se demuestra nada; todo se muestra sobre problemas lo bastante pequeños para dibujarlos o contarlos a mano, y después se mide con Kairos.

El capítulo tiene nueve secciones. Las tres primeras definen el significado de un modelo, la instanciación de sus acciones y el tamaño de su espacio de estados. Las dos siguientes explican la búsqueda hacia delante y las heurísticas. Otras dos tratan de lo que responde el planificador: un plan, un plan mejor, el mejor plan o ningún plan. Las dos últimas secciones convierten todo ello en consejos para el modelador y en dos añadidos al método MODEL: estimar el tamaño de un modelo antes de ejecutarlo y poner cifras a los criterios Tamaño y Guía. El capítulo no usa PDDL nuevo: solo el STRIPS del Capítulo 2, sobre el mundo del mensajero, el puzle deslizante y las torres de Hanói.

3.1 Qué significa un modelo . 49
Un problema lo bastante pequeño para dibujarlo . . 49
Estados 50
Acciones aplicables y sucesores 50
Planes 51
El espacio de estados . . . 52
3.2 Instanciación: de variables a objetos 52
Contar a mano 53
Lo que dice el planificador 53
La planta entera, el puzle y las torres 54
3.3 Por qué explota el espacio 56
Contar estados 56
3.4 Búsqueda hacia delante . 57
El bucle 57
Buscar a ciegas 58
3.5 Heurísticas: plantear un problema más sencillo . . 58
Un número para cada estado 58
Ignorar los borrados 59
La heurística en todo el espacio 60
Cuando la relajación ve poco 61
3.6 Un plan, un plan mejor, el mejor plan 61
Anytime: el primer plan y luego otros mejores 61
Elegir cuánto pedir 63
3.7 No hay plan, o no se ha encontrado 63
Demostrado por la instanciación 63
Demostrado por la búsqueda 64
No encontrado 64
3.8 Qué significa esto para el modelador 65
Los parámetros multiplican 65
Las precondiciones estáticas podan 66
Callejones sin salida 66
Simetría 67
3.9 El método MODEL: tamaño y guía en cifras 68
Alternativas comparadas . 68
3.10 Errores comunes 70
3.11 Resumen 71
3.12 Lecturas complementarias 72
3.13 Ejercicios 73

3.1. Qué significa un modelo

Un modelo PDDL es un texto, y un planificador es un programa que lo lee. Para que los dos estén de acuerdo, y para que dos planificadores estén de acuerdo entre sí, el texto debe tener un único significado. Esta sección lo establece para STRIPS. Los capítulos posteriores lo amplían, pero nunca lo cambian.

Un problema lo bastante pequeño para dibujarlo

La planta del hospital tiene ocho habitaciones, demasiadas para un primer dibujo. La reducimos a tres habitaciones en fila, con el robot en la del medio (Figura 3.1). El dominio es el del Capítulo 2, sin cambios; el problema es el Listado 3.1.

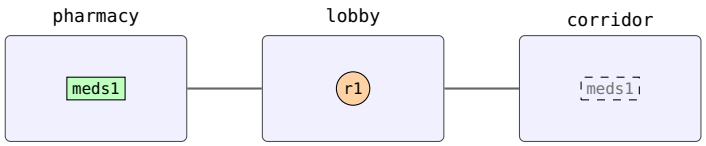


Figura 3.1.: El problema diminuto de este capítulo. El robot `r1` empieza en el vestíbulo con las manos vacías; los medicamentos están en la farmacia y deben acabar en el pasillo (en trazo discontinuo).

```
;; The tiny problem of chapter 3: three rooms in a row, one robot,
;; one item. Small enough to draw its whole state space.
(define (problem courier-p01)
  (:domain courier)
  (:objects r1 meds1 pharmacy lobby corridor)
  (:init
    (robot r1) (item meds1)
    (location pharmacy) (location lobby) (location corridor)
    (connected pharmacy lobby) (connected lobby pharmacy)
    (connected lobby corridor) (connected corridor lobby)
    (robot-at r1 lobby) (free r1)
    (item-at meds1 pharmacy))
  (:goal (item-at meds1 corridor)))
```

Listado 3.1: El problema diminuto: tres habitaciones, un robot, un artículo.

Estados

Un **hecho** es un predicado aplicado a objetos, como `(robot-at r1 lobby)`. Un **estado** es un conjunto de hechos: los que son verdaderos. Todo hecho que no está en el conjunto es falso; es la hipótesis del mundo cerrado del Capítulo 2. El estado inicial del problema diminuto tiene 16 hechos, los que aparecen en `:init`.

Trece de esos hechos no pueden cambiar nunca, porque ninguna acción los tiene en su efecto: los tres hechos de tipo, como `(location lobby)`, y el mapa. Son los mismos en todos los estados, así que los omitimos al escribir un estado. Lo que queda es breve:

$$s_0 = \{\ \texttt{(robot-at r1 lobby)},\ \texttt{(item-at meds1 pharmacy)},\ \texttt{(free r1)}\ \}$$

Acciones aplicables y sucesores

Una acción del dominio tiene parámetros. Al sustituir cada parámetro por un objeto se obtiene una **acción instanciada**, como `(move r1 lobby pharmacy)`. Una acción instanciada a tiene tres conjuntos de hechos: sus precondiciones $pre(a)$, los hechos que hace verdaderos $add(a)$ y los hechos que hace falsos $del(a)$. Para `(move r1 lobby pharmacy)`:

pre	`(robot r1)` `(location lobby)` `(location pharmacy)` `(robot-at r1 lobby)` `(connected lobby pharmacy)`
add	`(robot-at r1 pharmacy)`
del	`(robot-at r1 lobby)`

Una acción instanciada a es **aplicable** en un estado s cuando todas sus precondiciones están en s: $pre(a) \subseteq s$. Aplicarla da el estado **sucesor**: se quitan los hechos borrados y después se ponen los añadidos.

$$s' = (s \setminus del(a)) \cup add(a)$$

Todo lo demás se queda como estaba. Ese es todo el significado de una acción en STRIPS: tres conjuntos y una fórmula.

Un estado es el conjunto de los hechos verdaderos. Una acción se aplica cuando sus precondiciones están en el estado, y cambia el estado borrando unos hechos y añadiendo otros.

En s_0 hay dos acciones instanciadas aplicables: `(move r1 lobby pharmacy )` y `(move r1 lobby corridor)`. La acción `(pick r1 meds1 pharmacy)` no lo es, porque `(robot-at r1 pharmacy)` no está en s_0. Aplicar el primer movimiento da

$$s_1 = \{\ \texttt{(robot-at r1 pharmacy)},\ \texttt{(item-at meds1 pharmacy)}, \texttt{(free r1)}\ \}$$

y en s_1 la recogida es aplicable.

Planes

En STRIPS el objetivo también es un conjunto de hechos. Un estado s **satisface** el objetivo G cuando $G \subseteq s$: los hechos del objetivo son verdaderos, y el objetivo no dice nada del resto. Un **plan** es una secuencia de acciones instanciadas $a_1, a_2, \ldots, a_n$ tal que a_1 es aplicable en el estado inicial, cada acción siguiente es aplicable en el estado que produjo la anterior y el último estado satisface el objetivo. Ni más ni menos: un plan puede ser más largo de lo necesario, y puede dar vueltas.

La orden `kairos validate` comprueba exactamente esta definición. Escribimos un plan a mano en el fichero `courier-p01.plan`, una acción instanciada por línea, y preguntamos:

```
$ cat courier-p01.plan
; A plan for courier-p01, written by hand.
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(drop r1 meds1 corridor)
$ kairos validate courier-domain.pddl courier-p01.pddl courier-p01.
    plan
plan valid, makespan 5
```

La palabra *makespan* corresponde a los planes con tiempo (Capítulo 10); para una secuencia es el número de pasos. Una secuencia puede no ser un plan de dos maneras, y el validador las distingue. En `courier-p01-shortcut.plan` quitamos la tercera acción, de modo que el robot intenta salir de una habitación en la que no está:

```
$ kairos validate courier-domain.pddl courier-p01.pddl courier-p01-
    shortcut.plan
courier-p01-shortcut.plan:4: error: step 3 (move r1 lobby corridor)
    at 2.000: precondition false: (robot-at r1 lobby) is false [
    validate.precondition_failed]
 4 | (move r1 lobby corridor)
plan invalid
```

En `courier-p01-halfway.plan` todas las acciones son aplicables, pero el robot deja los medicamentos en el vestíbulo:

```
$ kairos validate courier-domain.pddl courier-p01.pddl courier-p01-
    halfway.plan
courier-p01-halfway.plan: error: the goal does not hold at the end
    of the plan: (item-at meds1 corridor) is false [validate.
    goal_not_reached]
plan invalid
```

El espacio de estados

Tomemos el estado inicial, apliquemos todas las acciones aplicables, hagamos lo mismo en cada estado nuevo y sigamos hasta que no aparezca nada nuevo. El resultado es un grafo: sus nodos son los **estados alcanzables**, y hay una flecha de s a s', etiquetada con una acción instanciada, cuando esa acción es aplicable en s y produce s'. Este grafo es el **espacio de estados** del problema. Un plan es un camino en él desde el estado inicial hasta cualquier estado que satisfaga el objetivo.

Para el problema diminuto, el grafo entero cabe en la Figura 3.2. Un estado queda determinado por dos cosas: dónde está el robot (tres posibilidades) y dónde están los medicamentos (en una de las tres habitaciones o en las manos del robot: cuatro posibilidades). Eso da $3 \times 4 = 12$ estados.

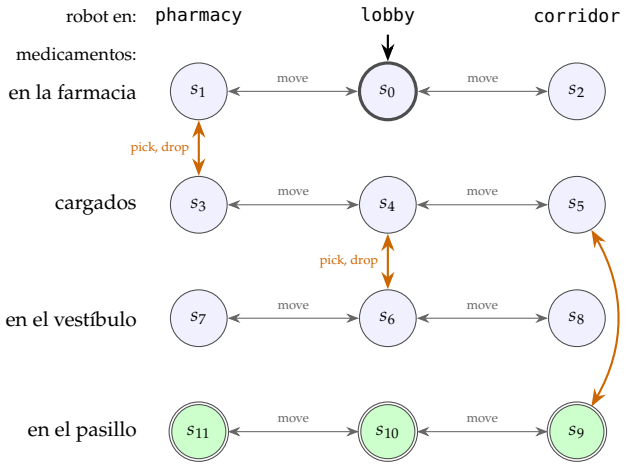


Figura 3.2.: El espacio de estados del problema diminuto: 12 estados. Cada flecha doble representa dos acciones instanciadas, una en cada sentido (flechas gruesas: `pick` y `drop`). El estado inicial es s_0; los tres estados de la fila inferior satisfacen el objetivo. El plan escrito a mano es el camino $s_0, s_1, s_3, s_4, s_5, s_9$.

Hay tres cosas en la figura que merecen una segunda mirada. Primero, hay varios estados objetivo: el objetivo dice dónde están los medicamentos, no dónde acaba el robot. Segundo, todas las flechas son dobles, así que nada en este problema es irreversible; no siempre será así. Tercero, el espacio de estados no está en los ficheros PDDL. Está *implícito* en ellos, y el planificador tiene que construir la parte que necesita.

3.2. Instanciación: de variables a objetos

La búsqueda trabaja con acciones instanciadas, pero el dominio tiene acciones con parámetros. La mayoría de los planificadores empiezan por generar todas las acciones instanciadas que podrían llegar a necesitarse, un paso llamado **instanciación** (*grounding*); Kairos también lo hace, como la mayoría de los planificadores. La Figura 3.3 muestra dónde se sitúa.

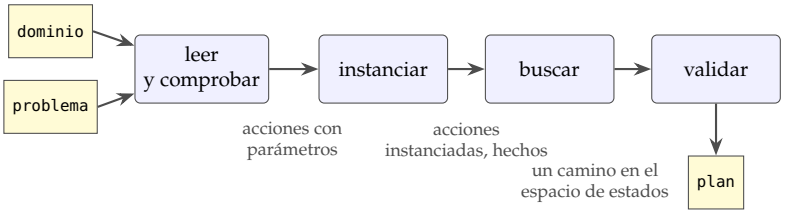


Figura 3.3.: Las etapas por las que pasa un modelo. La instanciación convierte las pocas acciones del dominio en las muchas acciones instanciadas de un problema; la búsqueda busca un camino entre los estados que estas crean.

Contar a mano

¿Cuántas acciones instanciadas tiene el problema diminuto? La primera respuesta es: todas las formas de sustituir los parámetros por objetos. El problema tiene 5 objetos (`r1`, `meds1` y tres habitaciones) y cada acción tiene 3 parámetros, así que cada acción tiene $5^3 = 125$ instancias, y las tres acciones, 375.

La mayoría de esas 375 no tienen sentido: `(move meds1 r1 r1)` es una de ellas. El planificador descarta una instancia en cuanto una de sus precondiciones es un hecho que nunca puede ser verdadero. Los hechos de tipo y el mapa deciden la mayoría de los casos, porque nunca cambian:

- `move ?r ?from ?to` necesita `(robot ?r)` y `(connected ?from ?to)`. Hay 1 robot, y el mapa tiene 4 hechos `connected`: $1 \times 4 = 4$ instancias.
- `pick ?r ?i ?l` necesita un robot, un artículo y una habitación: $1 \times 1 \times 3 = 3$ instancias.
- `drop ?r ?i ?l`, lo mismo: 3 instancias.

Son $4 + 3 + 3 = 10$ acciones instanciadas, de 375 candidatas. Los hechos que pueden cambiar se cuentan del mismo modo: `robot-at` para 1 robot y 3 habitaciones (3), `item-at` para 1 artículo y 3 habitaciones (3), `carrying` (1) y `free` (1): 8 hechos.

Lo que dice el planificador

La opción `-v` hace que Kairos informe de lo que hace, por la salida de error estándar. En este capítulo añadimos siempre `--threads 1`: con un solo hilo las ejecuciones son reproducibles, y las cifras impresas aquí son las que obtendrá el lector. (Los tiempos serán distintos; dependen de la máquina.)

```
$ kairos courier-domain.pddl courier-p01.pddl -v --threads 1
...
[   0.001s] [ground] grounded in 0.1 ms: 8 atoms, 0 numeric fluents,
      10 actions, 0 durative actions, 0 axioms (50 Datalog facts, 17
      rules, 0 instances pruned)
[   0.001s] [sas] SAS+ in 0.1 ms: 3 variables (2 from 3 mutex groups,
      1 binary), 3 invariants from 13 candidates, 0 axiom layers
[   0.001s] [orchestrator] search: lama, then novelty (auto)
[   0.001s] [orchestrator] search: 3 variables, 10 actions, search
     lama, costs unit
[   0.001s] [search] plan 0 with 5 actions, cost 5
; cost = 5
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(drop r1 meds1 corridor)
[   0.001s] [search] plan of cost 5 with 5 actions (6 expanded, 7
     evaluated)
[   0.001s] [orchestrator] solved (proven optimal) in 1.1 ms
kairos: solved (proven optimal), cost 5
kairos: parse 0.2 ms, normalize 0.1 ms, ground 0.2 ms, sas 0.1 ms,
     search 0.6 ms, total 1.1 ms, first plan 0.9 ms
kairos: 8 atoms, 10 actions (0 durative), 3 variables (2 mutex
     groups), 10 states, 13 expanded, 14 evaluated, 1 plans, 13958.4
      bytes per state, 7 MB peak
```

La línea `[ground]` confirma la cuenta: 8 átomos (**átomo** es otro nombre de un hecho instanciado) y 10 acciones. Volveremos sobre las demás

líneas a lo largo del capítulo. Para ver las propias acciones instanciadas, `--dump=ground` las escribe en el fichero `ground.txt`. Dos de las diez:

```
$ kairos courier-domain.pddl courier-p01.pddl --threads 1 --dump=
    ground --dump-dir out > /dev/null
$ cat out/ground.txt
...
(:action (move r1 lobby corridor)
  :precondition (robot-at r1 lobby)
  :effect
    (not (robot-at r1 lobby))
    (robot-at r1 corridor))
...
(:action (pick r1 meds1 lobby)
  :precondition (and (robot-at r1 lobby) (item-at meds1 lobby) (free
     r1))
  :effect
    (not (item-at meds1 lobby))
    (not (free r1))
    (carrying r1 meds1))
...
```

Dos detalles muestran cómo razona el instanciador. El movimiento ha perdido cuatro de sus cinco precondiciones: `(robot r1)`, los dos hechos `location` y `(connected lobby corridor)` se comprobaron una vez, durante la instanciación, y no hace falta volver a comprobarlos. Un hecho que ninguna acción cambia es un hecho **estático**, y una precondición sobre un hecho estático no cuesta nada durante la búsqueda. Y `(pick r1 meds1 lobby)` está ahí aunque los medicamentos no empiezan en el vestíbulo: podrían dejarse allí más tarde, así que la acción puede hacer falta.

La instanciación sustituye los parámetros por objetos de todas las formas que permiten los hechos estáticos, y conserva las acciones que podrían llegar a ser aplicables en algún momento.

El instanciador decide qué «podría hacer falta» con un análisis rápido y optimista: supone que un hecho, una vez verdadero, sigue siéndolo. Eso conserva unas pocas acciones que nunca pueden aplicarse, como enseguida mostrarán las torres de Hanói, pero nunca desecha una que un plan necesite.

Hechos que se excluyen entre sí

La línea `[sas]` dice «3 variables». El robot está siempre en exactamente una habitación, así que de los tres hechos `(robot-at r1 pharmacy)`, `(robot-at r1 lobby)` y `(robot-at r1 corridor)` exactamente uno es verdadero en cualquier estado alcanzable. Los planificadores encuentran esos grupos de hechos mutuamente excluyentes (*grupos mutex*) analizando las acciones, y guardan una variable con varios valores en lugar de varios hechos verdadero-o-falso: una variable para dónde está el robot, otra para dónde están los medicamentos (tres habitaciones, o en las manos del robot), otra para `(free r1)`. El fichero que escribe `--dump=sas` las muestra. Esta representación procede del planificador Fast Downward (Helmert, 2006, 2009); volveremos a encontrarla en el Capítulo 16, donde PDDL permite al modelador escribir directamente esas variables.

La planta entera, el puzle y las torres

La misma cuenta sirve para el problema del Capítulo 2, que aquí es `courier-p02`. Tiene un robot, un artículo y las ocho habitaciones, y su mapa tiene 7 conexiones, así que 14 hechos `connected`. Movimientos:

$1 \times 14 = 14$. Recogidas: $1 \times 1 \times 8 = 8$. Descargas: 8. Total: 30 acciones instanciadas, y $8 + 8 + 1 + 1 = 18$ átomos. Kairos coincide:

```
$ kairos courier-domain.pddl courier-p02.pddl -v --threads 1
...
[   0.002s] [ground] grounded in 0.4 ms: 18 atoms, 0 numeric fluents,
      30 actions, 0 durative actions, 0 axioms (130 Datalog facts,
     17 rules, 0 instances pruned)
...
```

Los otros dos dominios del capítulo son clásicos. En el **puzle deslizante** (Listado 3.2) ocho fichas numeradas ocupan un tablero de 3×3, y una ficha contigua a la casilla vacía puede deslizarse a ella. Su única acción, `slide ?t ?from ?to`, necesita una ficha y dos casillas adyacentes. El tablero tiene 12 pares de vecinas, así que 24 hechos `adjacent`: $8 \times 24 = 192$ acciones instanciadas. Los átomos son `tile-at` para 8 fichas y 9 casillas (72) y `empty` para 9 casillas: 81.

```
;; The sliding puzzle (8-puzzle): eight tiles on a 3x3 board and
;; one empty cell. A tile next to the empty cell slides into it.
(define (domain puzzle)
  (:requirements :strips)

  (:predicates
    (tile ?x)               ; ?x is a tile
    (cell ?x)               ; ?x is a cell of the board
    (adjacent ?c1 ?c2)      ; the cells share a side
    (tile-at ?t ?c)         ; tile ?t is on cell ?c
    (empty ?c))             ; no tile is on cell ?c

  (:action slide
    :parameters (?t ?from ?to)
    :precondition (and (tile ?t) (cell ?from) (cell ?to)
                       (adjacent ?from ?to)
                       (tile-at ?t ?from) (empty ?to))
    :effect (and (not (tile-at ?t ?from)) (not (empty ?to))
                 (tile-at ?t ?to) (empty ?from))))
```

Listado 3.2: El puzle deslizante.

En las **torres de Hanói** (Listado 3.3) una torre de discos debe pasar de una varilla a otra, moviendo un disco cada vez y sin poner nunca un disco sobre otro más pequeño. La codificación trata por igual varillas y discos, como cosas sobre las que puede descansar un disco, y enumera en el predicado estático `smaller` qué puede descansar sobre qué. El Listado 3.4 es el problema con tres discos.

```
;; Towers of Hanoi: discs on three pegs. Only the top disc of a
;; peg moves, and never onto a smaller one.
(define (domain hanoi)
  (:requirements :strips)

  (:predicates
    (disc ?x)               ; ?x is a disc (pegs are not)
    (smaller ?x ?y)         ; ?x may rest on ?y (larger disc or peg)
    (on ?x ?y)              ; disc ?x rests on ?y (a disc or a peg)
    (clear ?x))             ; nothing rests on ?x

  (:action move
    :parameters (?d ?from ?to)
    :precondition (and (disc ?d) (smaller ?d ?to)
                       (on ?d ?from) (clear ?d) (clear ?to))
    :effect (and (not (on ?d ?from)) (not (clear ?to))
                 (on ?d ?to) (clear ?from))))
```

Listado 3.3: Las torres de Hanói.

```
;; Towers of Hanoi, 3 discs (d1 is the smallest):
```

```
;; the whole tower goes from peg1 to peg3.
(define (problem hanoi-p02)
  (:domain hanoi)
  (:objects peg1 peg2 peg3 d1 d2 d3)
  (:init
    (disc d1) (disc d2) (disc d3)
    ; every disc is smaller than the pegs
    (smaller d1 peg1) (smaller d1 peg2) (smaller d1 peg3)
    (smaller d2 peg1) (smaller d2 peg2) (smaller d2 peg3)
    (smaller d3 peg1) (smaller d3 peg2) (smaller d3 peg3)
    ; and than the discs after it
    (smaller d1 d2) (smaller d1 d3)
    (smaller d2 d3)
    ; the tower on peg1
    (on d1 d2) (on d2 d3) (on d3 peg1)
    (clear d1) (clear peg2) (clear peg3))
  (:goal (and
    (on d1 d2) (on d2 d3) (on d3 peg3))))
```

Listado 3.4: Tres discos, de `peg1` a `peg3`.

Contemos `move ?d ?from ?to` para tres discos. El disco más pequeño, `d1`, puede descansar sobre 5 cosas (dos discos y tres varillas), así que tanto `?from` como `?to` tienen 5 posibilidades: 25 instancias. Para `d2` hay 4 cosas: 16. Para `d3`, las tres varillas: 9. Total: $25 + 16 + 9 = 50$, y eso es lo que indica Kairos. La cuenta incluye `(move d1 peg2 peg2)`, que nunca puede aplicarse: necesita a la vez `d1` sobre `peg2` y `peg2` despejada. El análisis optimista del instanciador no ve la contradicción. Estas acciones gastan un poco de memoria y, por lo demás, son inofensivas.

3.3. Por qué explota el espacio

Las acciones instanciadas crecen con el problema, pero con suavidad. Los estados no. Esta sección cuenta unas y otros a medida que el mensajero recibe más trabajo.

Contar estados

Tomemos la planta entera con un robot y n artículos. Un estado queda determinado por la habitación del robot (8 posibilidades) y por dónde está cada artículo. O bien el robot no lleva nada, y cada artículo está en una de 8 habitaciones (8^n formas); o bien lleva uno de los n artículos y los demás están en habitaciones ($n \cdot 8^{n-1}$ formas). Todos estos estados son alcanzables, ya que el robot puede llevar cualquier artículo a cualquier sitio. Así pues:

$$\text{estados} = 8 \cdot \left(8^n + n \cdot 8^{n-1}\right)$$

Para un artículo, eso da $8 \cdot 9 = 72$. Cada artículo nuevo multiplica el número por más de 8. La Tabla 3.1 pone la fórmula junto a lo que indica Kairos para los problemas `courier-p02` a `courier-p05`, y para `courier-p06`, que añade un segundo robot a las cuatro entregas (para dos robots, el mismo razonamiento da $64 \cdot (8^4 + 8 \cdot 8^3 + 12 \cdot 8^2)$).

Problema	Átomos	Acciones instanciadas	Estados alcanzables
`p02`: 1 artículo	18	30	72
`p03`: 2 artículos	27	46	640
`p04`: 3 artículos	36	62	5632
`p05`: 4 artículos	45	78	49 152
`p06`: 4 artículos, 2 robots	58	156	573 440

Tabla 3.1.: Tamaño de los problemas del mensajero. Los átomos y las acciones instanciadas son los que indica Kairos con `-v`; los estados alcanzables se cuentan con la fórmula del texto. Cada artículo añade 9 átomos y 16 acciones, y multiplica los estados por unos 9.

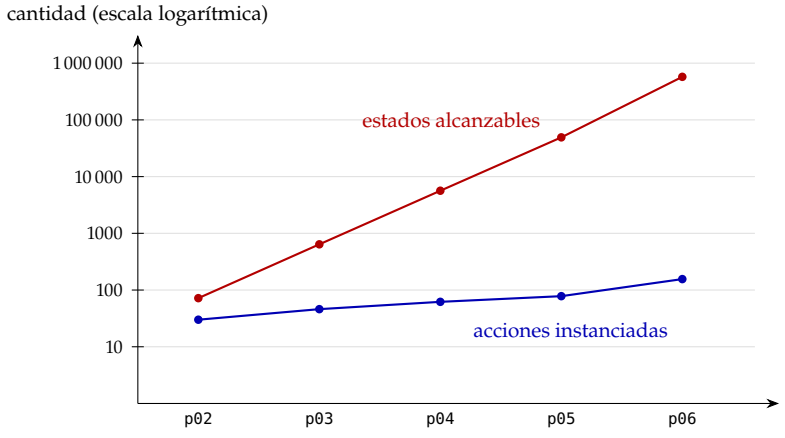


Figura 3.4.: Las cifras de la Tabla 3.1 en escala logarítmica. El modelo crece por suma (la línea inferior es casi plana); el espacio de estados crece por multiplicación.

Lo que escribe el modelador crece despacio con el problema; el espacio de estados que implica se multiplica con cada objeto nuevo que puede estar en varios sitios.

El puzle y las torres cuentan la misma historia. Un tablero del puzle deslizante es una disposición de 8 fichas y un hueco en 9 casillas: 9! = 362 880 disposiciones, de las que exactamente la mitad, 181 440, son alcanzables desde una dada (la otra mitad exigiría que dos fichas saltaran una por encima de la otra). En las torres de Hanói cada disco está en una de tres varillas y el orden en una varilla es forzoso, así que n discos tienen 3^n estados: 27 para tres discos, 59 049 para diez, 531 441 para doce. En los tres dominios, el texto PDDL tiene las mismas pocas líneas sea cual sea el tamaño.

¿Cómo de difícil es planificar?

La explosión no es un defecto de los planificadores actuales que una ingeniería mejor vaya a eliminar. Bylander demostró en 1994 que decidir si un problema STRIPS, ya instanciado, tiene un plan es PSPACE-completo (Bylander, 1994). Se cree que los problemas de esa clase son más difíciles que los famosos NP-completos: un plan puede necesitar un número de pasos exponencial en el tamaño del modelo (las torres de Hanói con n discos necesitan $2^n - 1$ movimientos), así que incluso escribir la respuesta puede llevar un tiempo exponencial. Ningún planificador puede ser rápido con todos los modelos. La consecuencia práctica es el tema de este capítulo: los planificadores son rápidos con los modelos que dan a su búsqueda algo a lo que agarrarse, y el modelador decide parte de eso.

3.4. Búsqueda hacia delante

Un planificador no puede construir un grafo de medio millón de estados para encontrar en él un camino de 27 pasos, al menos no como primer movimiento. Construye el grafo mientras lo explora, empezando por el estado inicial. Es la **búsqueda hacia delante**, la forma en que trabajan la mayoría de los planificadores actuales, y la única que describimos.[1]

1: Los planificadores también han buscado hacia atrás desde el objetivo, en el espacio de los planes parciales, y mediante traducciones a la lógica. Ghallab et al. (2004) los tratan todos.

El bucle

La búsqueda mantiene dos colecciones: la **lista abierta**, con los estados que ha encontrado y aún no ha explorado, y el conjunto de todos los estados que ha encontrado, para no explorar ninguno dos veces. Repite cuatro pasos:

1. Saca un estado de la lista abierta. Si la lista está vacía, para: no hay plan.
2. Si el estado satisface el objetivo, para: el camino que ha llevado hasta él es un plan.
3. **Expande** el estado: encuentra las acciones instanciadas aplicables en él y calcula el sucesor de cada una.
4. Mete en la lista abierta los sucesores que no se habían encontrado antes, cada uno con una nota del estado y la acción de los que procede.

Todo lo que distingue una búsqueda de otra está en el paso 1: *qué* estado sacar. Dos cifras miden el trabajo hecho: los estados **expandidos** (paso 3) y, cuando se usa una heurística, los estados **evaluados** con ella. Kairos imprime ambas con `-v`.

Buscar a ciegas

La elección más sencilla es sacar el estado más antiguo de la lista abierta. Es la búsqueda en anchura: explora todos los estados a una acción del inicio, después todos los que están a dos acciones, y así sucesivamente. Sigámosla sobre la Figura 3.2. Expande s_0 y encuentra s_1 y s_2. Expande s_1 (encuentra s_3) y s_2 (nada nuevo). Después s_3 (encuentra s_4), después s_4 (encuentra s_5 y s_6), después s_5 (encuentra s_9), después s_6 (encuentra s_7 y s_8). El siguiente estado que saca de la lista es s_9, que satisface el objetivo. Siete expansiones, y el plan que devuelve, de 5 acciones, es tan corto como puede serlo un plan, porque antes se probaron todos los caminos más cortos.

La búsqueda en anchura no usa ningún conocimiento del problema: se extiende en todas las direcciones como el agua. En el problema diminuto eso apenas importa, porque hay pocas direcciones. En `courier-p05`, cuyo plan más corto tiene 28 acciones, tendría que recorrer la mayor parte de los 49 152 estados antes de encontrar un estado objetivo. Una búsqueda que no sabe nada de dónde está el objetivo paga la explosión entera.

La búsqueda hacia delante construye el espacio de estados a medida que lo explora. Lo que hace buena o mala una búsqueda es el orden en que elige el siguiente estado que expandir.

3.5. Heurísticas: plantear un problema más sencillo

Una persona que mira la Figura 3.1 no se extiende como el agua: ir primero al pasillo es a todas luces una pérdida de tiempo, porque los medicamentos están al otro lado. El planificador necesita la misma clase de vista, y tiene que obtenerla solo del modelo, ya que no sabe nada de hospitales.

Un número para cada estado

Una **heurística** es una función h que da a cada estado una estimación de cuántas acciones hacen falta todavía para alcanzar el objetivo desde él. Con una heurística, el paso 1 del bucle se convierte en: saca el estado de la lista abierta con la h más pequeña. Es la **búsqueda voraz primero el mejor**. Si las estimaciones fueran exactas, la búsqueda iría derecha al objetivo, expandiendo solo los estados del plan. No son exactas, y el arte está en conseguir estimaciones baratas de calcular y lo bastante buenas.

Ignorar los borrados

La idea que cambió la planificación en torno al año 2000 es obtener la estimación resolviendo una versión más sencilla del mismo problema, una **relajación**, y la simplificación que funciona es sorprendentemente burda: *ignorar todos los efectos de borrado*. En el problema relajado, un hecho, una vez verdadero, sigue siéndolo para siempre. El robot que se mueve del vestíbulo a la farmacia está entonces en las dos habitaciones; un artículo que se recoge está en la mano y sigue en la estantería.

El problema relajado es fácil porque nunca se pierde nada: el conjunto de hechos verdaderos solo crece, así que no hay movimientos equivocados ni necesidad de buscar. Un plan para él, un **plan relajado**, se puede construir en dos pasadas. Hacia delante, reunimos por capas los hechos que se hacen verdaderos. Desde el s_0 del problema diminuto (Figura 3.5):

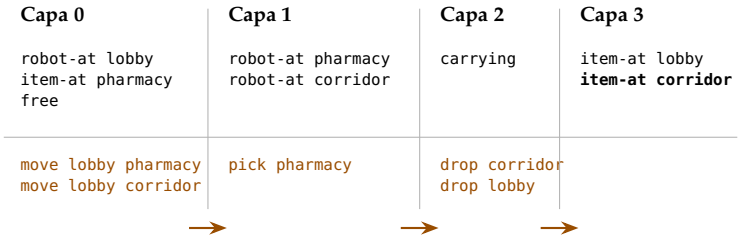


Figura 3.5.: El problema relajado desde s_0. Cada capa muestra los hechos nuevos (sin los nombres de objeto `r1` y `meds1`); bajo la línea, en color, las acciones aplicadas en esa capa, que producen la siguiente. Nunca se quita nada, así que a partir de la capa 1 el robot está en tres habitaciones. El hecho del objetivo aparece en la capa 3.

Hacia atrás desde el objetivo, elegimos para cada hecho necesario una acción que lo produzca, y después pedimos las precondiciones de esa acción. `(item-at meds1 corridor)` viene de `(drop r1 meds1 corridor)`, que necesita `(carrying r1 meds1)` y `(robot-at r1 corridor)`. El primero viene de la recogida, que necesita al robot en la farmacia, lo que viene de `(move r1 lobby pharmacy)`. El segundo viene de `(move r1 lobby corridor)`; el robot sigue en el vestíbulo, porque no se borra nada. El plan relajado tiene cuatro acciones:

```
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 lobby corridor)     ; from the lobby: it never left
(drop r1 meds1 corridor)
```

El valor heurístico es la longitud del plan relajado: $h(s_0) = 4$. La respuesta real es 5. La relajación ha olvidado el camino de vuelta de la farmacia al vestíbulo, ya que un robot que nunca sale de una habitación nunca necesita volver a ella.

La relajación de borrados finge que los hechos nunca se vuelven falsos. Sus planes son fáciles de encontrar, y su longitud estima a qué distancia está el objetivo en el problema real.

La planificación como búsqueda heurística

Estimar distancias con la relajación de borrados lo introdujeron Bonet y Geffner en el planificador HSP (Bonet & Geffner, 2001). El FF de Hoffmann y Nebel (Hoffmann & Nebel, 2001) extraía un plan relajado explícito, como acabamos de hacer, y usaba su longitud como heurística y sus acciones como pistas de qué probar primero; FF fue el planificador automático con más éxito de la competición de planificación de 2000. Los planificadores que le siguieron mantuvieron el esquema y lo refinaron: Fast Downward (Helmert, 2006) y LAMA (Richter & Westphal, 2010), que añade *landmarks* (hitos), hechos que todo plan tiene que hacer verdaderos en algún momento. Las heurísticas que Kairos usa por defecto para un problema como el nuestro son las dos de LAMA: el plan relajado

de FF y una cuenta de los hitos que quedan por alcanzar.

La heurística en todo el espacio

Podemos repetir el ejercicio en cada uno de los 12 estados. La Figura 3.6 muestra, para cada estado, la longitud de su plan relajado y, tras la barra, la distancia real al objetivo.

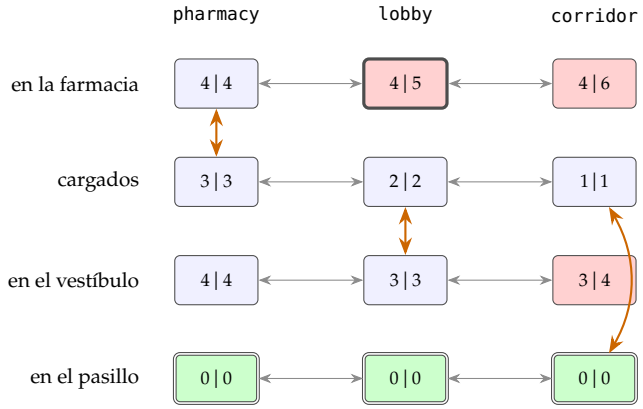


Figura 3.6.: El espacio de estados de la Figura 3.2 con, en cada estado, la longitud de su plan relajado y la distancia real al objetivo. Difieren en tres estados (en rojo), siempre porque la relajación olvida que el robot tiene que volver. El estado inicial es el del borde grueso.

La estimación es exacta en nueve estados y se queda corta en tres. Más importante que su precisión es su forma: a lo largo del plan los valores son 4, 4, 3, 2, 1, 0, siempre cuesta abajo, así que una búsqueda voraz se ve conducida al objetivo. Expande s_0 y ve dos sucesores, ambos con valor 4. Es una **meseta**: la heurística no dice qué camino es mejor, y la búsqueda tiene que probar. Si prueba primero s_1, después baja en línea recta: s_3, s_4, s_5 y el objetivo.

Kairos muestra este descenso con `--debug=search`, que imprime una línea cada vez que la búsqueda encuentra un estado con un valor más bajo que todos los anteriores. En la planta entera (`courier-p02`, cuyo plan más corto tiene 7 acciones):

```
$ kairos courier-domain.pddl courier-p02.pddl --debug=search --
    threads 1 --no-anytime
...
[   0.001s] [search] engine 0: heuristics ff,lmcount
[   0.001s] [search] initial h: ff 6.000000, lmcount 6.000000
[   0.001s] [search] new best h 5 (1 expanded, 2 evaluated)
[   0.001s] [search] new best h 4 (4 expanded, 5 evaluated)
[   0.001s] [search] new best h 3 (5 expanded, 6 evaluated)
[   0.001s] [search] new best h 2 (6 expanded, 7 evaluated)
[   0.001s] [search] new best h 1 (7 expanded, 8 evaluated)
[   0.001s] [search] new best h 0 (8 expanded, 9 evaluated)
[   0.001s] [search] plan 0 with 7 actions, cost 7
...
```

La estimación del estado inicial es 6 para una distancia real de 7 (el paso olvidado es, una vez más, la vuelta de la farmacia al vestíbulo), y la búsqueda alcanza el objetivo tras expandir 8 de los 72 estados. En `courier-p05` el primer plan llega tras 79 expansiones, en un espacio de 49 152 estados. Esa es toda la razón de que los planificadores funcionen.

Cuando la relajación ve poco

La relajación de borrados es buena en problemas donde la dificultad está en *conseguir* cosas, y mala donde la dificultad está en que las cosas *estorban*, porque estorbar se hace con borrados. Las torres de Hanói son el caso extremo. Tomemos tres discos. En el problema relajado, apartar `d1` deja despejado `d2`, y `d1` nunca vuelve a tapar nada; una varilla, una vez despejada, sigue despejada. Así que el plan relajado mueve cada disco una vez: `d1` a `peg2`, `d2` a `peg3`, `d3` a `peg3`, y ya está, porque `(on d1 d2)` y `(on d2 d3)` eran verdaderos al principio y nada los borra. La estimación es 3; la respuesta real es 7. La Tabla 3.2 muestra que la diferencia crece sin límite: la estimación para n discos es n, la realidad es $2^n - 1$.

Discos	Estados (3^n)	h inicial	Primer plan	Plan más corto
3	27	3	14 acciones, 23 expandidos	7
4	81	4	18 acciones, 28 expandidos	15
6	729	6	77 acciones, 284 expandidos	63
8	6561	8	357 acciones, 3170 expandidos	255
10	59 049	10	1322 acciones, 22 333 expandidos	1023

Tabla 3.2.: Las torres de Hanói con Kairos (problemas `hanoi-p02` a `hanoi-p06`). La estimación del estado inicial es el valor `ff` que imprime `--debug=search`. Con tan poca guía, la primera búsqueda expande una buena parte del espacio: más de un tercio para diez discos.

Comparemos la última fila con el mensajero: allí, 79 expansiones en un espacio de 49 152 estados; aquí, 22 333 en un espacio de 59 049. Los dos espacios tienen más o menos el mismo tamaño. Lo que cambia es cuánto ve la heurística de cada uno, y eso depende del problema y de cómo se modela.

3.6. Un plan, un plan mejor, el mejor plan

La búsqueda voraz va adonde apunta la heurística y devuelve el primer plan que encuentra. Nada hace que ese plan sea corto. La planificación distingue dos ambiciones. La planificación **satisfactoria** quiere un plan, razonable si es posible, encontrado deprisa. La planificación **óptima** quiere un plan con la garantía de que no existe otro mejor, y paga por la garantía. Hasta el Capítulo 9, «mejor» significa «con menos acciones».

Anytime: el primer plan y luego otros mejores

Kairos sigue un camino intermedio, llamado búsqueda ***anytime***: devuelve el primer plan en cuanto lo tiene, y sigue buscando otros mejores mientras haya tiempo. Cada plan mejor se imprime al encontrarse, así que el último plan impreso es el mejor. Un tablero difícil del puzle deslizante lo muestra (el fichero `puzzle-p02.pddl`; aquí los planes aparecen recortados):

```
$ kairos puzzle-domain.pddl puzzle-p02.pddl --threads 1
; cost = 95
(slide t5 c22 c32)
(slide t6 c12 c22)
...
; better plan 2
; cost = 51
...
; better plan 3
; cost = 39
...
```

```
; better plan 4
; cost = 31
(slide t1 c33 c32)
(slide t4 c23 c33)
...
```

Cuatro planes: 95 deslizamientos, después 51, 39 y 31. Con `-v` vemos cuándo llegó cada uno y cómo terminó la ejecución:

```
$ kairos puzzle-domain.pddl puzzle-p02.pddl --threads 1 -v
...
[   0.006s] [search] plan of cost 95 with 95 actions (440 expanded,
     441 evaluated)
[   0.010s] [search] plan of cost 51 with 51 actions (712 expanded,
     900 evaluated)
[   0.016s] [search] plan of cost 39 with 39 actions (1055 expanded,
      1477 evaluated)
[   0.050s] [search] plan of cost 31 with 31 actions (3239 expanded,
      5084 evaluated)
[   1.485s] [orchestrator] solved (proven optimal) in 1484.7 ms
kairos: solved (proven optimal), cost 31
kairos: parse 0.1 ms, normalize 0.1 ms, ground 0.3 ms, sas 0.2 ms,
     search 1484.0 ms, total 1484.7 ms, first plan 5.3 ms
kairos: 81 atoms, 192 actions (0 durative), 9 variables (9 mutex
     groups), 181440 states, 189149 expanded, 186522 evaluated, 4
     plans, 52.2 bytes per state, 25 MB peak
```

Fijémonos en los tiempos. El primer plan llevó 6 milisegundos y 440 expansiones. El plan de 31 deslizamientos se encontró a los 50 milisegundos. Después el planificador trabajó casi un segundo y medio más sin imprimir nada, y terminó con `proven optimal`: había buscado por todas partes un plan de 30 deslizamientos o menos, y no lo hay. Para estar seguro tuvo que encontrar 181 440 estados: todo el espacio alcanzable, la cifra que calculamos en la Sección 3.3.

Encontrar un plan, encontrar un buen plan y demostrar que un plan es el mejor son tres trabajos distintos, cada uno mucho más caro que el anterior.

Cómo funcionan las búsquedas de mejora queda fuera de este capítulo; en pocas palabras, cada una solo puede devolver un plan más barato que el mejor hasta el momento, y se fía de la heurística menos que la anterior, de modo que explora más ampliamente.[2] La última se fía tan poco que acaba mirando por todas partes. La Tabla 3.3 muestra los mismos tres trabajos en los problemas del mensajero.

2: La documentación de Kairos da los detalles: reinicios con A* ponderado de peso decreciente, como hace LAMA (Richter & Westphal, 2010).

	Primer plan		Demostración del mejor plan	
Problema	Acciones	Expandidos	Acciones	Estados encontrados
`courier-p02`	7	8	7	33
`courier-p03`	13	21	13	397
`courier-p04`	19	47	19	5054
`courier-p05`	28	79	28	48 048
`courier-p06`	31	393	27	544 039

Tabla 3.3.: El primer plan y la demostración de que un plan es el más corto, con Kairos y un hilo. El primer plan cuesta unas pocas decenas de expansiones; la demostración obliga al planificador a encontrar la mayoría de los estados alcanzables de la Tabla 3.1.

En `courier-p06`, el problema con dos robots, el primer plan (31 acciones) llega a los 4 milisegundos, el mejor (27 acciones) a los 10, y la demostración de que 27 es el mínimo a los 3,3 segundos en nuestra máquina. Para un robot que espera sus órdenes, las dos primeras cifras importan y la tercera no.

Elegir cuánto pedir

Tres opciones dicen cuándo parar. `--no-anytime` devuelve el primer plan y termina. `--timeout S` da a la búsqueda S segundos (30 si no se indica nada) y se queda con el mejor plan encontrado en ellos. `--target-cost C` se detiene en el primer plan de coste C o menor:

```
$ kairos puzzle-domain.pddl puzzle-p02.pddl --threads 1 --target-
     cost 40 -v
...
[   0.016s] [search] plan of cost 39 with 39 actions (1055 expanded,
      1477 evaluated)
[   0.017s] [orchestrator] solved (target reached) in 16.5 ms
kairos: solved (target reached), cost 39
...
```

Las palabras entre paréntesis tras `solved` dicen por qué terminó la búsqueda: `proven optimal`, `target reached`, `timeout` o `no improvement` (que es también lo que indica `--no-anytime`). Solo la primera es una garantía sobre el plan.

3.7. No hay plan, o no se ha encontrado

Cuando el planificador vuelve sin un plan, el modelador debe saber cuál de dos cosas muy distintas ha ocurrido: el problema no *tiene* plan, o el planificador no ha *encontrado* ninguno en el tiempo del que disponía. Lo primero es un hecho sobre el modelo, y suele revelar un error en él. Lo segundo es un hecho sobre la búsqueda.

Demostrado por la instanciación

En `courier-p07.pddl` los medicamentos deben ir a la UCI, y el mapa ha perdido los dos hechos que conectan `ward-b` e `icu`:

```
$ kairos courier-domain.pddl courier-p07.pddl --threads 1
note: the goal needs atoms that no action can reach
(item-at meds1 icu) can never be reached:
  drop adds it, but needs (robot-at ?r - object icu), which never
     holds
    move adds it, but needs (connected ?from - object icu), which
     never holds
      no action adds it, and the initial state does not have it [
     orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
$ echo $?
10
```

No hizo falta buscar. El análisis optimista de la instanciación, el que supone que los hechos nunca se vuelven falsos, no pudo hacer verdadero el hecho del objetivo ni aun así. Si el objetivo no se puede alcanzar cuando nunca se pierde nada, no se puede alcanzar en absoluto. La nota se lee como una cadena de razones, desde el objetivo hasta el hecho que falta, y es lo primero que hay que leer cuando un modelo que debería tener plan no lo tiene.

Demostrado por la búsqueda

En `courier-p08.pddl` la puerta de la UCI solo se abre desde fuera: el mapa tiene `(connected ward-b icu)` pero no el hecho opuesto. Una muestra espera en la UCI y debe ir al laboratorio.

```
$ kairos courier-domain.pddl courier-p08.pddl --threads 1
note: no plan: every state reachable from the initial one was
     searched (8 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

La instanciación no ve aquí nada raro: con los borrados ignorados, el robot entra en la UCI, recoge la muestra y sigue en el pasillo para llevarla al laboratorio. Solo la búsqueda lo descubre. Encuentra 8 estados, el robot en cada una de las ocho habitaciones. El de dentro de la UCI se desecha sin expandirlo: desde ahí ni siquiera el problema relajado tiene plan, ya que ningún movimiento empieza en la UCI. Los otros siete se expanden, no aparece nada nuevo y la lista abierta se queda vacía. Una lista abierta vacía es una demostración: se examinaron todos los estados que podían llevar a alguna parte.

Una demostración por búsqueda cuesta el espacio de estados entero. El tablero `puzzle-p03.pddl` es el objetivo con las fichas 7 y 8 intercambiadas, una de las disposiciones que no se pueden alcanzar:

```
$ kairos puzzle-domain.pddl puzzle-p03.pddl --threads 1
note: no plan: every state reachable from the initial one was
     searched (181440 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Tarda un segundo y medio. Una heurística no puede acortar esto: dice dónde mirar primero, y aquí el planificador tiene que mirar por todas partes.

No encontrado

Las torres de Hanói con doce discos tienen un plan, de 4095 movimientos. Con dos segundos:

```
$ kairos hanoi-domain.pddl hanoi-p07.pddl --threads 1 --timeout 2
note: no plan within the time limit of 2 s (73411 states searched).
     More time may find one: --timeout (SolveOptions::timeout). More
      threads run the phases of the search at once: --threads (
     SolveOptions::threads). [orchestrator.why]
kairos: timeout (timeout)
$ echo $?
11
```

El estado es `timeout`, no `unsolvable`, y el código de salida es 11, no 10. El planificador no afirma nada sobre el problema. (Con los 30 segundos por defecto encuentra un primer plan en unos 5; el número de estados de la nota depende de la velocidad de la máquina.)

La Tabla 3.4 reúne las respuestas vistas en este capítulo. El Apéndice A las enumera todas.

«Unsolvable» es una demostración de que no existe ningún plan, por la instanciación o agotando la búsqueda. «Timeout» solo dice que la búsqueda se detuvo; el problema bien puede tener un plan.

Línea de estado	Código de salida	Significado
`solved (...)`	0	al menos un plan; el paréntesis dice por qué terminó la búsqueda
`unsolvable (exhausted)`	10	demostrado: no existe ningún plan
`timeout (timeout)`	11	sin tiempo antes del primer plan; no se sabe nada

Tabla 3.4.: Lo que responde Kairos a un problema STRIPS. Los scripts leen el código de salida; las personas leen la línea de estado y la nota que la precede.

3.8. Qué significa esto para el modelador

El planificador es general, pero no le da igual cómo esté escrito un modelo. Cuatro efectos se siguen directamente de los mecanismos de este capítulo. Cada uno se mide a continuación sobre el mundo del mensajero.

Los parámetros multiplican

Cada parámetro de una acción multiplica sus instancias por el número de objetos que caben en él. Una acción con k parámetros sobre n objetos tiene hasta n^k instancias; las precondiciones sobre hechos estáticos son lo que reduce ese número.

Un parámetro que nada restringe no se reduce en absoluto. El dominio `courier-extra-domain.pddl` es el dominio del mensajero con un cambio, de la clase que sobrevive a un copiar y pegar: `drop` tiene un cuarto parámetro, `?to`, que su precondición y su efecto nunca mencionan.

```
  (:action drop
    :parameters (?r ?i ?l ?to)     ; ?to is never used
    :precondition (and (robot ?r) (item ?i) (location ?l)
                       (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l) (free ?r))))
```

Listado 3.5: Un parámetro de más (de `courier-extra-domain.pddl`).

Con las tres entregas de `courier-p04`, el dominio correcto tiene 62 acciones instanciadas. Este:

```
$ kairos courier-extra-domain.pddl courier-extra-p01.pddl --threads
    1 --no-anytime -v
...
[   0.001s] [ground] grounded in 0.3 ms: 36 atoms, 0 numeric fluents,
     326 actions, 0 durative actions, 0 axioms (504 Datalog facts,
    18 rules, 0 instances pruned)
...
(move r1 corridor ward-a)
(drop r1 meds1 ward-a dock)
...
```

Las 24 descargas se han convertido en $24 \times 12 = 288$, una por cada uno de los 12 objetos del problema en el lugar de `?to`, y $14 + 24 + 288 = 326$. Los planes siguen siendo correctos, y `kairos check` no encuentra nada que objetar. Los síntomas son el salto en la línea `[ground]` y un objeto que no pinta nada en la acción, `dock`, al final de cada descarga del plan.

Las precondiciones estáticas podan

Una precondición sobre un hecho estático se comprueba durante la instanciación y después no cuesta nada, así que es la forma más barata de decirle al planificador lo que no puede ocurrir. Omitir una tiene dos costes. En el dominio de courier-anywhere-domain.pddl, la acción `move` no pide `(connected ?from ?to)`:

```
$ kairos courier-anywhere-domain.pddl courier-anywhere-p01.pddl --
     threads 1 -v
...
[   0.001s] [ground] grounded in 0.1 ms: 18 atoms, 0 numeric fluents,
      80 actions, 0 durative actions, 0 axioms (166 Datalog facts,
     16 rules, 0 instances pruned)
...
; cost = 4
(move r1 dock pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy ward-a)
(drop r1 meds1 ward-a)
...
```

Los movimientos pasaron de 14 a $8 \times 8 = 64$ (la cuenta incluye movimientos de una habitación a sí misma, que ahora nada prohíbe), el modelo de 30 acciones instanciadas a 80, y el robot atraviesa paredes. Es la prueba de la trampa del Capítulo 2 vista desde el lado del planificador: el modelo que permite demasiado es a la vez erróneo y más grande.

Callejones sin salida

Un **callejón sin salida** es un estado desde el que no se puede alcanzar el objetivo. Un modelo tiene callejones sin salida cuando alguna acción no se puede deshacer. El mundo del mensajero de la Figura 3.2 no tiene ninguno; la puerta de sentido único de la UCI los crea (Figura 3.7). En `courier-p09.pddl` el robot debe llevar los medicamentos a la UCI y una muestra del laboratorio a la sala A. Una vez dentro de la UCI nunca sale, así que todo estado con el robot en la UCI y la muestra aún sin entregar es un callejón sin salida.

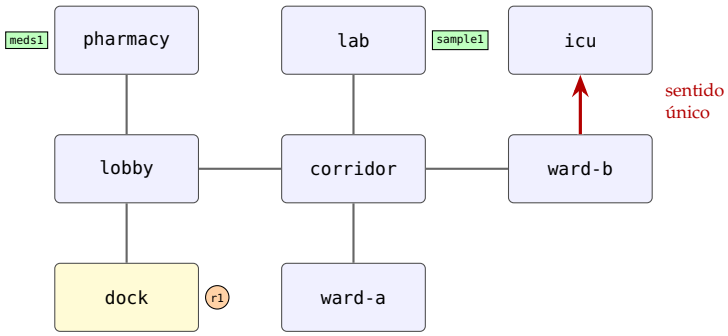


Figura 3.7.: La planta de `courier-p09`: la puerta de la UCI solo se abre desde fuera. `meds1` debe ir a la UCI y `sample1` a la sala A. Entrar en la UCI antes de entregar la muestra es un callejón sin salida.

```
$ kairos courier-domain.pddl courier-p09.pddl --threads 1 --no-
     anytime
; cost = 16
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
```

```
(move r1 lobby corridor)
(drop r1 meds1 corridor)
(move r1 corridor lab)
(pick r1 sample1 lab)
(move r1 lab corridor)
(move r1 corridor ward-a)
(drop r1 sample1 ward-a)
(move r1 ward-a corridor)
(pick r1 meds1 corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
```

El plan deja la UCI para el final, como debe: deja los medicamentos en el pasillo, se ocupa de la muestra y vuelve a por ellos. Aquí al planificador le ayudó su heurística. Desde un estado con el robot dentro de la UCI y la muestra sin entregar, ni siquiera el problema relajado tiene solución (ningún movimiento empieza en la UCI, con borrados o sin ellos), así que el estado se reconoce de inmediato como callejón sin salida y se desecha.

No todos los callejones sin salida se reconocen. En el `courier-p08` de la sección anterior todos los estados eran callejones sin salida, ya que el problema no tiene plan, pero la relajación solo reconoció el de dentro de la UCI: desde los otros siete veía un plan, y la búsqueda tuvo que agotar el espacio para saber la verdad. Un modelo con muchos callejones sin salida no reconocidos es difícil para una búsqueda voraz, que se mete en ellos uno tras otro. Cuando un modelo es lento y su mundo tiene acciones irreversibles (puertas que se cierran con llave, combustible que se gasta, cosas que se rompen), este es el primer sospechoso.

Las acciones irreversibles crean callejones sin salida. Unos los ve la heurística y no cuestan nada; los demás solo se descubren buscando en todo lo que hay detrás de ellos.

Simetría

Dos objetos son **simétricos** cuando intercambiarlos en todas partes convierte el problema en sí mismo. En `courier-p10.pddl` cuatro cajas de medicamentos, de `meds1` a `meds4`, empiezan todas en la farmacia y deben ir todas a la sala A. Nada las distingue, y para el hospital «el robot ha entregado `meds1` y `meds3`» y «el robot ha entregado `meds2` y `meds4`» son la misma situación. Para el planificador son estados distintos, porque son conjuntos de hechos distintos. La Tabla 3.5 muestra el precio a medida que aumentan las cajas.

Problema	Primer plan	Mejor plan	Estados (demostr.)
1 caja (`courier-p02`)	7	7	33
2 cajas	15	15	464
3 cajas	25	23	4670
4 cajas (`courier-p10`)	33	31	43 595
1 caja, 3 robots (`courier-p11`)	7	7	607

Tabla 3.5.: Objetos idénticos, con Kairos y un hilo: acciones del primer plan y del mejor, y estados encontrados para demostrar que el mejor lo es (los problemas con dos y tres cajas están en la carpeta `solutions`, para el Ejercicio 3.8.2). Cada caja idéntica multiplica los estados por unos diez, aunque la única pregunta real es cuántas cajas han llegado. Tres robots idénticos en lugar de uno los multiplican por dieciocho.

El primer plan apenas se ve afectado: la heurística elige una caja y sigue adelante. El daño lo sufre todo lo que explora ampliamente: la búsqueda de planes mejores, la demostración del mejor y, sobre todo, la demostración de que un problema no tiene plan. En STRIPS, el modelador puede hacer poco con objetos idénticos que de verdad hacen falta. El Capítulo 7 muestra cómo contar sin nombrar («tres cajas en la farmacia»),

y el Capítulo 8 lo hace con números. Lo que el modelador ya puede hacer es no crear simetría sin necesidad: nada de robots de repuesto «por si acaso», nada de objetos que el objetivo no exige.

3.9. El método MODEL: tamaño y guía en cifras

El Capítulo 2 aplicó el método MODEL por primera vez. El paso E, Ensayar el modelo, preguntaba si los planes eran correctos; el paso L, Lectura crítica, comparaba codificaciones según cinco criterios, dos de los cuales, Tamaño y Guía, solo podían argumentarse con palabras. Este capítulo les da cifras, y añade una pregunta a cada paso.

- **Paso E, una pregunta nueva: ¿cómo de grande es el modelo, y es lo que esperaba?** Antes de ejecutar, estima las acciones instanciadas (para cada acción, multiplica los objetos que caben en cada parámetro y quédate con lo que permiten las precondiciones estáticas) y los estados alcanzables (multiplica las posibilidades de cada cosa que puede cambiar). Después de ejecutar, lee la línea `[ground]` de `kairos -v`. Una cifra muy por encima de la estimación es un error del modelo mientras no se demuestre lo contrario.
- **Paso L, una pregunta nueva: ¿qué dicen las cifras?** El Tamaño son los átomos y las acciones instanciadas de la línea `[ground]`, y cómo crecen cuando crece el problema. La Guía son los estados expandidos hasta el primer plan, la longitud de ese plan frente a la del mejor, y los estados encontrados cuando el planificador tiene que mirar por todas partes.

El paso E en el mundo del mensajero: estimar y después ejecutar

Vamos a ejecutar `courier-p05`: un robot, cuatro artículos, ocho habitaciones, 14 hechos `connected`.
Estimación. Movimientos: $1 \times 14 = 14$. Recogidas: 1 robot, 4 artículos, 8 habitaciones: 32. Descargas: 32. Acciones instanciadas: 78. Átomos: $8 + 4 \times 8 + 4 + 1 = 45$. Estados: $8 \cdot (8^4 + 4 \cdot 8^3) = 49\,152$. Un espacio de este tamaño se puede agotar en mucho menos de un segundo, así que podemos pedir incluso la demostración del mejor plan.
Ejecución. `kairos courier-domain.pddl courier-p05.pddl -v --threads 1` informa de 45 átomos y 78 acciones, un primer plan de 28 acciones tras 79 expansiones, y `proven optimal` tras encontrar 48 048 estados.
Comparación. Todo cuadra. Si la línea `[ground]` hubiera dicho 326 acciones, como en la Sección 3.8, habríamos buscado un parámetro sobrante antes de mirar ningún plan.

Alternativas comparadas

Tres codificaciones del mundo del mensajero, todas en STRIPS, ejecutadas con las mismas tres entregas (`meds1` a la sala A, `sample1` al laboratorio, `meal1` a la UCI).

Codificación A, la base (`courier-domain.pddl`): `move`, `pick`, `drop`. Un artículo que se lleva encima no está en ninguna habitación.

Codificación B, movimientos cargados (`courier-loaded-domain.pddl`, Listado 3.6). Muchos primeros modelos razonan así: un artículo en las manos del robot sigue estando en algún sitio, así que `item-at` debería

seguir siendo verdadero e ir con el robot. Entonces moverse con un artículo cambia el lugar del artículo, y necesita el artículo como parámetro.

```
(:action move-empty
  :parameters (?r ?from ?to)
  :precondition (and (robot ?r) (location ?from) (location ?to)
                     (robot-at ?r ?from) (connected ?from ?to)
                     (free ?r))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))

(:action move-with
  :parameters (?r ?i ?from ?to)
  :precondition (and (robot ?r) (item ?i)
                     (location ?from) (location ?to)
                     (robot-at ?r ?from) (connected ?from ?to)
                     (carrying ?r ?i))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (item-at ?i ?from)) (item-at ?i ?to)))
```

Listado 3.6: Codificación B: los movimientos de `courier-loaded-domain.pddl`.

Codificación C, ir a cualquier parte (`courier-goto-domain.pddl`). Al hospital no le importa la ruta del robot, así que una sola acción, `go`, lleva al robot a cualquier habitación a la que pueda llegar, y el problema enumera en un predicado estático, `reachable`, los 56 pares de habitaciones distintas.

```
(:action go
  :parameters (?r ?from ?to)
  :precondition (and (robot ?r) (location ?from) (location ?to)
                     (robot-at ?r ?from) (reachable ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

Ejecutar B depara una sorpresa que ninguna tabla de tamaños habría mostrado:

```
$ kairos courier-loaded-domain.pddl courier-loaded-p01.pddl --
    threads 1 --no-anytime
; cost = 18
...
(move-empty r1 icu ward-b)
(pick r1 sample1 ward-b)
(move-with r1 sample1 ward-b corridor)
(move-with r1 sample1 corridor lab)
```

El plan tiene 18 acciones donde la codificación A necesita 19, y termina sin ninguna descarga. En la codificación B, `(item-at sample1 lab)` se hace verdadero cuando el robot *entra* en el laboratorio con la muestra en las manos, y el objetivo no pide nada más. El robot se queda en el laboratorio sujetando la muestra, y el modelo llama a eso una entrega. El objetivo tiene que decir más (Ejercicio 3.9.2).

En C el primer plan da vueltas (empieza con `(go r1 dock icu)`, sin motivo), y el segundo, el mejor, tiene 12 acciones: cuatro por entrega.

	A: base	B: movimientos cargados	C: ir a cualquier parte
Fidelidad	los planes se pueden llevar a cabo tal cual	un artículo llevado cuenta como entregado: el objetivo debe pedir también manos libres	correcta solo si vale cualquier ruta; la ruta no está en el plan
Tamaño	36 átomos, 62 acciones; por robot, 14 movimientos	36 átomos, 104 acciones; 14 movimientos más por artículo	36 átomos, 104 acciones; 56 movimientos por robot, que crecen con el cuadrado de las habitaciones
Guía	primer plan: 19 acciones, 47 expandidos; demostración: 5054 estados	primer plan: 18 acciones, 34 expandidos; demostración: 4567 estados	primer plan: 15 acciones, 26 expandidos; mejor: 12; demostración: 4937 estados
Claridad	tres acciones, un solo significado para `item-at`	cuatro acciones; `item-at` significa dos cosas	tres acciones, pero el problema debe enumerar lo que el mapa implica
Portabilidad	STRIPS puro	STRIPS puro	STRIPS puro

Tabla 3.6.: Tres codificaciones del mundo del mensajero con las mismas tres entregas (problemas `courier-p04`, `courier-loaded-p01` y `courier-goto-p01`; Kairos, un hilo).

Veredicto. B es la que hay que evitar: es más grande que A, no es más fácil de buscar, cuesta más de leer, y su hecho de más dejó pasar un plan erróneo. Su lección es general: un parámetro añadido para mantener al día un hecho redundante se paga en acciones instanciadas y en ocasiones de equivocarse. C es una alternativa real. Acorta los planes y hace menos profunda la búsqueda, y es la elección correcta cuando otro programa, un sistema de navegación por ejemplo, va a encontrar la ruta de todos modos. Es la elección equivocada en cuanto la ruta importa para el plan: las puertas del Capítulo 5 y la batería del Capítulo 8 tratan de lo que ocurre por el camino. Además crece con el cuadrado de las habitaciones, así que en un mapa grande es A la que sigue siendo pequeña. Nos quedamos con A para el resto del libro.

3.10. Errores comunes

Leer «timeout» como «no existe plan»

Un script ejecuta el planificador, no obtiene ningún plan e informa de que la misión es imposible. Pero la línea de estado decía

```
kairos: timeout (timeout)
```

con código de salida 11: la búsqueda se detuvo, y no se demostró nada. Solo `unsolvable`, código de salida 10, significa que no existe ningún plan. Comprueba el código de salida, no la ausencia de plan, y da más tiempo (`--timeout`) antes de culpar al modelo.

Tomar el primer plan por el mejor

Con `--no-anytime`, o cuando solo se lee el primer plan de la salida, el tablero difícil del puzle deslizante recibe un plan de 95 deslizamientos; el mejor tiene 31. El primer plan de una búsqueda voraz es

un plan. Cuando la calidad importa, deja correr la búsqueda anytime y quédate con el *último* plan impreso, y mira la línea de estado: `solved (proven optimal)` es una garantía; `solved (timeout)` es el mejor encontrado en el tiempo dado.

Esperar una demostración que nadie necesita

El error opuesto. En `courier-p06` el mejor plan está en la pantalla a los 10 milisegundos, y el planificador sigue ejecutándose más de tres segundos para demostrar que es el mejor; en problemas más grandes esa demostración no acabaría nunca, y la ejecución se detiene al llegar al límite de tiempo. Si basta con un buen plan, dilo: `--no-anytime` para el primer plan, `--timeout` o `--target-cost` para una mejora acotada.

Un parámetro que nada restringe

El Capítulo 7 da nombre a este error: el *parámetro ocioso*. El plan muestra acciones con un objeto que no pinta nada en ellas:

```
(drop r1 meds1 ward-a dock)
```

y la línea `[ground]` de `kairos -v` informa de 326 acciones donde se esperaban 62. Una acción tiene un parámetro que su precondición y su efecto no usan, y se instancia una vez por cada objeto. `kairos check` acepta el dominio, ya que es PDDL válido. Quita el parámetro. Si el parámetro se usa, pero solo en los efectos, busca la precondición estática que falta y que debería ligarlo a los demás.

Una precondición estática que falta

El plan es más corto de lo que tiene derecho a ser:

```
(move r1 dock pharmacy)
```

aunque la base de carga y la farmacia no están conectados, y el modelo tiene 80 acciones instanciadas donde se esperaban 30. Falta una precondición que relacione dos parámetros, aquí `(connected ?from ?to)`. El modelo es más grande *y* erróneo. Para eso sirve la estimación del paso E: una cuenta que no cuadra se nota antes de leer los planes.

Juzgar un modelo con un problema diminuto

Un modelo que responde en un milisegundo con un artículo se da por terminado. Con cuatro artículos y dos robots, el mismo modelo hace que el planificador encuentre 544 039 estados para demostrar su mejor plan. El tamaño y la guía solo se manifiestan cuando el problema crece. Prueba cada modelo con un problema dos o tres veces mayor que aquel para el que se escribió, y observa cómo crecen las cifras de `-v`.

3.11. Resumen

Un modelo tiene un significado preciso: los estados son conjuntos de hechos, una acción es aplicable cuando sus precondiciones están en el estado, produce un sucesor borrando y añadiendo hechos, y un plan es una secuencia de acciones aplicables que termina en un estado con los

hechos del objetivo. El planificador instancia las acciones, lo que da un número de acciones instanciadas que crece con suavidad con el problema, y después busca en un espacio de estados que crece de forma explosiva. Sobrevive a la explosión gracias a una heurística calculada a partir del propio modelo, casi siempre ignorando los borrados. La heurística no es igual de buena con todos los modelos, y lo que se le pide al planificador (un plan, un buen plan, el mejor plan, una demostración de que no hay ninguno) cambia el coste en órdenes de magnitud.

Idea	Qué recordar
Estado, acción, plan	$s' = (s \setminus del(a)) \cup add(a)$; `kairos validate` comprueba un plan frente a esta definición
Instanciación	los parámetros multiplican, las precondiciones estáticas podan; lee la línea `[ground]` de `kairos -v`
Espacio de estados	implícito en el modelo, nunca escrito; se multiplica con cada objeto nuevo
Heurística	la longitud de un plan que ignora los borrados; ciega a lo que estorba
Anytime	el primer plan, rápido; los mejores, después; la demostración, la última y la más cara
Irresoluble	una demostración (código de salida 10), por la instanciación o agotando la búsqueda; un límite de tiempo (11) no demuestra nada
Para el modelador	ningún parámetro innecesario, todas las precondiciones estáticas, cuidado con los callejones sin salida y los objetos idénticos
MODEL	paso E: estimar el tamaño y después comparar; paso L: Tamaño y Guía en cifras

3.12. Lecturas complementarias

La semántica de STRIPS que se da aquí se remonta al artículo que le dio nombre (Fikes & Nilsson, 1971); Ghallab et al. (2004) la desarrollan formalmente, junto con las clases de búsqueda que este capítulo ha dejado fuera. Geffner y Bonet (2013) es la exposición seria más breve de la planificación como búsqueda heurística, y sus capítulos sobre planificación clásica cubren la relajación de borrados, los hitos y los algoritmos de búsqueda de este capítulo, con sus demostraciones. Para la búsqueda en general (en anchura, voraz primero el mejor, A*), véase Russell y Norvig (2020).

Las fuentes originales se leen bien. Bonet y Geffner (2001) introducen las heurísticas basadas en la relajación de borrados; Hoffmann y Nebel (2001) describen FF, con el plan relajado y las acciones que sugiere; Helmert (2006) y Helmert (2009) explican la instanciación y la traducción a variables con varios valores; Richter y Westphal (2010) añaden los hitos y el esquema *anytime* que sigue Kairos. Lipovetzky y Geffner (2012) proponen otra fuente de guía, la novedad de un estado, que Kairos también usa. El resultado de complejidad está en Bylander (1994). La planificación sin instanciación, para problemas cuyas acciones instanciadas no caben en memoria, la describen Corrêa et al. (2020). Lo que hace el propio Kairos está en Martín Rico et al. (2026) y en el Apéndice A.

3.13. Ejercicios

Los ficheros que se citan en los ejercicios están en `examples/ch03`. Ejecuta Kairos con `--threads 1` para obtener las cifras de las soluciones.

Ejercicios de la Sección 3.1 — Qué significa un modelo

Ejercicio 3.1.1. En el problema diminuto, considera el estado en el que el robot está en el vestíbulo y lleva los medicamentos (s_4 en la Figura 3.2). Escríbelo como un conjunto de hechos, sin los estáticos. Enumera las acciones instanciadas aplicables en él y el estado sucesor de cada una.

Ejercicio 3.1.2. Escribe a mano un plan para `courier-p01` con exactamente siete acciones, y compruébalo con `kairos validate`. ¿Un plan más largo de lo necesario sigue siendo un plan?

Ejercicio 3.1.3. Sin ejecutar nada, di qué responderá `kairos validate` para esta secuencia en `courier-p01`, y en qué paso: `(move r1 lobby pharmacy)`, `(drop r1 meds1 pharmacy)`, `(pick r1 meds1 pharmacy)`. Después compruébalo.

Ejercicios de la Sección 3.2 — Instanciación: de variables a objetos

Ejercicio 3.2.1. Cuenta a mano los átomos y las acciones instanciadas de `courier-p03` (un robot, dos artículos, ocho habitaciones). Compruébalo con `kairos -v`.

Ejercicio 3.2.2. Da una fórmula para el número de acciones instanciadas que indica Kairos para las torres de Hanói con n discos, y compruébala para 4 y 6 discos (`hanoi-p03` y `hanoi-p04`). ¿Cuántas de las acciones instanciadas para 2 discos no pueden aplicarse nunca?

Ejercicio 3.2.3. El dominio *gripper*: un robot con dos pinzas, `left` y `right`, lleva bolas entre dos habitaciones; se mueve entre habitaciones, coge una bola con una pinza libre y la suelta. Escríbelo en STRIPS sin tipos, con un problema que lleve cuatro bolas de `rooma` a `roomb`. Estima sus acciones instanciadas antes de ejecutarlo y después compara.

Ejercicios de la Sección 3.3 — Por qué explota el espacio

Ejercicio 3.3.1. Deduce el número de estados alcanzables del mundo del mensajero con dos robots y cuatro artículos en las ocho habitaciones, la última fila de la Tabla 3.1. ¿Cuántos habría con tres robots?

Ejercicio 3.3.2. El 15-puzle es el puzle deslizante en un tablero de 4×4. ¿Cuántas acciones instanciadas y cuántos átomos le da nuestra codificación? ¿Cuántos estados alcanzables, sabiendo que, como en el tablero pequeño, se puede alcanzar la mitad de las disposiciones? A 52 bytes por estado, la cifra que dio Kairos para el 8-puzle, ¿cuánta memoria haría falta para guardarlos todos?

Ejercicios de la Sección 3.4 — Búsqueda hacia delante

Ejercicio 3.4.1. Dibuja el espacio de estados de las torres de Hanói con dos discos (`hanoi-p01`): tiene 9 estados. Haz a mano una búsqueda en anchura sobre él, desde la torre en `peg1` hasta la torre en `peg3`.

¿En qué orden se expanden los estados, y cuántos se expanden antes de encontrar un estado objetivo?

Ejercicios de la Sección 3.5 — Heurísticas: plantear un problema más sencillo

Ejercicio 3.5.1. En el problema diminuto, construye a mano el plan relajado desde el estado en el que el robot está en el pasillo con las manos vacías y los medicamentos están en la farmacia (s_2). Compara su longitud con la distancia real, y explica la diferencia.

Ejercicio 3.5.2. Explica por qué el plan relajado de las torres de Hanói con n discos tiene n acciones. Comprueba el valor para 6 discos con `--debug=search`.

Ejercicio 3.5.3. En el puzle deslizante, con los borrados ignorados, una casilla que queda vacía sigue vacía. Describe con palabras cómo es un plan relajado, y di si esperas que la heurística se quede corta o se pase en la estimación. Para `puzzle-p02`, compara el valor inicial que imprime `--debug=search` con la longitud del mejor plan.

Ejercicios de la Sección 3.6 — Un plan, un plan mejor, el mejor plan

Ejercicio 3.6.1. Ejecuta `hanoi-p05` (8 discos) tres veces: con `--no-anytime`, con `--target-cost 300` y sin ninguna opción. Para cada ejecución, da la longitud del plan devuelto y el motivo por el que terminó la búsqueda.

Ejercicio 3.6.2. Escribe el problema del puzle deslizante para el tablero «1 3 6 / 5 · 2 / 4 7 8» (el punto es la casilla vacía) y el objetivo habitual. ¿Cuánto mide el primer plan, cuánto el mejor, y cuántos estados encontró el planificador para demostrarlo?

Ejercicios de la Sección 3.7 — No hay plan, o no se ha encontrado

Ejercicio 3.7.1. Copia `courier-p02` y quita `(free r1)` de su estado inicial. Predice qué responderá Kairos, si tendrá que buscar y qué dirá su nota. Después ejecútalo.

Ejercicio 3.7.2. Un compañero dice: «el planificador ha estado funcionando los 30 segundos y no ha impreso ningún plan, así que el objetivo es imposible; se lo diré a las enfermeras». ¿Qué dos cosas habría que comprobar primero, y qué probarías después?

Ejercicios de la Sección 3.8 — Qué significa esto para el modelador

Ejercicio 3.8.1. Un estudiante escribe una acción `handover` para el mundo del mensajero, con la que un robot da el artículo que lleva a otro robot en la misma habitación: parámetros `?from ?to ?i ?l`. Con 2 robots, 4 artículos y 8 habitaciones, ¿cuántas instancias tiene como máximo? ¿Cuáles de ellas son inútiles, y puede STRIPS, tal como lo conocemos hasta ahora, excluirlas?

Ejercicio 3.8.2. Con `courier-p02`, `solutions/courier-boxes2.pddl`, `solutions/courier-boxes3.pddl` y `courier-p10`, reproduce la Tabla 3.5. Después compara la fila de tres cajas con `courier-p04`,

que tiene tres entregas *distintas*. ¿Cuál necesita más estados para la demostración, y qué concluyes?

Ejercicio 3.8.3. En el mundo de la puerta de sentido único, el robot empieza *dentro* de la UCI y debe llevar `meds1` de la farmacia a la UCI. Predice la respuesta de Kairos y cuántas acciones instanciadas indicará. Después ejecuta `solutions/courier-trapped.pddl`.

Ejercicios de la Sección 3.9 — El método MODEL: tamaño y guía en cifras

Ejercicio 3.9.1. Aplica la pregunta nueva del paso E a la codificación C con dos robots y las cuatro entregas de `courier-p06`: estima átomos y acciones instanciadas; después ejecuta el problema `solutions/courier-goto-p02.pddl` y compara. Compara también el mejor plan y el tiempo de la demostración con los de la codificación A en `courier-p06`.

Ejercicio 3.9.2. Corrige el uso de la codificación B para que una entrega no esté completa mientras el robot aún sujeta el artículo, cambiando solo el problema. Ejecútalo y vuelve a rellenar la columna B de la Tabla 3.6. ¿Cambia el veredicto?

Ejercicio 3.9.3. Alguien propone una cuarta codificación, D: mantener `move` como en A y añadir al dominio una acción `deliver ?r ?i ?from ?to` que recoge el artículo en `?from`, lo lleva y lo deja en `?to` en un solo paso, cuando `?to` es alcanzable desde `?from`. Sin escribirla, juzga D según los cinco criterios frente a A y C.

Parte II. Planificación clásica: PDDL 1.2

4. Tipos, constantes e igualdad

El modelo del mensajero del Capítulo 2 funciona, pero se fía demasiado de quien lo escribe. Cada objeto de un problema no es más que un nombre, y lo único que dice que `r1` es un robot y `pharmacy` una habitación es un hecho del estado inicial, `(robot r1)` o `(location pharmacy)`, que las acciones comprueban en sus precondiciones. Supongamos que escribimos `(item-at pharmacy meds1)` en lugar de `(item-at meds1 pharmacy)`. Nadie se queja. El planificador busca un plan, no lo encuentra y dice que el problema no tiene solución, y nos toca leer el estado inicial entero para dar con dos palabras intercambiadas. El Capítulo 3 mostró un segundo coste: una acción con tres parámetros se instancia con todas las combinaciones de tres objetos, y la mayoría de esas combinaciones (una habitación que lleva un robot de un artículo a otro artículo) son disparates que el planificador tiene que desechar.

Este capítulo da al modelo un vocabulario para las *clases* de cosas. Con los **tipos**, cada objeto y cada parámetro declara lo que es, un comprobador rechaza los hechos y las acciones que mezclan clases y el planificador solo prueba las combinaciones que tienen sentido. Los tipos pueden organizarse en una jerarquía (un medicamento es un artículo; un robot es un portador), de modo que una acción sirva para toda una familia de objetos y otra solo para una parte de ella. Con las **constantes**, un dominio puede nombrar un objeto que existe en todos los problemas, como la farmacia. Con la **igualdad**, una acción puede exigir que dos de sus parámetros sean objetos distintos, para que un robot no sea su propio testigo.

Primero medimos lo que cuestan los predicados de tipo del Capítulo 2; después ponemos tipos al mundo del mensajero y vemos qué errores detecta ahora Kairos. Siguen las jerarquías, con un equipo hospitalario de robots y celadores y con el dominio logistics, y después `either`, una construcción para un parámetro que admite varios tipos, junto con su alternativa. Las constantes y la igualdad cierran la parte de lenguaje. La sección del método añade después sus preguntas a los pasos O y D del método MODEL y compara, con cifras, predicados de tipo frente a tipos y una constante frente a un parámetro.

4.1 Lo que cuestan los predicados de tipo 77
4.2 Tipos 78
Lo que ahora detecta el comprobador 80
Lo que gana el planificador 81
4.3 Jerarquías de tipos 81
Un segundo dominio: logistics 84
4.4 Un parámetro de varios tipos: `either` 86
La alternativa: dar nombre a la unión 87
4.5 Constantes 88
4.6 Igualdad 89
Cuándo hace falta la desigualdad 91
4.7 El método MODEL: clases de cosas 92
Alternativas comparadas . 92
4.8 Errores comunes 94
4.9 Resumen 96
4.10 Lecturas complementarias 97
4.11 Ejercicios 97

4.1. Lo que cuestan los predicados de tipo

En el Capítulo 2, las clases de cosas eran predicados corrientes. Los llamamos **predicados de tipo**: `(robot ?x)`, `(location ?x)` y `(item ?x)`. El estado inicial enumeraba uno de esos hechos por objeto, y cada acción empezaba su precondición con uno de ellos por parámetro:

```
(:action pick
  :parameters (?r ?i ?l)
  :precondition (and (robot ?r) (item ?i) (location ?l)
                     (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
  ...
```

Es correcto y funciona en cualquier planificador. Tiene tres costes.

Es ruido. Un tercio de cada precondición y diez hechos del estado inicial de `courier-p01` no dicen nada de la situación; solo repiten lo que los nombres ya le sugieren a un lector humano.

No comprueba nada. Un predicado como `item-at` acepta dos objetos cualesquiera. El fichero `courier-untyped-p01-swapped.pddl` es el primer problema del Capítulo 2 con un despiste, el de la introducción: su estado inicial dice `(item-at pharmacy meds1)`. Pedimos a Kairos que lo compruebe y después que lo resuelva:

```
$ kairos check ../ch02/courier-domain.pddl courier-untyped-p01-
    swapped.pddl
valid, plannable
$ kairos ../ch02/courier-domain.pddl courier-untyped-p01-swapped.
    pddl
note: the goal needs atoms that no action can reach
(item-at meds1 ward-a) can never be reached:
  drop adds it, but needs (carrying ?r - object meds1), which never
     holds
    pick adds it, but needs (item-at meds1 ?l - object), which never
      holds
      drop adds it, but needs (carrying ?r - object meds1), which
     never holds [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

La comprobación pasa: para el lenguaje, ese hecho vale tanto como cualquier otro. Después, el planificador demuestra, con razón, que el problema *tal como está escrito* no tiene plan. La nota ayuda (nadie puede coger `meds1` porque no está en ninguna parte), pero el modelo no le dio al planificador ningún medio para decir qué falla de verdad: que se ha puesto una habitación donde corresponde un artículo.

Multiplica las candidatas. `courier-p01` tiene 10 objetos. Cada una de las tres acciones tiene tres parámetros, así que hay $10^3 = 1000$ formas de instanciar cada una, 3000 en total. Solo 30 de ellas pueden aplicarse alguna vez: 14 movimientos por las 7 conexiones en ambos sentidos, 8 acciones de coger y 8 de dejar. Un buen instanciador encuentra esas 30 sin construir las otras 2970, porque advierte que `(robot pharmacy)` nunca llega a ser cierto; el Capítulo 3 mostró cómo. Pero el trabajo de descartar es real, crece con el cubo del número de objetos y se repite en cada problema.

Los predicados de tipo son correctos, pero añaden ruido, dejan pasar hechos erróneos y dejan que el planificador descubra por su cuenta qué combinaciones de objetos tienen sentido.

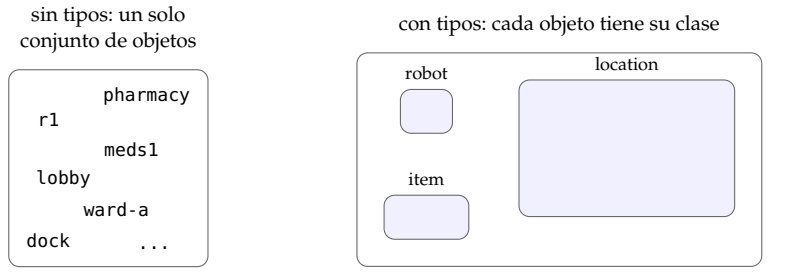


Figura 4.1.: Los objetos de `courier-p01` sin tipos y con tipos. A la izquierda, cualquier objeto puede ocupar cualquier parámetro. A la derecha, cada parámetro solo toma objetos de la caja de su tipo.

4.2. Tipos

Un **tipo** es un nombre para un conjunto de objetos. Un modelo con tipos dice tres cosas: qué tipos existen, qué tipo tiene cada objeto y qué tipo

admite cada parámetro de un predicado o de una acción. Un objeto solo puede ocupar un parámetro si su tipo encaja. El requisito es `:typing`.

El Listado 4.1 es el dominio del mensajero con tipos. Es la versión de la que parte el resto del libro.

```
;; The courier world with types: the base of the later chapters.
(define (domain courier)
  (:requirements :strips :typing)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot))

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to)))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
    :effect (and (not (item-at ?i ?l)) (not (free ?r))
                 (carrying ?r ?i)))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l) (free ?r))))
```

Listado 4.1: El dominio del mensajero con tipos (`examples/ch04/courier-domain.pddl`).

Han cambiado cuatro cosas respecto al Capítulo 2.

- **El requisito.** La línea 3 añade `:typing` a `:strips`.
- **La sección `:types`.** La línea 5 declara los tres tipos. Va después de los requisitos y antes de los predicados.
- **Variables con tipo.** Una variable va seguida de un guion y de su tipo: `?r - robot`. El guion lleva un espacio a cada lado, y eso lo distingue del guion que hay dentro de un nombre como `robot-at`. Un tipo se aplica a todas las variables escritas antes que él que aún no tienen tipo: en `(connected ?from ?to - location)`, tanto `?from` como `?to` son lugares.
- **Sin predicados de tipo.** Los predicados `robot`, `location` e `item` han desaparecido, de las declaraciones y de todas las precondiciones. Lo que decían está ahora en las listas de parámetros.

El problema cambia de la misma manera: `:objects` da a cada objeto su tipo, con la misma notación del guion, y el estado inicial pierde los diez hechos que solo nombraban clases.

```
;; Take the medicines from the pharmacy to ward A (typed version).
(define (problem courier-p01)
  (:domain courier)
  (:objects
    r1 - robot
    lobby corridor pharmacy lab ward-a ward-b icu dock - location
    meds1 - item)
  (:init
    (connected lobby corridor) (connected corridor lobby)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (connected lobby dock) (connected dock lobby)
```

```
    (connected corridor lab) (connected lab corridor)
    (connected corridor ward-a) (connected ward-a corridor)
    (connected corridor ward-b) (connected ward-b corridor)
    (connected ward-b icu) (connected icu ward-b)
    (robot-at r1 dock) (free r1)
    (item-at meds1 pharmacy))
  (:goal (item-at meds1 ward-a)))
```

```
$ kairos courier-domain.pddl courier-p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Listado 4.2: El primer problema del mensajero, con tipos (`examples/ch04/courier-p01.pddl`).

El plan es el del Capítulo 2, paso por paso: los tipos no cambian lo que el robot puede hacer. Cambian lo que el *modelador* puede escribir.

Lo que ahora detecta el comprobador

Cometemos ahora el mismo despiste en el problema con tipos y lo comprobamos:

```
$ kairos check courier-domain.pddl courier-p01-swapped.pddl
courier-p01-swapped.pddl:17:14: error: in the initial state:
     argument 1 of 'item-at' must be item, 'pharmacy' is location [
     semantic.type_mismatch]
 17 |     (item-at pharmacy meds1))
    |              ^
courier-p01-swapped.pddl:17:23: error: in the initial state:
     argument 2 of 'item-at' must be location, 'meds1' is item [
     semantic.type_mismatch]
 17 |     (item-at pharmacy meds1))
    |                       ^
courier-p01-swapped.pddl:17:5: note: arguments 1 and 2 of 'item-at'
     seem to be swapped [semantic.swapped_arguments]
 17 |     (item-at pharmacy meds1))
    |     ^
invalid
```

El mensaje da el fichero, la línea y la columna, dice qué se esperaba y qué se ha encontrado, y añade una conjetura que, en este caso, acierta. El error que costaba una búsqueda y la lectura del fichero entero cuesta ahora un segundo. Un objeto de la clase equivocada, como en `courier-p01-wrongtype.pddl`, cuyo estado inicial dice `(robot-at meds1 dock)`, se señala de la misma forma:

```
$ kairos check courier-domain.pddl courier-p01-wrongtype.pddl
courier-p01-wrongtype.pddl:16:15: error: in the initial state:
     argument 1 of 'robot-at' must be robot, 'meds1' is item [
     semantic.type_mismatch]
 16 |     (robot-at meds1 dock) (free r1)
    |               ^
invalid
```

La misma comprobación se aplica dentro del dominio. Cada uso de un predicado en una precondición o en un efecto se compara con su

declaración, así que una acción que escriba `(carrying ?i ?r)`, con el artículo primero y el robot después, no pasa.

Con tipos, un hecho o una condición que mezcla clases es un error que encuentra el comprobador, con su línea, antes de cualquier búsqueda.

Lo que gana el planificador

Con tipos, las candidatas de una acción se cuentan multiplicando los tamaños de los tipos de sus parámetros, no el número de objetos. En `courier-p01` hay 1 robot, 1 artículo y 8 lugares:

Acción	Sin tipos	Con tipos
`move`	$10^3 = 1000$	$1 \times 8 \times 8 = 64$
`pick`	$10^3 = 1000$	$1 \times 1 \times 8 = 8$
`drop`	$10^3 = 1000$	$1 \times 1 \times 8 = 8$
Total	3000	80

De las 80 candidatas con tipos sobreviven 30: los 50 movimientos entre habitaciones no conectadas se descartan, como antes, porque `connected` es estático. Kairos da las mismas 30 acciones instanciadas para los dos modelos (ejecútalos con `-v` y compara las líneas `[ground]`). Así que, para un planificador con un instanciador cuidadoso, la ganancia está en el trabajo de instanciación, no en la búsqueda que viene después: el problema instanciado es el mismo. Para un planificador que instancia de forma ingenua, la diferencia es de 3000 a 80 acciones, y se agranda deprisa: con 2 robots, 3 artículos y las mismas 8 habitaciones (`courier-p02`) es de 6591 frente a 224.

Los tipos llegaron con el primer PDDL

PDDL lo definieron en 1998 Drew McDermott y el comité de la primera competición de planificación, celebrada en el congreso AIPS-98 McDermott, 2000; McDermott et al., 1998. Aquella primera versión, conocida hoy como PDDL 1.2, ya tenía `:typing`, `:equality` y constantes de dominio: todo lo de este capítulo es tan antiguo como el lenguaje. Los indicadores de requisitos también estaban desde el principio, para que un planificador pudiera decir qué partes del lenguaje admitía y rechazar el resto.

4.3. Jerarquías de tipos

El hospital contrata celadores para trabajar junto a los robots. Un celador recorre la misma planta y coge y deja los mismos artículos. Y los artículos ya no son todos iguales: hay medicamentos, muestras y comidas, y una comida hay que *servírsela* al paciente, cosa que solo hace una persona.

Con los tipos de la sección anterior necesitaríamos un tipo `porter` y, como un parámetro tiene un solo tipo, una segunda copia de todo: `porter-at`, `move-porter`, `pick-porter`... Las copias se diferenciarían en una palabra. Lo que queremos decir es que los robots y los celadores son dos clases de la misma cosa.

Una **jerarquía de tipos** permite decirlo. Un tipo puede declararse **subtipo** de otro, su **supertipo**, con la notación del guion que se usa para los objetos:

```
(:types
  location
  robot porter - carrier
  medicine sample meal - item)
```

Se lee así: `robot` y `porter` son subtipos de `carrier`; `medicine`, `sample` y `meal` son subtipos de `item`; `location` va por libre. La regla que da utilidad a las jerarquías es breve: *un objeto de un tipo es también un objeto de todos los supertipos de ese tipo*. Un parámetro declarado `?c - carrier` admite robots y celadores; un parámetro declarado `?p - porter` solo admite celadores.

El tipo `object`

Todas las jerarquías tienen la misma raíz, el tipo predefinido `object`. Un tipo declarado sin supertipo, como `location` arriba, es subtipo de `object`. Una variable o un objeto escritos sin tipo son de tipo `object`: un parámetro `?x` sin tipo lo admite todo, y un objeto sin tipo solo encaja en parámetros de tipo `object`. Un modelo sin tipos es, por tanto, un modelo con tipos en el que todo es un `object`; por eso Kairos escribió `?r - object` en la nota de la Sección 4.1.

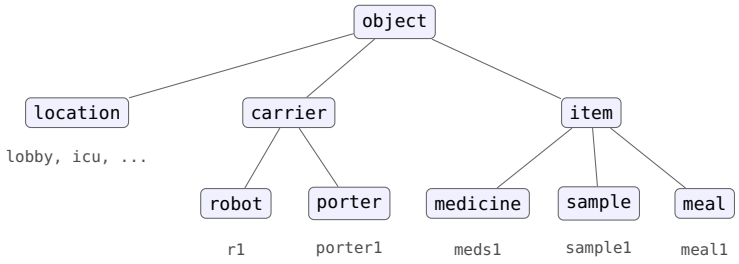


Figura 4.2.: La jerarquía de tipos de `courier-team`, con algunos objetos bajo sus tipos. Un objeto encaja en un parámetro si el tipo del parámetro es el suyo o está por encima de él: `meal1` encaja en `?m - meal`, en `?i - item` y en un `?x` sin tipo.

El Listado 4.3 usa la jerarquía. Los predicados y las tres acciones básicas se declaran sobre los supertipos, así que sirven para cualquier portador y cualquier artículo. Como ya no tratan solo de robots, `robot-at` pasa a llamarse `carrier-at` en este dominio.[1] La nueva acción `serve` se declara sobre dos subtipos: su primer parámetro es un celador; el segundo, una comida.

1: Los capítulos siguientes vuelven a una planta solo con robots y al nombre `robot-at` del Listado 4.1.

```
;; The courier world with a type hierarchy: robots and porters
;; are carriers; medicines, samples and meals are items.
(define (domain courier-team)
  (:requirements :strips :typing)

  (:types
    location
    robot porter - carrier
    medicine sample meal - item)

  (:predicates
    (connected ?from ?to - location)
    (carrier-at ?c - carrier ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?c - carrier ?i - item)
    (free ?c - carrier)
    (served ?m - meal ?l - location))   ; ?m was served in ?l

  ;; Any carrier moves, picks and drops any item.
  (:action move
    :parameters (?c - carrier ?from ?to - location)
    :precondition (and (carrier-at ?c ?from)
                       (connected ?from ?to))
    :effect (and (not (carrier-at ?c ?from))
```

```
                (carrier-at ?c ?to)))

  (:action pick
    :parameters (?c - carrier ?i - item ?l - location)
    :precondition (and (carrier-at ?c ?l) (item-at ?i ?l)
                       (free ?c))
    :effect (and (not (item-at ?i ?l)) (not (free ?c))
                 (carrying ?c ?i)))

  (:action drop
    :parameters (?c - carrier ?i - item ?l - location)
    :precondition (and (carrier-at ?c ?l) (carrying ?c ?i))
    :effect (and (not (carrying ?c ?i))
                 (item-at ?i ?l) (free ?c)))

  ;; Only a porter serves, and only a meal is served.
  (:action serve
    :parameters (?p - porter ?m - meal ?l - location)
    :precondition (and (carrier-at ?p ?l) (carrying ?p ?m))
    :effect (and (not (carrying ?p ?m)) (free ?p)
                 (served ?m ?l))))
```

Listado 4.3: Una jerarquía de portadores y de artículos (`examples/ch04/courier-team-domain.pddl`).

En el problema, cada objeto se declara con su tipo más específico: `r1 - robot`, `porter1 - porter`, `meds1 - medicine`, `meal1 - meal`. El robot y la comida empiezan en el vestíbulo; el celador, en la sala B; los medicamentos, en la farmacia. El objetivo es `(and (served meal1 ward-a) (item-at meds1 icu))`.

```
$ kairos courier-team-domain.pddl courier-team-p01.pddl
...
; better plan 2
; cost = 13
(move r1 lobby pharmacy)
(move porter1 ward-b corridor)
(move porter1 corridor lobby)
(pick porter1 meal1 lobby)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
(move porter1 lobby corridor)
(move porter1 corridor ward-a)
(serve porter1 meal1 ward-a)
```

Leamos el plan. Los mismos `move` y `pick` aparecen con `r1` y con `porter1` como primer argumento: una sola acción, dos clases de portador. El robot lleva los medicamentos a la UCI. El celador va al vestíbulo a por la comida y la sirve en la sala A; el robot está junto a la comida al principio, pero los planes en los que él la lleva a la sala tienen más pasos en total, y servir no es algo que el robot pueda hacer. Para asegurarnos de esto último, escribimos un plan en el que sirve el robot (`courier-team-p01-robot-serves.plan`) y lo validamos:

```
$ kairos validate courier-team-domain.pddl courier-team-p01.pddl
     courier-team-p01-robot-serves.plan
courier-team-p01-robot-serves.plan:5: error: step 4 (serve r1 meal1
     ward-a) at 3.000: argument 1 must be porter, 'r1' is robot [
     validate.type_mismatch]
 5 | (serve r1 meal1 ward-a)
plan invalid
```

Declara un predicado o una acción sobre un supertipo cuando vale para toda la familia, y sobre un subtipo cuando vale para una parte de ella.

El paso O en el equipo: clases antes que propiedades

El paso O del método MODEL pregunta qué cosas existen y qué necesita saber de ellas el planificador. Con tipos, la primera mitad de la pregunta tiene su propio artefacto, una lista de clases que se escribe antes de la tabla de estado:

Clase	Es un	Por qué es una clase propia
`location`		las cosas están *en* él
`carrier`		se mueve, coge y deja
`robot`, `porter`	`carrier`	solo sirve un celador
`item`		se coge y se deja
`meal`	`item`	solo se sirve una comida
`medicine`, `sample`	`item`	los nombran objetivos y reglas futuras

La tercera columna es la prueba de un tipo: *algún predicado o alguna acción trata a estos objetos de forma distinta que al resto*. Una clase que no pasa esa prueba («artículos rojos», «artículos del segundo estante») es una propiedad y va en la tabla de estado como predicado. Lo mismo vale para todo lo que cambia mientras se ejecuta el plan: un objeto nunca cambia de tipo.

Un segundo dominio: logistics

Logistics es uno de los dominios de la competición de 1998 y desde entonces es una prueba clásica. Unos paquetes tienen que viajar entre lugares de varias ciudades. Dentro de una ciudad van en camión; entre ciudades van en avión, y un avión solo aterriza en aeropuertos. La Figura 4.3 muestra el pequeño problema que resolvemos.

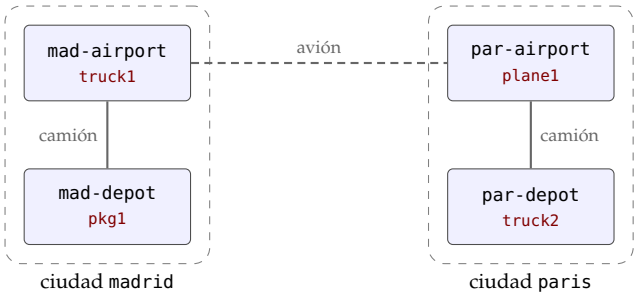


Figura 4.3.: El problema `logistics-p01`: `pkg1` debe ir del almacén de Madrid al almacén de París. Cada lugar muestra, en rojo, lo que hay en él al principio. Las líneas continuas se recorren en camión; la discontinua, en avión.

Las clases, con la prueba de la caja de método: los camiones y los aviones se cargan y se descargan, pero se mueven de forma distinta, así que son dos subtipos de un tipo `vehicle`. Los aeropuertos y los almacenes guardan paquetes y camiones, pero solo los aeropuertos reciben aviones, así que son dos subtipos de `place`. Los paquetes y los vehículos están *en* lugares: comparten un supertipo, `physobj`, para que un único predicado `at` sirva para ambos.

```
;; Logistics with a type hierarchy: packages travel by truck
;; inside a city and by airplane between airports.
(define (domain logistics)
  (:requirements :strips :typing)

  (:types
    city
    airport depot - place
```

```
9     truck airplane - vehicle
10    package vehicle - physobj)
11
12  (:predicates
13    (in-city ?p - place ?c - city)
14    (at ?x - physobj ?p - place)        ; packages and vehicles
15    (in ?k - package ?v - vehicle))
16
17  ;; One load and one unload serve trucks and airplanes.
18  (:action load
19    :parameters (?k - package ?v - vehicle ?p - place)
20    :precondition (and (at ?k ?p) (at ?v ?p))
21    :effect (and (not (at ?k ?p)) (in ?k ?v)))
22
23  (:action unload
24    :parameters (?k - package ?v - vehicle ?p - place)
25    :precondition (and (in ?k ?v) (at ?v ?p))
26    :effect (and (not (in ?k ?v)) (at ?k ?p)))
27
28  ;; A truck drives between any two places of one city.
29  (:action drive
30    :parameters (?t - truck ?from ?to - place ?c - city)
31    :precondition (and (at ?t ?from)
32                       (in-city ?from ?c) (in-city ?to ?c))
33    :effect (and (not (at ?t ?from)) (at ?t ?to)))
34
35  ;; An airplane flies only between airports.
36  (:action fly
37    :parameters (?a - airplane ?from ?to - airport)
38    :precondition (at ?a ?from)
39    :effect (and (not (at ?a ?from)) (at ?a ?to))))
```

Listado 4.4: Logistics con una jerarquía de tres niveles (`examples/ch04/logistics-domain.pddl`).

Una jerarquía puede tener cualquier profundidad: `truck` es un `vehicle`, un `vehicle` es un `physobj` y un `physobj` es un `object`. Las líneas de `:types` pueden ir en cualquier orden, y un tipo puede usarse como supertipo antes de la línea que lo declara (aquí `vehicle`, en las líneas 9 y 10).

Cada parámetro lleva un tipo tan alto en la jerarquía como permite la regla que expresa, y no más. `load` admite cualquier `vehicle` en cualquier `place`; las codificaciones clásicas sin tipos de este dominio necesitan dos acciones, `load-truck` y `load-airplane`, con el mismo cuerpo. `drive` admite un `truck`, y `fly` un `airplane` y dos parámetros de tipo `airport`: la regla «un avión solo aterriza en aeropuertos» queda escrita en la lista de parámetros y no cuesta ninguna precondición.

```
$ kairos logistics-domain.pddl logistics-p01.pddl
; cost = 12
(drive truck1 mad-airport mad-depot madrid)
(load pkg1 truck1 mad-depot)
(drive truck1 mad-depot mad-airport madrid)
(unload pkg1 truck1 mad-airport)
(fly plane1 par-airport mad-airport)
(load pkg1 plane1 mad-airport)
(fly plane1 mad-airport par-airport)
(unload pkg1 plane1 par-airport)
(drive truck2 par-depot par-airport paris)
(load pkg1 truck2 par-airport)
(drive truck2 par-airport par-depot paris)
(unload pkg1 truck2 par-depot)
```

El paquete cambia dos veces de vehículo, en los dos aeropuertos, y la única acción `load` aparece con un camión y con el avión. Un plan más corto llevaría el paquete en avión directamente al almacén de París. El fichero `logistics-p01-fly-to-depot.plan` lo intenta:

```
$ kairos validate logistics-domain.pddl logistics-p01.pddl logistics-
    p01-fly-to-depot.plan
logistics-p01-fly-to-depot.plan:8: error: step 7 (fly plane1 mad-
    airport par-depot) at 6.000: argument 3 must be airport, 'par-
    depot' is depot [validate.type_mismatch]
 8 | (fly plane1 mad-airport par-depot)
plan invalid
```

4.4. Un parámetro de varios tipos: `either`

Una biblioteca pública presta libros y DVD. Las revistas no salen del edificio: un socio coge una de la estantería, la lee y la devuelve a su sitio. Así que `on-shelf` trata de las tres clases, y `borrowed`, de dos de ellas. Si los tipos son `book`, `dvd` y `magazine`, sin jerarquía, ningún tipo encaja por sí solo en el parámetro de `borrow`.

PDDL tiene una construcción para este caso. Donde se espera un tipo, `(either t1 t2 ...)` representa la unión de varios tipos: el parámetro admite un objeto de cualquiera de ellos. Forma parte de `:typing` y no necesita un requisito propio.

```
;; A library: books and DVDs can be borrowed, magazines cannot.
;; The parameter of borrow uses (either ...).
(define (domain library)
  (:requirements :strips :typing)

  (:types book dvd magazine member)

  (:predicates
    (on-shelf ?x - (either book dvd magazine))
    (borrowed ?x - (either book dvd) ?m - member)
    (reading ?g - magazine ?m - member)
    (can-borrow ?m - member))     ; one loan at a time

  (:action borrow
    :parameters (?m - member ?x - (either book dvd))
    :precondition (and (on-shelf ?x) (can-borrow ?m))
    :effect (and (not (on-shelf ?x)) (not (can-borrow ?m))
                 (borrowed ?x ?m)))

  (:action give-back
    :parameters (?m - member ?x - (either book dvd))
    :precondition (borrowed ?x ?m)
    :effect (and (not (borrowed ?x ?m))
                 (on-shelf ?x) (can-borrow ?m)))

  ;; Magazines are read in the library.
  (:action take-to-read
    :parameters (?m - member ?g - magazine)
    :precondition (on-shelf ?g)
    :effect (and (not (on-shelf ?g)) (reading ?g ?m)))

  (:action put-back
    :parameters (?m - member ?g - magazine)
    :precondition (reading ?g ?m)
    :effect (and (not (reading ?g ?m)) (on-shelf ?g))))
```

Listado 4.5: Una biblioteca con `either` (`examples/ch04/library-domain.pddl`).

En el problema `library-p01`, Ana tiene `book1` en préstamo y quiere `dvd1`; Bob quiere `book1` y quiere leer `mag1`. Un socio tiene un solo préstamo a la vez.

```
$ kairos library-domain.pddl library-p01.pddl
; cost = 4
(take-to-read bob mag1)
(give-back ana book1)
(borrow ana dvd1)
(borrow bob book1)
```

Ana tiene que devolver el libro antes de llevarse el DVD (un préstamo a la vez) y antes de que Bob pueda llevarse el libro. Un objetivo que pide una revista prestada, como en `library-p02-wrong.pddl`, lo rechaza el comprobador, que imprime la unión como tipo esperado:

```
$ kairos check library-domain.pddl library-p02-wrong.pddl
library-p02-wrong.pddl:8:20: error: in the goal: argument 1 of '
     borrowed' must be (either book dvd), 'mag1' is magazine [
     semantic.type_mismatch]
 8 |   (:goal (borrowed mag1 ana)))
   |                  ^
invalid
```

La alternativa: dar nombre a la unión

`(either book dvd)` aparece tres veces en el Listado 4.5, y `(either book dvd magazine)`, una. Una unión que el dominio repite una y otra vez es un concepto del dominio que todavía no tiene nombre. Aquí los nombres son fáciles de encontrar: lo que se puede tomar prestado es *loanable* (prestable), y todo lo que hay en las estanterías es *media* (fondos). Una jerarquía lo dice una sola vez:

```
(:types
  member
  loanable magazine - media
  book dvd - loanable)

(:predicates
  (on-shelf ?x - media)
  (borrowed ?x - loanable ?m - member)
  ...
```

El fichero `library-hier-domain.pddl` es el dominio completo escrito así. Resuelve `library-p01` con el mismo plan y rechaza `library-p02-wrong` con «must be loanable» en el mensaje. La Figura 4.4 dibuja ambas versiones.

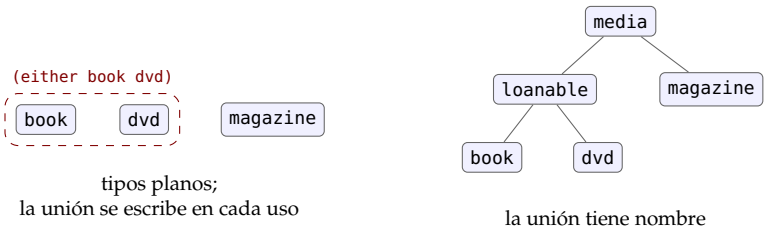


Figura 4.4.: Dos formas de decir «un libro o un DVD». Con `either`, la unión se construye donde hace falta. Con una jerarquía, es un tipo que se declara una sola vez.

¿Cuál elegir? La jerarquía se lee mejor, y una nueva clase de cosa prestable (un audiolibro) es una palabra en `:types`, frente a una modificación en cada `either`. También es más portable: entre los planificadores que compara este libro, Fast Downward (y por tanto LAMA) rechaza `either` en los parámetros de las acciones y ENHSP no lo admite en absoluto,

mientras que todos leen jerarquías. A `either` le queda un uso: las uniones que *atraviesan* una jerarquía que ya es la adecuada para todo lo demás. Un tipo tiene un solo supertipo, así que una jerarquía es un árbol, y dos agrupaciones que se solapan (lo que se puede tomar prestado y lo que necesita un reproductor) no caben las dos en él. Entonces una agrupación se convierte en el árbol y la otra en un `either`, o en un predicado como `(needs-player ?x)`.

Prefiere un supertipo con nombre a `either`. Reserva `either` para una unión que atraviesa la jerarquía, y recuerda que algunos planificadores lo rechazan.

4.5. Constantes

Hasta ahora, un dominio nunca ha mencionado un objeto. Las acciones hablan de `?r` y de `?l`, y solo el problema sabe que hay una `pharmacy`. Eso es lo que hace que un dominio se pueda reutilizar en cualquier planta de cualquier hospital. Pero algunas reglas del mundo tratan de una cosa concreta. En nuestro hospital, los medicamentos no se quedan tirados en una habitación para que un robot los recoja: están en las existencias de la farmacia hasta que el farmacéutico se los entrega a un robot. La regla nombra la farmacia. Una acción que la exprese no puede tener un parámetro de lugar, porque el lugar no es una elección.

Una **constante** es un objeto declarado en el dominio. Existe en todos los problemas de ese dominio, y las acciones pueden usar su nombre allí donde podrían usar una variable. Las constantes se declaran en una sección `:constants`, después de `:types` y antes de `:predicates`, con la notación de `:objects`. No necesitan ningún requisito.

```
;; The courier world with a constant: medicines are dispensed
;; in the pharmacy, a room that every problem has.
(define (domain courier-pharmacy)
  (:requirements :strips :typing)

  (:types
    robot location
    medicine sample meal - item)

  (:constants pharmacy - location)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (in-stock ?m - medicine))     ; ?m is still in the pharmacy
  ;; The pharmacist hands a medicine to a robot. The action
  ;; names the pharmacy: it has no location parameter.
  (:action dispense
    :parameters (?r - robot ?m - medicine)
    :precondition (and (robot-at ?r pharmacy) (in-stock ?m)
                       (free ?r))
    :effect (and (not (in-stock ?m)) (not (free ?r))
                 (carrying ?r ?m))))
```

Listado 4.6: La farmacia como constante (`examples/ch04/courier-pharmacy-domain.pddl`).

(El listado omite `move`, `pick` y `drop`, que son las del Listado 4.1.) La acción `dispense` tiene dos parámetros. Su precondición `(robot-at ?r pharmacy)` mezcla una variable y una constante, algo que se permite en cualquier posición de cualquier predicado, en precondiciones y en efectos. El nuevo predicado `(in-stock ?m - medicine)` dice que un medicamento aún no se ha dispensado; mientras está en existencias no está `item-at` en ninguna habitación, así que `pick` no puede cogerlo.

El problema usa la constante como cualquier otro objeto, en el estado inicial y en el objetivo, pero *no* la declara:

```
;; Medicines from the stock of the pharmacy to ward A and the ICU.
;; pharmacy is a constant of the domain: it is not in :objects.
(define (problem courier-pharmacy-p01)
  (:domain courier-pharmacy)
  (:objects
    r1 - robot
    lobby corridor lab ward-a ward-b icu dock - location
    meds1 meds2 - medicine
    sample1 - sample)
  (:init
    (connected lobby corridor) (connected corridor lobby)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (connected lobby dock) (connected dock lobby)
    (connected corridor lab) (connected lab corridor)
    (connected corridor ward-a) (connected ward-a corridor)
    (connected corridor ward-b) (connected ward-b corridor)
    (connected ward-b icu) (connected icu ward-b)
    (robot-at r1 dock) (free r1)
    (in-stock meds1) (in-stock meds2)
    (item-at sample1 icu))
  (:goal (and (item-at meds1 ward-a) (item-at meds2 icu)
              (item-at sample1 lab))))
```

```
$ kairos courier-pharmacy-domain.pddl courier-pharmacy-p01.pddl
; cost = 21
(move r1 dock lobby)
(move r1 lobby pharmacy)
(dispense r1 meds2)
(move r1 pharmacy lobby)
...
(move r1 lobby pharmacy)
(dispense r1 meds1)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Listado 4.7: Un problema del dominio con una constante: `pharmacy` aparece en `:init` pero no en `:objects` (`examples/ch04/courier-pharmacy-p01.pddl`).

Los pasos `(dispense r1 meds2)` y `(dispense r1 meds1)` no llevan lugar: el plan solo dice lo que se ha elegido. El robot va dos veces a la farmacia, porque lleva un solo artículo cada vez; entre medias (la parte recortada de la salida) deja `meds2` en la UCI y trae de allí `sample1` al laboratorio, con el `pick` de siempre.

Una constante es un objeto que declara el propio dominio porque una de sus reglas lo nombra. Pertenece a todos los problemas, y ningún problema vuelve a declararlo.

Constantes típicas son un lugar destacado (la farmacia, la base de carga, el almacén de una empresa de reparto) y los valores de una escala pequeña y fija que nombran las acciones, como `(:constants low medium high - level)`. La prueba es siempre la misma: *¿alguna regla del dominio menciona este objeto por su nombre?* Si la respuesta es no, el objeto va en los problemas. El vestíbulo también está en todos los problemas de este libro, pero ninguna acción lo trata de forma distinta que al pasillo, así que no es una constante.

4.6. Igualdad

El hospital tiene ahora una regla para los medicamentos: antes de entregar un medicamento, un segundo robot debe verificarlo. El robot que lleva el medicamento y un robot que hace de testigo se encuentran en alguna

habitación, y el medicamento pasa a estar `verified`. Una primera versión de la acción:

```
(:action verify
  :parameters (?r ?w - robot ?m - medicine ?l - location)
  :precondition (and (robot-at ?r ?l) (robot-at ?w ?l)
                     (carrying ?r ?m))
  :effect (verified ?m))
```

Siguiendo el paso E del método, la probamos con un problema que *no* debería tener plan: `courier-check-p01` tiene un solo robot, y su objetivo es `(and (item-at meds1 icu) (verified meds1))`.

```
$ kairos courier-check-noeq-domain.pddl courier-check-p01.pddl
; cost = 9
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(verify r1 r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
```

Hay plan, y su cuarto paso deja ver el agujero: `(verify r1 r1 meds1 pharmacy)`. El robot es su propio testigo. Nada en la acción decía que `?r` y `?w` fueran robots distintos, y el planificador no lo supone: *dos parámetros de una acción siempre pueden ligarse al mismo objeto, si su tipo encaja en ambos*. Que las variables tengan nombres distintos no garantiza nada.

Para exigir dos objetos distintos, PDDL ofrece el predicado `=`, que necesita el requisito `:equality`. `(= ?a ?b)` es cierto cuando las dos variables están ligadas al mismo objeto.[2] No se declara en `:predicates`, nunca está en el estado inicial y ningún efecto puede cambiarlo. Casi siempre se usa negado, como la condición $a \neq b$:

2: En PDDL, dos nombres distintos nunca denotan el mismo objeto (la *hipótesis de nombres únicos*), así que `(= r1 r2)` es falso sea cual sea el estado.

```
;; The courier world with a double check: a medicine is verified
;; by a second robot, different from the one that carries it.
(define (domain courier-check)
  (:requirements :strips :typing :equality
                 :negative-preconditions)
  ;; Robot ?r carries the medicine, robot ?w witnesses.
  (:action verify
    :parameters (?r ?w - robot ?m - medicine ?l - location)
    :precondition (and (robot-at ?r ?l) (robot-at ?w ?l)
                       (carrying ?r ?m)
                       (not (= ?r ?w)))
    :effect (verified ?m)))
```

Listado 4.8: La doble verificación: los requisitos y la acción `verify`, con la desigualdad como última precondición (`examples/ch04/courier-check-domain.pddl`).

El `not` que rodea la igualdad es la primera condición de este libro que no es un simple hecho. Las condiciones con `not` son el tema del Capítulo 5 y tienen su propio requisito, `:negative-preconditions`, que el dominio declara junto a `:equality`. Muchos planificadores aceptan `(not (= ?a ?b))` solo con `:equality`; Kairos también lo acepta, con un aviso que pide el segundo requisito.

Ahora el problema con un solo robot no tiene plan, como debe ser:

```
$ kairos courier-check-domain.pddl courier-check-p01.pddl
note: the goal needs atoms that no action can reach
(verified meds1) can never be reached:
```

```
   verify adds it, but is never applicable (types, negative or
      numeric conditions) [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

Con un segundo robot esperando en el pasillo (`courier-check-p02`, Figura 4.5), el planificador encuentra tres planes, cada uno mejor que el anterior. El mejor:

```
$ kairos courier-check-domain.pddl courier-check-p02.pddl
...
; better plan 3
; cost = 9
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(verify r1 r2 meds1 corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
```

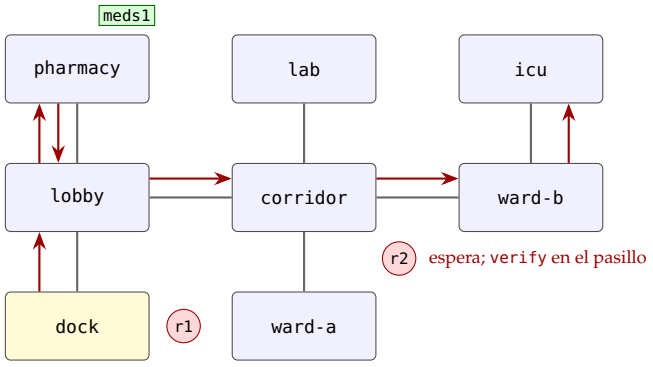


Figura 4.5.: El mejor plan de `courier-check-p02`. La ruta de `r1` pasa por el pasillo, donde espera `r2`, así que la verificación cuesta un paso y ningún rodeo.

El primer plan que imprimió Kairos mandaba al testigo a buscar el medicamento, y el segundo hacía que `r2` fuera hasta la sala B; el tercero aprovecha que `r1` cruza el pasillo de todos modos.

El paso E: la prueba de la trampa para parámetros del mismo tipo

La batería de pruebas del paso E incluye un problema que tienta al planificador a hacer trampa. Para una acción con dos parámetros del mismo tipo, o de tipos en los que uno está por encima del otro, la trampa es siempre la misma: *ligar los dos al mismo objeto*. La receta: escribe un problema con un solo objeto de ese tipo y pide algo que debería necesitar dos. Si sale un plan, lee el paso en el que el mismo nombre aparece dos veces.

Cuándo hace falta la desigualdad

No todo par de parámetros del mismo tipo necesita una desigualdad. En `(move ?r ?from ?to)` nada impide que `?from` y `?to` sean la misma habitación, pero la precondición `(connected ?from ?to)` hace el trabajo: ningún problema declara `(connected lobby lobby)`, así que ese movimiento nunca es aplicable. La desigualdad hace falta cuando *ninguna otra precondición* separa los dos parámetros. El `drive` del Listado 4.4 es uno

de esos casos: sus precondiciones dicen que los dos lugares están en la misma ciudad, cosa que un lugar cumple consigo mismo, así que `(drive truck1 mad-depot mad-depot madrid)` es un paso legal. Es inofensivo (no cambia nada y ningún plan sensato lo contiene), pero es una acción instanciada más por cada camión y cada lugar. Una desigualdad que solo elimina acciones inútiles es buena limpieza; una que elimina planes imposibles, como en `verify`, forma parte del significado del modelo.

Dos parámetros pueden ligarse al mismo objeto. Añade `(not (= ?a ?b))` cuando la acción no tiene sentido con un mismo objeto en los dos papeles y ninguna otra precondición lo impide.

4.7. El método MODEL: clases de cosas

El capítulo cambia poco la ficha del modelo del mundo del mensajero (paso M): las historias han añadido un celador, un farmacéutico y una regla de doble verificación, y cada una ha entrado en la línea Alcance como «dentro». Lo que cambia es cómo se llevan a cabo los pasos O y D. Cada característica trae una pregunta.

Tabla 4.1.: Lo que este capítulo añade al método MODEL.

Paso	Preguntas nuevas
O	**¿Qué clases de cosas hay?** Una clase es un tipo si algún predicado o alguna acción trata a sus objetos de forma distinta que al resto y ningún objeto cambia nunca de clase.
O	**¿Hay una jerarquía?** Busca clases que compartan predicados o acciones (un supertipo común) y acciones que se apliquen a una parte de una clase (un subtipo).
O	**¿Pertenece un objeto a todos los problemas?** Si una regla del dominio lo nombra, es una constante.
D	**¿Pueden dos parámetros ser el mismo objeto?** Para cada par de parámetros cuyos tipos se solapan: si la respuesta es no y ninguna precondición lo impide ya, añade una desigualdad.

Los artefactos cambian en consecuencia. La tabla de estado del paso O va precedida de la lista de clases (como en la caja de método de la Sección 4.3), y su columna «de qué trata» contiene ahora tipos: `carrying` trata de un `carrier` y de un `item`. La tabla de acciones del paso D da un tipo a cada parámetro y gana una línea «distintos» para los pares que deben diferir. La batería de pruebas del paso E gana, por cada uno de esos pares, la prueba de la trampa de la sección anterior, y dos costumbres baratas: ejecutar `kairos check` sobre cada fichero antes de resolver, y escribir al menos un plan erróneo por cada regla de subtipo y ver cómo lo rechaza `kairos validate`.

Alternativas comparadas

Predicados de tipo o tipos. Tenemos los dos modelos: el que no tiene tipos, del Capítulo 2, y el del Listado 4.1. Los ejecutamos sobre el mismo problema con dos robots y tres artículos (`courier-untyped-p02` y `courier-p02`). Los dos devuelven los mismos planes, primero uno de 21 pasos y después el óptimo de 20, en el mismo tiempo.

Criterio	Predicados de tipo	Tipos
Fidelidad	Los mismos planes. Un hecho con los argumentos intercambiados se acepta y da «unsolvable» o, peor, un plan erróneo.	Los mismos planes. Los argumentos intercambiados o de la clase equivocada son errores de `check`.
Tamaño	6591 acciones candidatas; 124 acciones instanciadas tras podar con los hechos estáticos de tipo.	224 candidatas; las mismas 124 acciones instanciadas.
Guía	El mismo problema instanciado, la misma búsqueda.	Lo mismo.
Claridad	13 hechos de más por problema y una condición de más por parámetro. Jerarquías a mano: `(item meds1)` *y* `(medicine meds1)`.	Las clases se leen en las listas de parámetros; una jerarquía es una línea.
Portabilidad	`:strips` a secas: todos los planificadores.	`:typing`: todos los planificadores de la comparativa del Apéndice B que leen dominios clásicos.

Tabla 4.2.: Predicados de tipo frente a tipos, medidos sobre el problema del mensajero con dos robots.

Veredicto. Usa tipos. No cuestan nada en portabilidad ni en búsqueda, y convierten toda una clase de errores silenciosos en mensajes de error. A los predicados de tipo les quedan dos usos. Uno es una clase que *cambia*: una muestra que pasa a estar «analizada» tiene una propiedad, no un tipo nuevo. El otro es una clasificación que atraviesa el árbol, en la que un objeto necesitaría dos supertipos; un predicado estático como `(fragile ?i)` lo dice sin alterar la jerarquía.

Una constante o un parámetro. La regla de la farmacia puede escribirse sin constante. En `courier-dispensary-domain.pddl`, la habitación es un parámetro de la acción, un predicado estático marca las habitaciones donde se dispensan medicamentos y las existencias dicen dónde están:

```
(:action dispense
  :parameters (?r - robot ?m - medicine ?l - location)
  :precondition (and (dispensary ?l) (robot-at ?r ?l)
                     (in-stock ?m ?l) (free ?r))
  :effect (and (not (in-stock ?m ?l)) (not (free ?r))
               (carrying ?r ?m)))
```

El problema declara `pharmacy` entre sus lugares y añade `(dispensary pharmacy)` al estado inicial. Los dos dominios resuelven las mismas entregas con planes de 21 pasos.

Criterio	Constante `pharmacy`	Parámetro y `dispensary`
Fidelidad	Correcta mientras cada hospital tenga exactamente una farmacia. Una segunda no puede modelarse sin modificar el dominio.	Cualquier número de dispensarios, cada uno con sus existencias.
Tamaño	`dispense`: $1 \times 2 = 2$ candidatas. 64 acciones instanciadas en total.	`dispense`: $1 \times 2 \times 8 = 16$ candidatas, 2 tras la poda. 64 acciones instanciadas en total.
Guía	El mismo problema instanciado.	Lo mismo.
Claridad	La regla se lee en el dominio. Los pasos del plan son cortos: `(dispense r1 meds2)`.	El dominio es general; la regla queda repartida entre la acción y un hecho de cada problema. Los pasos repiten lo que ya se sabe: `(dispense r1 meds2 pharmacy)`.
Portabilidad	Constantes: todos los planificadores de la comparativa que leen dominios clásicos. Un problema que vuelva a declarar la constante puede ser rechazado.	STRIPS con tipos, sin más.

Tabla 4.3.: Una constante frente a un parámetro con un predicado estático, para la regla de la farmacia.

Veredicto. Una constante es lo adecuado cuando el objeto forma parte de las *reglas*: hay exactamente uno en cada problema, y sin él el dominio sería otro dominio. Un parámetro con un predicado estático es lo adecuado cuando el objeto forma parte de la *situación*: puede haber uno, varios o ninguno. El mundo del mensajero de los capítulos siguientes toma el segundo camino para la base de carga (un predicado `has-charger`, en el Capítulo 8), porque una planta más grande puede tener dos cargadores. En caso de duda, elige el parámetro: convertir después un predicado en una constante supone borrar cosas, mientras que lo contrario supone modificar todas las acciones y todos los problemas.

4.8. Errores comunes

Un tipo que se traga la variable anterior

Un tipo se aplica a *todas* las variables sin tipo que lo preceden. Borrar «`- robot`» por descuido no deja a `?r` sin tipo; le da el siguiente:

```
(:action pick
  :parameters (?r ?i - item ?l - location)
```

Aquí `?r` es un artículo. Kairos señala cada uso de `?r` que necesita un robot (`courier-dash-domain.pddl`):

```
$ kairos check courier-dash-domain.pddl
courier-dash-domain.pddl:23:34: error: in action 'pick':
     argument 1 of 'robot-at' must be robot, '?r' is item [
     semantic.type_mismatch]
 23 |     :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (
     free ?r))
    |                                  ^
...
invalid
```

Lo mismo ocurre en `:objects`, en el sentido contrario: un nombre escrito *después* del último tipo de la lista no tiene tipo y es un simple `object`. Cuando un mensaje diga que una variable o un objeto «es» de un tipo que no pretendías, mira los guiones de su declaración, no la línea del mensaje.

Tipos sin `:typing`

Un dominio que usa tipos pero solo declara `:strips` no es PDDL estándar. Kairos lo lee y avisa (`courier-noreq-domain.pddl`):

```
$ kairos check courier-noreq-domain.pddl
courier-noreq-domain.pddl:3:18: warning: uses typing but :
     requirements does not include :typing [semantic.
     undeclared_requirement]
 3 |   (:requirements :strips)
   |                  ^
valid, plannable
  typing: full
```

Otros planificadores se detienen con un error de sintaxis en el primer guion, lejos de la causa. El aviso solo aparece con `check` o con `-v`: un motivo más para comprobar antes de resolver. Lo mismo vale para `:equality`.

Un objeto declarado con el supertipo

Un objeto encaja en los parámetros de su tipo y de los tipos que están *por encima*, nunca por debajo. Si el problema declara `meds1 meds2 - item`, los medicamentos son artículos de los que no se sabe que sean medicamentos, y los predicados y las acciones que piden un `medicine` los rechazan (`courier-pharmacy-p01-supertype.pddl`):

```
$ kairos check courier-pharmacy-domain.pddl courier-pharmacy-p01
     -supertype.pddl
courier-pharmacy-p01-supertype.pddl:19:15: error: in the initial
      state: argument 1 of 'in-stock' must be medicine, 'meds1'
     is item [semantic.type_mismatch]
 19 |     (in-stock meds1) (in-stock meds2)
    |               ^
...
invalid
```

Cuando ningún hecho del problema menciona el objeto en una posición de subtipo, no hay mensaje: el objeto simplemente nunca se elige para las acciones del subtipo, y el síntoma es un plan que falta o que es más largo de lo esperado. Declara cada objeto con su tipo más específico.

Una constante declarada otra vez en el problema

Una constante ya existe en todos los problemas. Incluirla en `:objects` declara el mismo nombre dos veces (`courier-pharmacy-p01-redeclared.pddl`):

```
$ kairos check courier-pharmacy-domain.pddl courier-pharmacy-p01
    -redeclared.pddl
courier-pharmacy-p01-redeclared.pddl:7:20: warning: 'pharmacy'
    is already a domain constant [semantic.redeclared_constant]
 7 |     lobby corridor pharmacy lab ward-a ward-b icu dock -
    location
   |                    ^
valid, plannable
  typing: full
```

Kairos se queda con un solo objeto y sigue adelante; un planificador más estricto rechaza el problema. El error es frecuente cuando un dominio gana una constante tarde y sus problemas antiguos no se actualizan.

Confiar en que variables distintas son objetos distintos

El planificador liga `?r` y `?w` al mismo robot siempre que eso le ayuda, como en el `(verify r1 r1 meds1 pharmacy)` de la Sección 4.6. No hay mensaje de error para esto, porque es PDDL legal: el síntoma es un plan más corto de lo que debería, o un plan para un problema que no debería tenerlo. Solo la prueba de la trampa lo encuentra. La solución es `(not (= ?r ?w))` y el requisito `:equality`.

4.9. Resumen

Los tipos dicen de qué clase es cada objeto y qué clase admite cada parámetro. Eliminan los predicados de tipo y sus hechos, permiten que `kairos check` y `kairos validate` rechacen lo que mezcla clases, y reducen las candidatas de cada acción de una potencia del número de objetos a un producto de los tamaños de sus tipos. Una jerarquía permite enunciar una regla una sola vez, en el nivel en el que se cumple. Las constantes dan al dominio un nombre para los objetos que mencionan sus reglas. La igualdad permite que una acción exija que dos parámetros sean distintos, algo que el planificador nunca supone.

Construcción	Requisito	Qué dice
`(:types a b - c)`	`:typing`	`a` y `b` son tipos, subtipos de `c` (de `object` si no se da supertipo)
`?x - t, name - t`	`:typing`	la variable admite, o el objeto es, un `t` o cualquier subtipo suyo
`(either t1 t2)`	`:typing`	en lugar de un tipo: un objeto de cualquiera de ellos
`(:constants n - t)`	ninguno	un objeto de todos los problemas, utilizable en las acciones
`(= ?a ?b)`	`:equality`	los dos son el mismo objeto
`(not (= ?a ?b))`	`:equality`*	son objetos distintos

Tabla 4.4.: Las construcciones de este capítulo. (*) Junto con `:negative-preconditions`, por el `not`.

En cuanto al método: enumera las clases antes que las propiedades; da a cada parámetro un tipo tan alto en la jerarquía como permita su regla, y no más; haz constante solo lo que nombran las reglas; y, para cada par de parámetros cuyos tipos se solapan, pregúntate si pueden ser el mismo objeto.

4.10. Lecturas complementarias

El manual del primer PDDL McDermott et al., 1998 define `:typing`, con `either`, `:constants` y `:equality`, en unas pocas páginas; el artículo sobre la competición de 1998 McDermott, 2000 cuenta cómo se usó allí el lenguaje y describe sus dominios, logistics entre ellos. El libro de texto de Haslum, Lipovetzky, Magazzeni y Muise Haslum et al., 2019 presenta el PDDL con tipos con más detalle formal que este capítulo. Cómo instancia un planificador de forma eficiente una tarea con tipos, y por qué los hechos estáticos de tipo hacen desaparecer la mayoría de las candidatas, se explica en la descripción del traductor de Fast Downward Helmert, 2009. Los planificadores que no instancian en absoluto y trabajan directamente sobre las acciones con tipos sin instanciar son un tema de investigación actual Corrêa et al., 2020.

4.11. Ejercicios

Ejercicios de la Sección 4.1 — Lo que cuestan los predicados de tipo

Ejercicio 4.1.1. Sustituye `(robot-at r1 dock)` por `(robot-at dock r1)` en el problema sin tipos `examples/ch02/courier-p01.pddl`. Antes de ejecutar nada, predice qué responderán `kairos check` y `kairos`. Después ejecútalos y explica la nota que imprime Kairos.

Ejercicio 4.1.2. Un problema del mensajero sin tipos tiene 3 robots, 5 artículos y 12 habitaciones unidas por 15 conexiones (cada una en ambos sentidos). ¿Cuántas instanciaciones candidatas tienen en total las tres acciones? ¿Cuántas de ellas pueden aplicarse en algún estado alcanzable? Da de nuevo ambas cifras para el dominio con tipos.

Ejercicios de la Sección 4.2 — Tipos

Ejercicio 4.2.1. El dominio gripper: un robot con dos pinzas, `left` y `right`, lleva bolas entre habitaciones. Predicados `at-robby` (la habitación del robot), `at` (la habitación de una bola), `empty` (una pinza no sujeta nada) y `holding` (una pinza sujeta una bola); acciones `move`, `pick` y `drop`. Escríbelo con tipos y resuelve un problema que lleve tres bolas de `rooma` a `roomb`. ¿Cuántas candidatas tiene `pick`, y cuántas tendría sin tipos?

Ejercicio 4.2.2. Un problema del dominio del mensajero con tipos declara así sus objetos:

```
(:objects
  r1 - robot
  lobby corridor pharmacy lab ward-a ward-b dock - location
  meds1 - item
  icu)
```

¿Qué tipo tiene `icu`? ¿Qué hechos del estado inicial habitual rechazará el comprobador, y con qué mensaje? Comprueba tu respuesta con Kairos.

Ejercicio 4.2.3. ¿Es legal esta lista de parámetros? ¿Qué tipo tiene cada variable? `(?a ?b - robot ?c ?d)`. Reescríbela para que `?a` sea un robot y `?b` un artículo, dejando `?c` y `?d` como están.

Ejercicios de la Sección 4.3 — Jerarquías de tipos

Ejercicio 4.3.1. Con los objetos de `courier-team-p01` y la jerarquía del Listado 4.3, di cuáles de estas son acciones instanciadas o hechos legales (legal significa que los tipos encajan, no que la acción sea aplicable): (a) `(pick porter1 meds1 lobby)`; (b) `(serve porter1 meds1 icu)`; (c) `(carrying meal1 porter1)`; (d) `(serve porter1 meal1 dock)`; (e) `(move meal1 lobby corridor)`.

Ejercicio 4.3.2. Modela un zoo a partir de esta historia. *En un zoo trabajan cuidadores y veterinarios, que se desplazan a pie entre sus recintos. Los animales viven en recintos. Cualquier cuidador puede alimentar a un herbívoro (cebras, jirafas) en el recinto donde vive. A los grandes felinos (leones, tigres) solo los alimenta un cuidador sénior; un cuidador sénior puede hacer todo lo que hace un cuidador. Solo un veterinario trata a un animal, de cualquier clase.* Dibuja primero la jerarquía, escribe el dominio y resuelve un problema con un cuidador, un cuidador sénior, un veterinario, un león, una cebra y una jirafa en el que hay que alimentar a todos los animales y tratar a la jirafa. Escribe un plan en el que el cuidador normal alimenta al león y mira qué dice `kairos validate`.

Ejercicio 4.3.3. En el Listado 4.4, ¿qué cambiaría en los planes si los dos parámetros de lugar de `fly` se declararan `place` en lugar de `airport`? ¿Qué criterio del paso L sale perjudicado? ¿Y si `drive` se declarara con `?t - vehicle`?

Ejercicios de la Sección 4.4 — Un parámetro de varios tipos: `either`

Ejercicio 4.4.1. Reescribe el dominio del equipo del Listado 4.3, sin `serve`, de modo que un único predicado `at` diga dónde están los portadores *y* los artículos. Hazlo dos veces: con `either` y con un supertipo nuevo. Resuelve el mismo problema con ambos. ¿Cuál de los dos le darías a un compañero que usa otro planificador?

Ejercicio 4.4.2. La biblioteca compra audiolibros, que se pueden tomar prestados, y decide que los DVD y los audiolibros (pero no los libros) necesitan un reproductor, que el socio debe tener antes de llevárselos. Explica por qué la jerarquía `loanable` de la Sección 4.4 no puede expresar ambas agrupaciones solo con tipos, y propón una codificación.

Ejercicios de la Sección 4.5 — Constantes

Ejercicio 4.5.1. Añade al dominio del mensajero con tipos una constante `dock` y una acción `park`, con la que un robot que está en la base de carga y no lleva nada termina su turno: `(parked ?r)`. Escribe un problema cuyo objetivo sea entregar `meds1` en la sala A y quedar aparcado, con el robot empezando en la base de carga. Haz la prueba de la trampa: lee el plan. Después repara el modelo usando solo lo que sabes hasta ahora (sin condiciones negativas).

Ejercicio 4.5.2. Para cada uno de estos objetos, di si lo harías constante de su dominio, y por qué: (a) la mesa del mundo de los bloques; (b) la ciudad `madrid` en logistics; (c) las pinzas `left` y `right` del dominio gripper; (d) el laboratorio, en un mundo del mensajero en el que las muestras se analizan con una acción `analyze`.

Ejercicios de la Sección 4.6 — Igualdad

Ejercicio 4.6.1. Dos robots que están en la misma habitación y llevan un artículo cada uno pueden intercambiarlos. Escribe la acción `swap` para el dominio del mensajero con tipos, primero sin desigualdad. ¿Es aplicable `(swap r1 r1 meds1 meds1 lab)` cuando `r1` lleva `meds1` en el laboratorio? ¿Qué cambia? Cuenta las acciones instanciadas con `kairos -v` con y sin `(not (= ?a ?b))`, en un problema en el que el intercambio sea útil.

Ejercicio 4.6.2. Añade desigualdades a `drive` y `fly` del Listado 4.4. Sin ellas, Kairos instancia `logistics-p01` en 24 acciones. Predice el número con ellas y di qué acciones desaparecen. ¿Cambia el plan?

Ejercicio 4.6.3. Un estudiante añade `(not (= ?from ?to))` a `move` en el dominio del mensajero «por si acaso». ¿Cambia los planes? ¿Y las acciones instanciadas? Da una variante del mundo del mensajero en la que sí importaría.

Ejercicios de la Sección 4.7 — El método MODEL: clases de cosas

Ejercicio 4.7.1. Aplica el paso O a esta historia y elabora la lista de clases, con el motivo de cada tipo, y la tabla de estado. *Un almacén tiene estanterías y puestos de empaquetado unidos por pasillos. Las carretillas elevadoras y las carretillas de mano mueven palés; solo una carretilla elevadora puede coger un palé de una estantería alta. Algunos palés llevan productos congelados y solo pueden dejarse en un puesto frío.* ¿Qué palabras de la historia se convierten en tipos, cuáles en predicados, y hay alguna constante?

Ejercicio 4.7.2. Un compañero propone modelar «el robot `r2` está averiado» declarando `r2 - broken-robot`, un tipo nuevo que ninguna acción admite. Evalúa la propuesta con los cinco criterios del paso L frente a un predicado `(working ?r)` que exijan todas las acciones.

Condiciones más ricas | 5.

Todas las precondiciones y todos los objetivos escritos hasta ahora son listas de hechos que deben ser ciertos a la vez. Eso basta para una planta de salas abiertas, y deja de bastar en la primera puerta. Un robot solo puede abrir una puerta si la puerta *no* está abierta, y nuestras precondiciones solo saben pedir hechos presentes. Un robot puede entrar en la UCI si tiene una tarjeta de acceso *o* si otro robot con tarjeta le deja pasar, y nuestras precondiciones solo saben decir «y». El turno de noche termina cuando se han entregado *todos* los artículos y *no* queda ninguno en el pasillo, y nuestros objetivos solo pueden nombrar los artículos uno a uno, así que un objetivo escrito para tres artículos está mal para cuatro.

Este capítulo amplía lo que puede decir una condición. Además de hechos unidos por `and`, una precondición o un objetivo puede contener `not` (el hecho está ausente), `or` e `imply` (alternativas), `exists` (algún objeto la cumple) y `forall` (todos la cumplen). Cada uno tiene su propio indicador de requisito, y cada uno se puede evitar con hechos simples y algo de trabajo extra, así que para cada uno nos preguntamos también cuándo merece la pena usarlo. Los efectos siguen como estaban: las acciones siguen añadiendo y borrando hechos. Los efectos que dependen de la situación son el tema del Capítulo 6.

El mundo del mensajero gana puertas, que pueden estar abiertas, cerradas o cerradas con llave (precondiciones negativas y objetivos negativos), salas restringidas que exigen una tarjeta de acceso (`or` e `imply`), una escolta para los robots que no la tienen (`exists`) y objetivos sobre todos los artículos y todas las puertas (`forall`). El mundo de bloques y un rompecabezas de llaves y cofres muestran las mismas construcciones fuera del hospital. Una sección breve examina qué hace un planificador con estas condiciones, la sección del método añade cinco preguntas a los pasos O y D y compara tres pares de codificaciones, y el capítulo termina con los errores habituales.

5.1 Condiciones sobre la ausencia: precondiciones negativas 100
Tres estados con dos predicados 102
Ensayar las puertas . . . 102
5.2 Objetivos negativos . . . 103
5.3 Alternativas en una condición: `or` e `imply` . . 104
5.4 Algún objeto: `exists` . . 105
Un rompecabezas de llaves y cofres 106
5.5 Todos los objetos: `forall` 107
El mundo de bloques: `clear` es la ausencia de `on` 108
Un solo `move` para todos los pasos 109
Objetivos cuantificados . 109
5.6 Qué hace un planificador con las condiciones ricas 110
5.7 El método MODEL: ausencia, alguno, todos . 111
Alternativas comparadas 112
5.8 Errores comunes 114
5.9 Resumen 115
5.10 Lecturas complementarias 116
5.11 Ejercicios 116

5.1. Condiciones sobre la ausencia: precondiciones negativas

Tres pasos de la planta del hospital reciben una puerta: los de la farmacia, el laboratorio y la UCI (Figura 5.1). Un robot solo puede cruzar una puerta cuando está abierta, y puede abrir una puerta que esté cerrada. Así que el modelo necesita saber, para cada puerta, si está abierta, y necesita una acción cuya precondición sea que la puerta *no* está abierta.

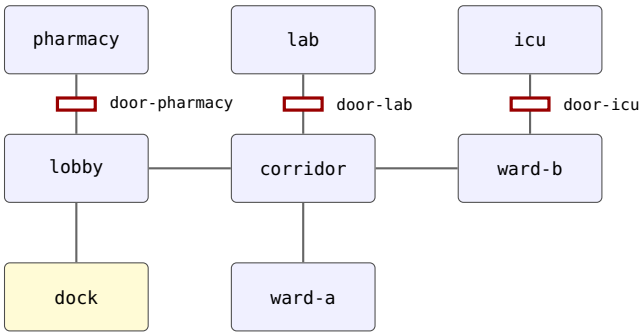


Figura 5.1.: La planta con puertas. Los cuatro pasos sin marca siguen siendo hechos `connected`; los tres que tienen puerta pasan a ser hechos `joins`, uno por sentido.

En STRIPS una precondición solo puede exigir que ciertos hechos sean verdaderos. Con el requisito `:negative-preconditions`, una precondición también puede contener `(not (p ...))`, una **precondición negativa**, que se cumple en un estado cuando el hecho `(p ...)` *no* está en él. El `not` de un efecto borra un hecho; el `not` de una precondición comprueba que falta un hecho. Se escriben igual y significan cosas distintas.

```
;; The courier world with doors. A door is open, closed (not
;; open, not locked) or locked. Needs negative preconditions.
(define (domain courier-doors)
  (:requirements :strips :typing :negative-preconditions)

  (:types
    robot location door
    medicine sample meal - item)

  (:predicates
    (connected ?from ?to - location)    ; a passage with no door
    (joins ?d - door ?a ?b - location)  ; ?d leads from ?a to ?b
    (open ?d - door)
    (locked ?d - door)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (has-keycard ?r - robot))
  ;; Through a door: only if it is open.
  (:action move-through
    :parameters (?r - robot ?d - door ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (joins ?d ?from ?to)
                       (open ?d))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to)))

  ;; A closed door: one that is not open and not locked.
  (:action open-door
    :parameters (?r - robot ?d - door ?a ?b - location)
    :precondition (and (robot-at ?r ?a) (joins ?d ?a ?b)
                       (not (open ?d)) (not (locked ?d)))
    :effect (open ?d))

  (:action close-door
    :parameters (?r - robot ?d - door ?a ?b - location)
    :precondition (and (robot-at ?r ?a) (joins ?d ?a ?b)
                       (open ?d))
    :effect (not (open ?d)))

  (:action unlock
    :parameters (?r - robot ?d - door ?a ?b - location)
    :precondition (and (robot-at ?r ?a) (joins ?d ?a ?b)
                       (locked ?d) (has-keycard ?r))
    :effect (not (locked ?d)))
```

Listado 5.1: El mundo del mensajero con puertas; `move`, `pick` y `drop` son los del Capítulo 4 (`examples/ch05/courier-doors-domain.pddl`).

El dominio usa los nombres que conserva el resto del libro: un tipo `door`, `(joins ?d ?a ?b)` para una puerta entre dos salas, `(open ?d)` y `(locked`

`?d)`. Ahora un paso entre salas puede ser de dos clases. Sin puerta, es un hecho `connected` y se cruza con el `move` de siempre. Con puerta, es un hecho `joins` y se cruza con `move-through`, que exige que la puerta esté abierta. En `open-door` aparecen dos precondiciones negativas: la puerta no está abierta y no está cerrada con llave.

Tres estados con dos predicados

Una puerta puede estar en tres estados, y ningún predicado se llama `closed`. Llamamos «cerrada» a una puerta que no está ni abierta ni cerrada con llave. La Figura 5.2 muestra los estados, las acciones que llevan de uno a otro y cómo los codifican dos predicados.

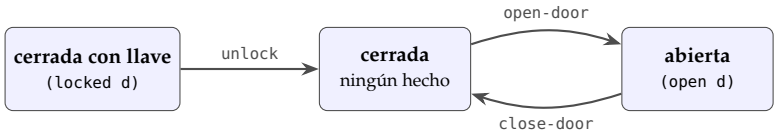


Figura 5.2.: Los tres estados de una puerta y los hechos que codifican cada uno. La puerta cerrada es aquella sobre la que el estado no dice nada.

Es la hipótesis del mundo cerrado del Capítulo 2 puesta a trabajar. Un estado enumera los hechos que son ciertos; todo lo demás es falso. Así que `(not (open door-lab))` no necesita nada en el estado para cumplirse: se cumple porque `(open door-lab)` no está. El estado inicial de `courier-doors-p01`, donde todas las puertas están cerradas, sencillamente no tiene ningún hecho sobre el estado de las puertas.

```
$ kairos courier-doors-domain.pddl courier-doors-p01.pddl
; cost = 8
(move r1 dock lobby)
(open-door r1 door-pharmacy lobby pharmacy)
(move-through r1 door-pharmacy lobby pharmacy)
(pick r1 meds1 pharmacy)
(move-through r1 door-pharmacy pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

El plan del Capítulo 4 tenía siete pasos; este tiene ocho, porque hay que abrir la puerta una vez. Nada pide al robot que vuelva a cerrarla, así que se queda abierta.

Una precondición negativa se cumple cuando el hecho no está en el estado. Con la hipótesis del mundo cerrado, un solo predicado codifica los dos lados de una propiedad de sí o no.

Ensayar las puertas

El paso E con las puertas: cuatro problemas

La batería de pruebas del paso E, para las puertas. *Trivial*: `p01`, el de arriba, con una puerta cerrada en el camino. *Típica*: `p03`, una muestra que va de la UCI al laboratorio, con la puerta de la UCI cerrada con llave y un robot que tiene tarjeta. *Sin plan*: `p04`, lo mismo sin la tarjeta. *Prueba de la trampa*: otra vez `p04`, con cada versión anterior del dominio: ¿alguna de ellas consigue que un robot cruce una puerta cerrada con llave?

El problema típico se resuelve en doce pasos:

```
$ kairos courier-doors-domain.pddl courier-doors-p03.pddl
; cost = 12
(move r1 dock lobby)
(move r1 lobby corridor)
(open-door r1 door-lab corridor lab)
(move r1 corridor ward-b)
(unlock r1 door-icu ward-b icu)
(open-door r1 door-icu ward-b icu)
(move-through r1 door-icu ward-b icu)
(pick r1 sample1 icu)
(move-through r1 door-icu icu ward-b)
(move r1 ward-b corridor)
(move-through r1 door-lab corridor lab)
(drop r1 sample1 lab)
```

La puerta cerrada con llave necesita dos pasos, `unlock` y después `open-door`, en ese orden: `open-door` exige `(not (locked ?d))`. La puerta del laboratorio se abre a la ida, cuando el robot pasa por delante. Cualquier otro momento valdría igual; un plan es un orden correcto, no el orden que elegiría una persona.

Sin la tarjeta, Kairos demuestra que no hay plan:

```
$ kairos courier-doors-domain.pddl courier-doors-p04.pddl
note: no plan: every state reachable from the initial one was
     searched (1 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

Aquí la prueba de la trampa se gana su sitio. Una primera versión del dominio (`courier-doors-v0-domain.pddl`) tenía un `open-door` con una sola precondición negativa, `(not (open ?d))`. Resuelve `p01` con el mismo plan, y también «resuelve» el problema que no debe tener plan:

```
$ kairos courier-doors-v0-domain.pddl courier-doors-p04.pddl
; cost = 11
...
(move r1 corridor ward-b)
(open-door r1 door-icu ward-b icu)
(move-through r1 door-icu ward-b icu)
...
```

Una puerta cerrada con llave no está abierta, así que el robot la abrió. Con dos predicados hay cuatro combinaciones, y la cuarta, una puerta abierta *y* cerrada con llave, se había vuelto alcanzable. Cada vez que una precondición negativa defina un estado por lo que falta, comprueba qué otros estados tienen esa misma ausencia.

5.2. Objetivos negativos

El farmacéutico se queja: el robot deja la puerta abierta. El objetivo tiene que decir que, al final, la puerta de la farmacia no está abierta. Un objetivo es una condición, igual que una precondición, y el mismo requisito, `:negative-preconditions`, permite usar `not` en él. El problema `courier-doors-p02` es `p01` con un **objetivo negativo**:

```
(:goal (and (item-at meds1 ward-a)
            (not (open door-pharmacy))))
```

El plan gana un paso, justo después de que el robot salga de la farmacia:

```
$ kairos courier-doors-domain.pddl courier-doors-p02.pddl
; cost = 9
...
(move-through r1 door-pharmacy pharmacy lobby)
(close-door r1 door-pharmacy lobby pharmacy)
(move r1 lobby corridor)
...
```

Un objetivo positivo pide al planificador que haga cierto algo, y alguien tiene que hacerlo. Un objetivo negativo se satisface con un hecho que falta, y un hecho puede faltar por razones en las que el modelador no pensó. El objetivo negativo de `p02` ya es cierto en el estado inicial; cuesta un paso solo porque la entrega obliga a abrir la puerta por el camino. Un problema cuyos objetivos son todos negativos suele tener como solución el plan vacío. Lo volveremos a encontrar, como trampa, en la Sección 5.5.

Un objetivo negativo pide que un hecho falte al final. A menudo es cierto desde el principio, y puede hacerse cierto de maneras que nadie pretendía.

Las condiciones negativas no se aceptan en todas partes. De los planificadores que compara este libro, POPF y OPTIC no admiten ni precondiciones negativas ni objetivos negativos; los demás planificadores clásicos de la comparación leen ambos. La Sección 5.7 muestra la codificación que necesitan esos planificadores.

5.3. Alternativas en una condición: `or` e `imply`

La UCI y el laboratorio pasan a ser salas **restringidas**: solo puede entrar en ellas un robot con tarjeta de acceso. La regla se refiere a la sala *en la que* entra el robot, así que va en la precondición de `move`. Con palabras: «el destino no está restringido, o el robot tiene tarjeta». Una lista de hechos unidos por `and` no puede decirlo, porque hay dos maneras distintas de tener permiso para entrar.

El requisito `:disjunctive-preconditions` añade dos conectivas. `(or a b ...)` se cumple cuando se cumple al menos una de sus partes. `(imply a b)` se cumple cuando `a` es falsa o `b` es cierta: «si `a`, entonces `b`». Se pueden anidar libremente con `and` y `not`. Para que esta sección trate de una sola cosa, el dominio `courier-access` no tiene puertas: es el dominio con tipos del Capítulo 4 con dos predicados estáticos, `(restricted ?l)` y `(has-keycard ?r)`, y este `move`:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (or (not (restricted ?to))
                         (has-keycard ?r)))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

La misma regla se lee con más naturalidad como una implicación (`courier-access-imply-domain.pddl`): «si la sala está restringida, el robot tiene tarjeta».

```
                     (imply (restricted ?to)
                            (has-keycard ?r))
```

Las dos significan lo mismo: `(imply a b)` es una abreviatura de `(or (not a) b)`. Elige la que se lea como la regla de la historia. En `courier-access-p01` los medicamentos están en la farmacia, con `r1`

a su lado, y deben llegar a la UCI; solo `r2`, que está en la sala A, tiene tarjeta.

```
$ kairos courier-access-domain.pddl courier-access-p01.pddl
; cost = 9
(move r2 ward-a corridor)
(move r2 corridor lobby)
(move r2 lobby pharmacy)
(pick r2 meds1 pharmacy)
(move r2 pharmacy lobby)
(move r2 lobby corridor)
(move r2 corridor ward-b)
(move r2 ward-b icu)
(drop r2 meds1 icu)
```

El robot que está junto a los medicamentos no hace nada: podría llevarlos hasta la sala B, pero hacer allí el relevo costaría más pasos que dejar que `r2` haga todo el viaje. Si nadie tiene tarjeta (`courier-access-p02`), no hay plan, y la nota de Kairos muestra la disyunción por dentro:

```
$ kairos courier-access-domain.pddl courier-access-p02.pddl
note: the goal needs atoms that no action can reach
(item-at meds1 icu) can never be reached:
  drop adds it, but needs (robot-at ?r - robot icu), which never
     holds
    move adds it, but is never applicable (types, negative or
     numeric conditions)
    move adds it, but needs (has-keycard ?r - robot), which never
     holds
      no action adds it, and the initial state does not have it [
     orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

Hay dos líneas sobre `move`, una por cada manera de satisfacer el `or`. La primera exige que la UCI no esté restringida, lo que nunca se cumple; la segunda exige una tarjeta, que no tiene nadie. El planificador ha partido la acción en dos, una por alternativa. Volveremos sobre ello en la Sección 5.6.

`(or a b)` se cumple cuando se cumple alguna de sus partes; `(imply a b)` es «si a, entonces b». Las dos necesitan `:disjunctive-preconditions`.

5.4. Algún objeto: `exists`

El hospital relaja la regla. Un robot sin tarjeta puede entrar en una sala restringida si va escoltado: algún robot que tiene tarjeta está a su lado, en la sala de la que viene. La precondición de `move` recibe una tercera alternativa, y esa alternativa no habla de los parámetros de la acción. Habla de *algún* robot, sea cual sea.

Una opción es añadir un parámetro `?e` para la escolta. La otra es decir «hay uno» dentro de la condición. PDDL tiene una construcción para eso, con un indicador de requisito propio, `:existential-preconditions`: `(exists (?x - t) c)` se cumple cuando la condición `c` se cumple para al menos un objeto de tipo `t`. La variable se declara como un parámetro, con su tipo, y solo existe dentro de los paréntesis del `exists`. El dominio `courier-escort`:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition
    (and (robot-at ?r ?from) (connected ?from ?to)
         (or (not (restricted ?to))
```

```
              (has-keycard ?r)
              (exists (?e - robot)          ; an escort
                (and (has-keycard ?e) (robot-at ?e ?from)))))
   :effect (and (not (robot-at ?r ?from))
                (robot-at ?r ?to)))
```

Listado 5.2: El `move` de `examples/ch05/courier-escort-domain.pddl`: tres maneras de tener permiso para entrar en una sala.

La condición dentro del `exists` mezcla su propia variable, `?e`, con un parámetro de la acción, `?from`. En la misma situación de antes, el plan cambia:

```
$ kairos courier-escort-domain.pddl courier-escort-p01.pddl
; cost = 8
(move r2 ward-a corridor)
(move r2 corridor ward-b)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
```

Ahora `r1` lleva los medicamentos durante todo el camino, y `r2` solo va hasta la sala B y espera allí. El paso `(move r1 ward-b icu)` es legal porque, en ese momento, hay un robot con tarjeta en la sala B. El plan no dice qué robot: la escolta no es un argumento del paso. Para saberlo hay que seguir el estado. Ese es el precio de `exists`, y también su utilidad: la acción conserva sus tres parámetros, y el planificador no tiene que elegir una escolta cuando le vale cualquiera.

¿Hace falta aquí la desigualdad del Capítulo 4, para que un robot no se escolte a sí mismo? No. Si `?e` se liga a `?r`, la condición pide que `?r` tenga tarjeta, y entonces la segunda alternativa del `or` ya le deja pasar.

Un rompecabezas de llaves y cofres

Una fila de cofres. Cada cofre se abre con ciertas llaves y guarda más llaves. Empiezas con una llave y quieres abrir el cofre del tesoro. «Tienes alguna llave que encaja» es un `exists`:

```
;; A puzzle of keys and chests: a chest opens if you have SOME
;; key that fits it. Chests hold more keys.
(define (domain chests)
  (:requirements :strips :typing :negative-preconditions
                 :existential-preconditions)

  (:types chest key)

  (:predicates
    (have ?k - key)
    (inside ?k - key ?c - chest)
    (fits ?k - key ?c - chest)
    (opened ?c - chest))

  (:action open
    :parameters (?c - chest)
    :precondition (and (not (opened ?c))
                       (exists (?k - key)
                         (and (have ?k) (fits ?k ?c))))
    :effect (opened ?c))

  (:action take
    :parameters (?k - key ?c - chest)
    :precondition (and (opened ?c) (inside ?k ?c))
    :effect (and (not (inside ?k ?c)) (have ?k))))
```

En `chests-p01` (Figura 5.3) la llave de hierro abre los cofres de roble y de pino. El de roble guarda la llave de latón, que abre el cofre de cedro, que guarda la llave de oro del tesoro. El cofre de pino es una distracción: su llave de plata solo encaja en el propio cofre de pino.

Listado 5.3: Un rompecabezas con `exists` (`examples/ch05/chests-domain.pddl`).

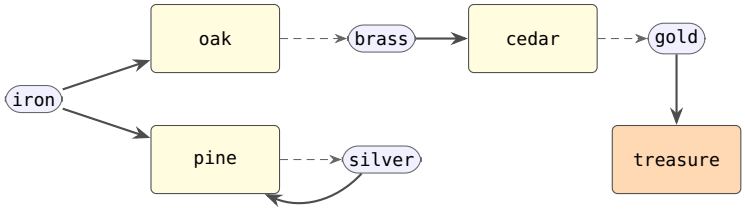


Figura 5.3.: El problema `chests-p01`. Una flecha continua de una llave a un cofre indica que la llave encaja; una flecha discontinua de un cofre a una llave indica que la llave está dentro.

```
$ kairos chests-domain.pddl chests-p01.pddl
; cost = 5
(open oak)
(take brass oak)
(open cedar)
(take gold cedar)
(open treasure)
```

Cambiemos ahora una regla: una llave se rompe al usarla. El efecto de `open` debe borrar `(have ?k)` para la llave usada, y un efecto no puede mencionar la variable de un `exists`, que no existe fuera de la condición. La llave tiene que pasar a ser un parámetro (`chests-once-domain.pddl`):

```
(:action open
  :parameters (?c - chest ?k - key)
  :precondition (and (not (opened ?c)) (have ?k) (fits ?k ?c))
  :effect (and (opened ?c) (not (have ?k))))
```

Ahora el rompecabezas es más difícil, porque abrir primero el cofre de pino malgasta la llave de hierro, y el plan nombra cada llave: `(open oak iron)`, `(open cedar brass)`, `(open treasure gold)`.

`(exists (?x - t) c)` se cumple cuando `c` se cumple para algún objeto del tipo. Úsalo cuando la acción necesita a alguien y no le hace nada; cuando un efecto debe nombrar al objeto, conviértelo en parámetro.

5.5. Todos los objetos: `forall`

La última conectiva habla de todos los objetos de un tipo. Con el requisito `:universal-preconditions`, `(forall (?x - t) c)` se cumple cuando la condición `c` se cumple para *todos* los objetos de tipo `t`. `exists` y `forall` son los dos **cuantificadores**, y el requisito `:quantified-preconditions` declara ambos a la vez.

Los objetos de un problema son una lista finita, así que un cuantificador es una abreviatura. Con tres puertas, la Figura 5.4 muestra qué representa cada uno.

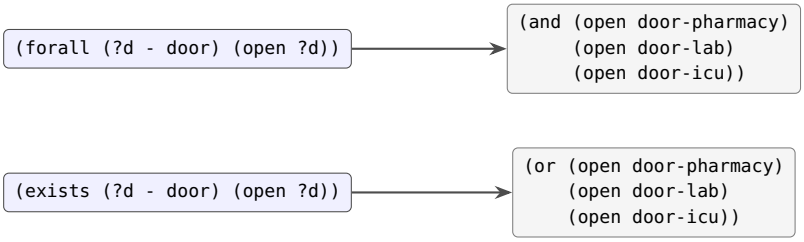


Figura 5.4.: Conocidos los objetos de un problema, `forall` es una conjunción y `exists` una disyunción, con una parte por cada objeto del tipo. El dominio se escribe una vez; la expansión es distinta en cada problema.

De la expansión se siguen dos consecuencias. Si el tipo no tiene objetos, el `forall` es una conjunción vacía, que es cierta, y el `exists` una disyunción vacía, que es falsa. Y los dos están ligados por la negación: «ninguna puerta está abierta» puede escribirse `(not (exists (?d - door) (open ?d)))` o `(forall (?d - door) (not (open ?d)))`.

El mundo de bloques: `clear` es la ausencia de `on`

El mundo de bloques del Capítulo 2 mantiene un predicado `(clear ?x)`: no hay nada encima del bloque `?x`. Cada acción que mueve un bloque añade un hecho `clear` y borra otro para que siga fiel a las torres. Pero `clear` no dice nada nuevo. Un bloque está despejado cuando no hay ningún bloque encima, y eso se puede preguntar directamente:

```
(:action pick-up
  :parameters (?x - block)
  :precondition (and (on-table ?x) (hand-empty)
                     (not (exists (?z - block) (on ?z ?x))))
  :effect (and (not (on-table ?x)) (not (hand-empty))
               (holding ?x)))
;; ?y must be another block, with nothing on it.
(:action stack
  :parameters (?x ?y - block)
  :precondition (and (holding ?x) (not (= ?x ?y))
                     (not (exists (?z - block) (on ?z ?y))))
  :effect (and (not (holding ?x)) (hand-empty)
               (on ?x ?y)))
```

Listado 5.4: Dos acciones del mundo de bloques sin `clear` (`examples/ch05/blocks-domain.pddl`).

Los efectos son más cortos que los clásicos, porque hay un predicado menos que mantener al día, y el estado inicial de un problema solo describe las torres (Figura 5.5). La desigualdad de `stack` estaba oculta en el dominio clásico: allí, un bloque en la mano no está `clear`, así que no se podía apilar sobre sí mismo. Aquí no hay nada encima del bloque que se sostiene, y sin `(not (= ?x ?y))` la acción `(stack a a)` sería aplicable.

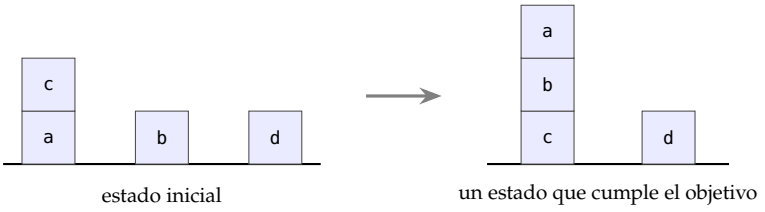


Figura 5.5.: El problema `blocks-p01`. Su estado inicial tiene cinco hechos: `(on c a)`, tres hechos `on-table` y `(hand-empty)`. Qué bloques están despejados se deduce de la ausencia de hechos `on`.

```
$ kairos blocks-domain.pddl blocks-p01.pddl
; cost = 6
(unstack c a)
(put-down c)
(pick-up b)
(stack b c)
(pick-up a)
(stack a b)
```

El dominio clásico (`blocks-clear-domain.pddl`, con un problema que enumera tres hechos `clear`) devuelve los mismos seis pasos. La Sección 5.7 compara los dos estilos con las puertas.

Un solo `move` para todos los pasos

Las puertas de la Sección 5.1 necesitaban dos acciones para moverse, una por cada clase de paso entre salas. Un cuantificador puede fusionarlas. Hagamos que todo paso vuelva a ser un hecho `connected`, y que `joins` solo diga qué pasos tienen puerta. Entonces un robot puede moverse cuando *todas las puertas de ese paso están abiertas*:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition
    (and (robot-at ?r ?from) (connected ?from ?to)
         (forall (?d - door)
           (imply (joins ?d ?from ?to) (open ?d))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

Listado 5.5: Un solo `move` (`examples/ch05/courier-ward-domain.pddl`).

El patrón `(forall (?x - t) (imply a b))` es la manera de decir «todo `x` que cumple `a` cumple también `b`». El cuantificador recorre todas las puertas de la planta; el `imply` descarta las que no están en este paso. En un paso sin puerta, todas las implicaciones son ciertas porque su primera parte es falsa, y el robot se mueve libremente. Escribir `and` donde va `imply` es el error más común con `forall`, y tiene su caja en la Sección 5.8.

Objetivos cuantificados

Los objetivos también admiten cuantificadores, con los mismos requisitos. Ahí es donde más se usa `forall`, porque hace que un objetivo no dependa de los objetos del problema. El turno de noche de `courier-ward-p01`:

```
(:goal (and (forall (?m - medicine) (item-at ?m icu))
            (forall (?d - door) (not (open ?d)))))
```

Todos los medicamentos en la UCI, todas las puertas cerradas. El tipo de la variable selecciona los objetos: el problema tiene también una muestra, y el primer `forall` no la toca. El mejor plan tiene 22 pasos; este es su final:

```
$ kairos courier-ward-domain.pddl courier-ward-p01.pddl
...
; better plan 3
; cost = 22
...
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(close-door r1 door-pharmacy lobby pharmacy)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
(close-door r1 door-icu icu ward-b)
```

El robot cierra la puerta de la farmacia tras su segunda y última visita, y cierra la puerta de la UCI desde dentro. Nada en el objetivo dice que el robot deba salir.

El otro objetivo de la introducción, «ningún artículo en el pasillo», es un `forall` sobre una condición negativa. Le aplicamos la prueba de la trampa. En `courier-ward-p02` hay una comida en el pasillo, el robot

también está allí, y el objetivo es `(forall (?i - item) (not (item-at ?i corridor)))`.

```
$ kairos courier-ward-domain.pddl courier-ward-p02.pddl
; cost = 1
(pick r1 meal1 corridor)
```

El robot coge la comida y se para. Un artículo en la pinza no está `item-at` en ninguna sala, así que el pasillo está «vacío». El objetivo decía dónde no debían estar los artículos y olvidaba decir dónde pueden estar. En `courier-ward-p03` se añade un segundo `forall`, `(forall (?r - robot) (free ?r))`, y el plan pasa a ser `pick`, `move`, `drop`: la comida acaba en la sala B.

`(forall (?x - t) c)` se cumple cuando `c` se cumple para todos los objetos del tipo, y también cuando no hay ninguno. Un objetivo sobre la ausencia debe decir además dónde pueden acabar las cosas.

De STRIPS a ADL

Las condiciones de este capítulo y los efectos del siguiente entraron en la planificación a través de ADL, el *Action Description Language* que Edwin Pednault propuso en 1989 Pednault, 1989 como término medio entre los hechos simples de STRIPS Fikes y Nilsson, 1971 y la lógica completa. Cuando se definió PDDL en 1998 McDermott et al., 1998, tomó esas construcciones y dio a cada una un indicador de requisito, además de un indicador `:adl` que las representa a todas juntas (Capítulo 6).

5.6. Qué hace un planificador con las condiciones ricas

Los algoritmos de búsqueda y las heurísticas son más sencillos cuando cada acción instanciada tiene como precondición una simple lista de hechos. Por eso la mayoría de los planificadores reescriben las condiciones ricas antes de buscar, y la reescritura se hace sobre el problema instanciado, donde ya se conocen los objetos Helmert, 2009. Conocerla a grandes rasgos ayuda a prever lo que cuesta una condición.

- **Los cuantificadores se expanden** sobre los objetos, como en la Figura 5.4. El tamaño de la condición crece con el número de objetos del tipo, y un cuantificador dentro de otro multiplica ambos.
- **Las partes estáticas se evalúan.** En el `move` del Listado 5.5, `(joins ?d ?from ?to)` no cambia nunca, así que cada implicación de la expansión o bien desaparece (allí no hay tal puerta) o bien se reduce a `(open d)`. La acción instanciada que va del vestíbulo a la farmacia se queda con una sola precondición sobre puertas, la que habría escrito una persona.
- **Las disyunciones parten las acciones.** Una acción cuya precondición es `(or a b)` se convierte en dos acciones con los mismos efectos, una que exige `a` y otra que exige `b`. Un `exists` es una disyunción, así que también se parte, en una acción por objeto.
- **La negación** de un hecho la tratan directamente muchos planificadores; otros añaden un segundo hecho que se mantiene opuesto al primero, el truco que la Sección 5.7 hace a mano.

La partición se puede observar con `kairos -v`. En `courier-access-p01`, el dominio con `or` tiene 64 acciones instanciadas. Un segundo fichero, `courier-access-split-domain.pddl`, escribe a mano los dos casos como dos acciones que se excluyen mutuamente (`move` para las salas

no restringidas, `move-with-keycard` para las salas restringidas y los robots con tarjeta) y tiene 52. Las 12 acciones de más son movimientos de `r2`, el robot con tarjeta, a salas no restringidas: se cumplen las dos partes del `or`, así que se conservan las dos copias de la acción, y las dos hacen lo mismo. Las búsquedas son idénticas (los mismos 352 estados), porque las acciones duplicadas llevan al mismo sucesor. La escolta cuenta la misma historia: 98 acciones instanciadas con `exists`, 74 cuando la escolta es un parámetro de una acción aparte (`courier-escort-param-domain.pddl`).

Los planificadores expanden los cuantificadores sobre los objetos y parten las disyunciones en varias acciones. Una condición rica cuesta lo mismo que aquello en lo que se expande.

Estas cifras son pequeñas. Dejan de serlo cuando las alternativas se multiplican: una conjunción de diez condiciones con dos alternativas cada una, todas sobre hechos que cambian, se expande en 2^{10} combinaciones en un planificador que las parta todas. La señal de alarma es un `forall` cuyo cuerpo contiene un `or`, un `imply` o un `exists` sobre hechos que no son estáticos.

5.7. El método MODEL: ausencia, alguno, todos

La ficha del modelo del mundo del mensajero cambia en su línea Alcance: las puertas y las reglas de acceso quedan ahora «dentro»; quién sujeta la puerta a quién y cuánto tarda una puerta en abrirse siguen «fuera». Los pasos O y D ganan preguntas, que se hacen a cada regla de la historia a medida que se convierte en una condición.

Tabla 5.1.: Lo que este capítulo añade al método MODEL.

Paso	Preguntas nuevas
O	**¿La propiedad es de sí o no, o tiene varios valores?** Si es de sí o no, guarda un lado y deja que la hipótesis del mundo cerrado dé el otro; elige el lado que es cierto con menos frecuencia, para que los estados iniciales sean cortos. Con tres valores o más, enumera las combinaciones de los predicados y comprueba que las que no tienen sentido no pueden alcanzarse.
D	**¿La condición trata de una ausencia?** «No abierta», «nadie dentro»: `not`.
D	**¿Trata de algún objeto?** «Algún robot con tarjeta»: `exists` si la acción no le hace nada a ese objeto; un parámetro si se lo hace o si el plan debe nombrarlo.
D	**¿Trata de todos los objetos?** «Todas las puertas cerradas»: `forall`, con `imply` para seleccionar de qué objetos habla la regla.
D	**¿Hay varias maneras de satisfacerla?** `or`, o una acción por cada manera.

El paso E gana dos pruebas de la trampa, las dos vistas más arriba. Para cada estado definido por una ausencia, busca otro estado con la misma ausencia (la puerta cerrada con llave que «no estaba abierta»). Para cada objetivo sobre una ausencia o sobre «todos», pregúntate cómo podría cumplirse por una razón tonta: el artículo en la pinza que no está en ninguna sala, el tipo que no tiene objetos.

Alternativas comparadas

Una precondición negativa o un predicado complementario. Las puertas se pueden modelar sin `not`. El dominio `courier-closed` tiene tres predicados, `open`, `closed` y `locked`, y cada acción borra uno y añade otro, de modo que para cada puerta se cumple exactamente uno de ellos:

```
(:action open-door
  :parameters (?r - robot ?d - door ?a ?b - location)
  :precondition (and (robot-at ?r ?a) (joins ?d ?a ?b)
                     (closed ?d))
  :effect (and (not (closed ?d)) (open ?d)))
```

`closed` es un **predicado complementario**: es cierto exactamente cuando otra condición es falsa, y los efectos lo mantienen así. Ahora el estado inicial debe decir `(closed door-lab)` para cada puerta cerrada. En los problemas `p01` y `p02` los dos dominios devuelven los mismos planes, tras visitar el mismo número de estados.

Tabla 5.2.: Precondiciones negativas frente a un predicado complementario, para las puertas.

Criterio	Precondiciones negativas	Predicado complementario
Fidelidad	Los mismos planes. Una combinación sin sentido (abierta y cerrada con llave) se puede escribir y, con una acción descuidada, alcanzar.	Los mismos planes. Una puerta sobre la que el estado inicial no dice nada no está en ningún estado, y no se le aplica ninguna acción.
Tamaño	21 hechos y 42 acciones instanciadas en `p01`.	24 hechos, 42 acciones instanciadas.
Guía	102 estados visitados.	102 estados visitados.
Claridad	Un predicado por propiedad; efectos y estados iniciales cortos. Tres estados se leen en dos predicados.	Un predicado por estado, cada uno con su nombre. Cada acción mantiene el grupo; cada problema enumera todas las puertas.
Portabilidad	No las aceptan POPF ni OPTIC.	STRIPS con tipos, sin más: cualquier planificador.

Veredicto. Para una propiedad de sí o no, usa la precondición negativa: no puede desacompasarse, porque no hay nada que mantener acompasado. Usa predicados complementarios cuando el modelo tenga que ejecutarse en un planificador sin precondiciones negativas (los planificadores temporales POPF y OPTIC son el caso habitual), y tenlos en cuenta cuando una propiedad tenga tres o más valores que merezcan nombre propio. El coste de un olvido es distinto. Con la negación, un hecho olvidado significa una puerta cerrada, que es un estado legal. Con predicados complementarios significa una puerta que no es nada: en `courier-closed-p01-forgot` la puerta de la farmacia no tiene ningún hecho, y Kairos responde «unsolvable» a un problema en el que su autor no ve nada raro.

`or` o dos acciones. La Sección 5.6 ya ejecutó las dos codificaciones de las salas restringidas, el dominio con `or` y el partido en dos.

Criterio	Una acción con `or`	Dos acciones
Fidelidad	Los mismos planes.	Los mismos planes, si los casos lo cubren todo. Un caso olvidado es una solución perdida.
Tamaño	64 acciones instanciadas (12 duplicadas).	52 acciones instanciadas.
Guía	352 estados visitados.	352 estados visitados.
Claridad	La regla está en un solo sitio. Todos los pasos son `move`.	La regla se reparte entre dos acciones que hay que cambiar a la vez. Los pasos dicen qué caso se usó: `move-with-keycard`.
Portabilidad	No la aceptan POPF ni OPTIC.	Solo necesita lo que necesita cada caso (aquí, una precondición negativa).

Tabla 5.3.: Una disyunción frente a una acción por alternativa, para las salas restringidas.

Veredicto. Usa `or` (o `imply`) cuando las alternativas sean un detalle de una sola regla y los efectos sean los mismos. Escribe dos acciones cuando el planificador que tienes a mano no admita disyunciones, cuando las alternativas vayan a diferir pronto en sus efectos o en sus costes, o cuando el plan deba distinguirlas. Con tres reglas independientes de dos alternativas cada una, la partición necesita ocho acciones; ahí la disyunción gana solo por Claridad.

Un objetivo `forall` o una lista. El objetivo de `courier-ward-p01` se puede escribir hecho a hecho, como en `courier-ward-p01-listed`: dos hechos `item-at` y tres negativos sobre las puertas.

Criterio	Objetivo `forall`	Objetivo enumerado
Fidelidad	Sigue a los objetos: un tercer medicamento queda cubierto sin tocar el objetivo. Cierto para un tipo sin objetos.	Dice exactamente lo que enumera. Un medicamento añadido a `:objects` y no al objetivo se queda fuera sin aviso.
Tamaño	46 hechos tras la instanciación.	45 hechos.
Guía	23 679 estados visitados para demostrar que el plan de 22 pasos es óptimo.	23 679 estados.
Claridad	Expresa la intención («todos los medicamentos»).	Expresa la instancia. Largo con muchos objetos.
Portabilidad	LPG-td y Metric-FF no aceptan condiciones cuantificadas.	Salvo por los objetivos negativos, cualquier planificador.

Tabla 5.4.: Un objetivo cuantificado frente al mismo objetivo enumerado, en `courier-ward-p01`.

Veredicto. Para el planificador son el mismo problema. Usa `forall` cuando el objetivo sea una regla («todo entregado», «nada olvidado») y cuando los problemas se escriban a mano y se editen a menudo. Usa la lista cuando los problemas los genere un programa, que no se olvida de ningún objeto, o cuando el planificador de destino no tenga cuantificadores.

5.8. Errores comunes

`and` bajo un `forall` donde tocaba `imply`

La regla «todas las puertas entre las dos salas están abiertas» no se escribe así:

```
(forall (?d - door)
  (and (joins ?d ?from ?to) (open ?d)))
```

Eso pide que todas las puertas del edificio unan estas dos salas *y* estén abiertas. Con tres puertas nunca es cierto, ningún robot se mueve jamás, y el síntoma queda lejos de la causa (`courier-ward-and-domain.pddl`):

```
$ kairos courier-ward-and-domain.pddl courier-ward-p01.pddl
note: no plan: every state reachable from the initial one was
     searched (1 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Nada en la respuesta apunta al cuantificador. Lo que sí apunta es el problema trivial del paso E: si un robot no puede dar un solo paso entre dos salas abiertas, mira `move`. La regla práctica: bajo `forall`, selecciona con `imply`; bajo `exists`, selecciona con `and`.

Un `forall` que se cumple sin objetos

En `courier-ward-p04-vacuous.pddl` los medicamentos se declararon `meds1 meds2 - item`, y el objetivo es `(forall (?m - medicine) (item-at ?m icu))`. No hay ningún objeto de tipo `medicine`, así que el objetivo se cumple en el estado inicial:

```
$ kairos courier-ward-domain.pddl courier-ward-p04-vacuous.pddl
; cost = 0
```

Un plan vacío con coste 0 es una respuesta válida, y `check` no tiene nada que objetar. Cuando un objetivo cuantificado dé un plan demasiado corto, cuenta los objetos de su tipo.

Una variable cuantificada sin tipo

`(forall (?m) (item-at ?m icu))` recorre *todos* los objetos: pide también que el robot, las salas y las puertas sean artículos que están en la UCI. Kairos acepta una variable de tipo `object` donde se espera un `item` (la condición es simplemente falsa para los objetos de otros tipos), así que la comprobación pasa y el problema no tiene plan (`courier-ward-p01-untyped.pddl`):

```
$ kairos courier-ward-domain.pddl courier-ward-p01-untyped.pddl
note: no plan: every state reachable from the initial one was
     searched (1 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Bajo `exists` el mismo descuido es inofensivo y bajo un `forall` negativo es invisible, y así es como sobrevive hasta el día en que importa. Pon siempre tipo a la variable de un cuantificador.

Requisitos olvidados

Cada construcción de este capítulo tiene su indicador. `courier-escort-noreq-domain.pddl` declara solo `:strips` y `:typing`:

```
$ kairos check courier-escort-noreq-domain.pddl
courier-escort-noreq-domain.pddl:4:18: warning: uses negative-
     preconditions but :requirements does not include :negative-
     preconditions [semantic.undeclared_requirement]
 4 |   (:requirements :strips :typing)
   |                  ^
courier-escort-noreq-domain.pddl:4:18: warning: uses disjunctive
     -preconditions but :requirements does not include :
     disjunctive-preconditions [semantic.undeclared_requirement]
...
valid, plannable
...
```

Kairos planifica de todos modos; otros planificadores rechazan el fichero o, peor aún, ignoran la parte que no esperaban. Los avisos son también la forma más rápida de saber qué usa de verdad un dominio y, por tanto, en qué planificadores puede ejecutarse.

`not` en el estado inicial

El estado inicial enumera lo que es cierto. Escribir allí `(not (open door-pharmacy))` no añade nada, porque la puerta ya no estaba abierta, y Kairos lo dice: «negative initial facts have no effect». El peligro es la creencia que hay detrás, que un hecho es desconocido hasta que se declara. En estos modelos nada es desconocido: lo que no está en la lista es falso.

Predicados complementarios desacompasados

El Capítulo 7 da nombre a este error: *gemelos desacompasados*. Un dominio con `open` y `closed` depende de que cada acción y cada estado inicial mantengan exactamente uno de los dos cierto. Un efecto olvidado (un `close-door` que añade `closed` y no borra `open`) da una puerta que es las dos cosas a la vez, por la que pasan los robots y que se puede volver a «abrir». Ningún mensaje lo avisa; los planes simplemente se vuelven extraños. Si un dominio tiene una pareja así, prueba cada acción por separado y lee el estado que deja.

5.9. Resumen

Una condición, en una precondición o en un objetivo, ya no es solo una lista de hechos. Puede exigir que un hecho falte, ofrecer alternativas y hablar de alguno o de todos los objetos de un tipo. Todo ello se evalúa sobre el estado actual bajo la hipótesis del mundo cerrado, y todo ello puede expandirse, una vez conocidos los objetos, en los hechos, `and`, `or` y `not` que abrevia.

Construcción	Se cumple cuando	Requisito
`(not c)`	`c` no se cumple	`:negative-preconditions`
`(or c1 c2 ...)`	se cumple alguna parte	`:disjunctive-preconditions`
`(imply c1 c2)`	`c1` no se cumple o `c2` se cumple	`:disjunctive-preconditions`
`(exists (?x - t) c)`	`c` se cumple para algún objeto de tipo `t`	`:existential-preconditions`
`(forall (?x - t) c)`	`c` se cumple para todo objeto de tipo `t`	`:universal-preconditions`

Tabla 5.5.: Las construcciones de este capítulo. `:quantified-preconditions` declara las dos últimas juntas. Los mismos indicadores sirven para los objetivos.

Para el método: pregunta de cada regla si trata de una ausencia, de algún objeto o de todos los objetos; prefiere una precondición negativa a un segundo predicado que haya que mantener opuesto; haz que un objeto sea un parámetro cuando un efecto lo toque o el plan deba nombrarlo, y usa un `exists` en otro caso; y desconfía de todo objetivo que pueda cumplirse porque falta algo.

5.10. Lecturas complementarias

El artículo de Pednault sobre ADL Pednault, 1989 es el origen de estas condiciones en planificación. El manual de PDDL McDermott et al., 1998 define su sintaxis y sus indicadores de requisito. El libro de texto de Haslum y colaboradores Haslum et al., 2019 trata las condiciones cuantificadas y disyuntivas de manera más formal. Cómo las elimina un planificador (expandiendo los cuantificadores, partiendo las disyunciones y empujando las negaciones hasta los hechos) se describe paso a paso para el traductor de Fast Downward Helmert, 2009. El libro de planificación de Ghallab, Nau y Traverso Ghallab et al., 2004 estudia estas extensiones de la representación clásica y lo que le cuestan a un planificador.

5.11. Ejercicios

Ejercicios de la Sección 5.1 — Condiciones sobre la ausencia: precondiciones negativas

Ejercicio 5.1.1. En `courier-doors-domain.pddl` una puerta se describe con dos predicados, así que existen cuatro combinaciones. Di qué estado codifica cada una. ¿Qué combinación no tiene sentido?

Argumenta, acción por acción, que no puede alcanzarse desde un estado inicial que no la contenga.

Ejercicio 5.1.2. Un robot necesita la pinza libre para abrir o cerrar una puerta. Añádelo al dominio y resuelve `courier-doors-p02`, donde la puerta de la farmacia debe acabar cerrada. Antes de ejecutarlo, predice cómo se las arreglará el robot con los medicamentos en la pinza.

Ejercicio 5.1.3. Añade una acción `lock`: un robot con tarjeta cierra con llave una puerta que está cerrada. Escribe un problema en el que el robot saque los medicamentos de la farmacia y la deje cerrada con llave. ¿Qué precondiciones negativas necesita `lock`, y qué le pasa al Ejercicio 5.1.1 si se olvida una?

Ejercicios de la Sección 5.2 — Objetivos negativos

Ejercicio 5.2.1. Cambia el objetivo de `courier-doors-p01` para que el robot entregue los medicamentos en la sala A y no se quede allí. ¿Cuántos pasos gana el plan, y dónde acaba el robot? ¿Es eso lo que quería la supervisora de enfermería?

Ejercicio 5.2.2. Un estudiante escribe `(not (open door-pharmacy))` en el `:init` de un problema «para asegurarse de que la puerta empieza cerrada». ¿Qué dice `kairos check`? ¿Cambia el plan?

Ejercicios de la Sección 5.3 — Alternativas en una condición: `or` e `imply`

Ejercicio 5.3.1. La Solución 4.4.2 necesitaba dos acciones `borrow` en la biblioteca, porque una precondición no podía decir «si la cosa necesita un reproductor, el socio tiene uno». Escribe la biblioteca con un único `borrow` y un `imply`, con un predicado estático `(needs-player ?x)`, y resuelve un problema en el que Ana quiere un audiolibro y no tiene reproductor.

Ejercicio 5.3.2. Demuestra con una tabla de verdad que `(imply a b)`, `(or (not a) b)` y `(not (and a (not b)))` se cumplen en los mismos estados. Después escribe la regla de las salas restringidas en la tercera forma y comprueba que los planes no cambian.

Ejercicios de la Sección 5.4 — Algún objeto: `exists`

Ejercicio 5.4.1. Regla nueva: ningún robot entra en una sala restringida mientras lleva una comida, tenga tarjeta o no. Añádela a `courier-access` y pruébala con un problema que tenga plan y otro que no deba tenerlo.

Ejercicio 5.4.2. En el dominio de la escolta, la escolta debe estar en la sala de la que sale el robot. Reescribe la condición para un hospital en el que la escolta espera *dentro* de la sala restringida y deja pasar al robot. ¿Sigue funcionando el plan de `courier-escort-p01`?

Ejercicio 5.4.3. En `chests-once-p01`, ¿qué pasa si el primer paso es `(open pine iron)`? ¿Por qué la versión con `exists` no tiene esa trampa? Di en una frase cuándo una variable tiene que ser un parámetro.

Ejercicios de la Sección 5.5 — Todos los objetos: `forall`

Ejercicio 5.5.1. La planta cierra por la noche con una acción `close-floor`, aplicable cuando todos los robots están en la base de carga y todas las puertas están cerradas; después de ella, ningún robot se mueve. Escríbela sobre `courier-ward-domain.pddl`, con `dock` como constante, y resuelve un problema con dos robots y una entrega cuyo objetivo incluya `(floor-closed)`.

Ejercicio 5.5.2. En una planta sin medicamentos, ¿qué responde un planificador al objetivo `(forall (?m - medicine) (item-at ?m icu))`? ¿Y a `(exists (?m - medicine) (item-at ?m icu))`? Comprueba el segundo con Kairos.

Ejercicio 5.5.3. En el mundo de bloques sin `clear`, escribe el objetivo «todos los bloques están sobre la mesa» y resuélvelo para una torre de tres bloques y un cuarto bloque aparte. Después escribe la condición «`?x` está despejado» con `forall` en lugar de `exists`.

Ejercicio 5.5.4. Escribe la negación de `(forall (?d - door) (imply (joins ?d ?from ?to) (open ?d)))` sin ningún `not` delante de un cuantificador. Exprésala con palabras.

Ejercicios de la Sección 5.6 — Qué hace un planificador con las condiciones ricas

Ejercicio 5.6.1. Una acción tiene la precondición `(and (or a b) (or c d) (or e f))`, donde los seis hechos cambian durante el plan. ¿En cuántas acciones simples la parte un planificador? Reescribe la condición del Listado 5.2 como la lista de alternativas que obtiene el planificador, con dos robots.

Ejercicios de la Sección 5.7 — El método MODEL: ausencia, alguno, todos

Ejercicio 5.7.1. Aplica el paso L a los dos mundos de bloques de este capítulo, `blocks-domain.pddl` y `blocks-clear-domain.pddl`. Ejecuta los dos con su primer problema y `kairos -v`, y rellena una tabla con los cinco criterios. ¿Cuál elegirías para un curso, y cuál para un planificador temporal?

Ejercicio 5.7.2. Modela una cafetera a partir de esta historia, con los pasos O y D por escrito. *La máquina prepara una bebida en la taza que hay bajo su boquilla, si la taza está vacía y hay agua y café en grano. Bajo la boquilla solo cabe una taza. Un capuchino necesita además leche en la máquina; un expreso no. Después de cada bebida hay que limpiar la máquina antes de la siguiente, y solo se puede limpiar cuando no hay ninguna taza bajo la boquilla.* Resuelve un problema que pida un expreso y un capuchino, que empiece sin leche y que termine con la máquina limpia.

6. Efectos más ricos

El capítulo anterior hizo las condiciones de una acción tan ricas como quisiéramos: un robot puede entrar en una habitación si la puerta está abierta, o si tiene una tarjeta de acceso, o si no se ha quedado ningún artículo en el pasillo. Los efectos siguieron como en STRIPS: una lista fija de hechos que añadir y hechos que borrar, la misma lista cada vez que se aplica la acción. Ahora el hospital compra un carro. El mensajero carga artículos en él y lo empuja de habitación en habitación, y lo que va en el carro viaja con él. ¿Cuántos hechos cambia un empujón? Depende: dos si el carro está vacío, cuatro con una caja encima, doce con cinco. Una lista fija no puede decir eso. Tampoco puede decir qué ocurre en la puerta del laboratorio, donde una esclusa desinfecta todo lo que entra: una muestra sellada pasa intacta, una abierta se estropea. El movimiento es el mismo; lo que cambia no.

Este capítulo da a los efectos la misma libertad que ya tienen las condiciones. Un **efecto condicional**, escrito con `when`, ocurre solo si una condición se cumple en el estado en el que se aplica la acción. Un **efecto universal**, escrito con `forall`, ocurre para cada objeto de un tipo. Juntos dicen «todo artículo que está en el carro cambia de habitación». Con ellos el lenguaje alcanza el nivel conocido como ADL, que es todo lo que PDDL 1.2 ofrece para la planificación clásica.

Empezamos con el carro y vemos por qué no bastan las herramientas de los capítulos anteriores. Después llegan las dos construcciones, primero cada una por separado en un dominio de lámparas y luego las dos juntas en tres modelos: el rompecabezas Lights Out, el maletín que dio a estos efectos su primer ejemplo famoso, y el carro. El laboratorio estéril muestra la otra cara de un efecto condicional: algo que el planificador debe procurar *no* desencadenar. Tras el requisito `:adl` y un vistazo a lo que hacen los planificadores con estos efectos, el capítulo añade dos preguntas al paso D del método MODEL y compara tres formas de codificar el carro.

6.1 Efectos que dependen de la situación 119
6.2 Efectos condicionales: `when` 121
Qué significa un efecto condicional 122
A ejecutarlo 122
Una condición en el efecto o en la precondición . . . 123
6.3 Efectos universales: `forall` 123
6.4 Los dos juntos: para todo objeto tal que 124
Lights Out 124
El maletín 126
El carro 127
6.5 Efectos que hay que evitar: el laboratorio estéril 130
Ensayar el modelo 130
Una acción por caso . . . 132
6.6 El requisito `:adl` 133
6.7 Qué hacen los planificadores con los efectos condicionales 133
6.8 El método MODEL: efectos que dependen de la situación 135
Alternativas comparadas 135
Una cuarta vía: no almacenar lo que se puede deducir 137
6.9 Errores comunes 138
6.10 Resumen 140
6.11 Lecturas complementarias 141
6.12 Ejercicios 141

6.1. Efectos que dependen de la situación

El carro del hospital es una carretilla con baldas. Un robot que lleva un artículo puede dejarlo en el carro; un robot con la mano libre puede coger el asa y empujar el carro hasta una habitación conectada. La Figura 6.1 muestra la entrega que queremos planificar: dos cajas de medicamentos que esperan en la farmacia, junto al carro, hacen falta en la UCI, a cuatro habitaciones de distancia.

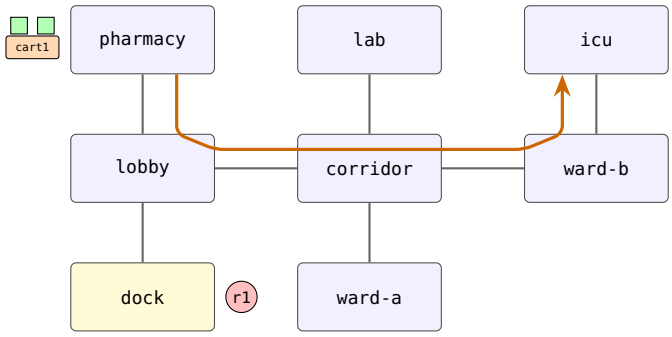


Figura 6.1.: La entrega con el carro. Las dos cajas (en verde) están en `cart1`, dibujado junto a la farmacia, donde se encuentra; la flecha naranja es el camino del carro hasta la UCI. Cada uno de los cuatro empujones cambia la posición del robot, del carro y de las dos cajas.

Siguiendo el vocabulario común del mundo del mensajero, el modelo gana un tipo `cart` y tres predicados: `(cart-at ?c ?l)`, `(on-cart ?i ?c)` y `(pushing ?r ?c)`. Decidimos que un artículo que está en el carro sigue estando *en algún sitio*: `(item-at meds1 pharmacy)` sigue siendo cierto mientras `meds1` está en un carro que está en la farmacia, de modo que los objetivos y las precondiciones siguen preguntando dónde está un artículo como lo han hecho siempre. El precio es que el empujón debe mantener esos hechos al día.

Intentemos escribir `push-cart` con lo que sabemos. El robot y el carro son fáciles: dos borrados y dos adiciones, como en `move`. El problema son los artículos. Un efecto solo puede nombrar los parámetros de la acción, así que cada artículo que se mueve necesita un parámetro, y una acción tiene un número fijo de parámetros. Nos vemos obligados a escribir una acción para cada número de artículos en el carro. El Listado 6.1 muestra la de dos artículos, de un dominio que tiene cuatro (de `push-cart-0` a `push-cart-3`).

```
90  ; Exactly ?i and ?j on the cart.
91  (:action push-cart-2
92    :parameters (?r - robot ?c - cart ?from ?to - location
93                 ?i ?j - item)
94    :precondition (and (pushing ?r ?c) (robot-at ?r ?from)
95                       (connected ?from ?to)
96                       (on-cart ?i ?c) (on-cart ?j ?c)
97                       (not (= ?i ?j))
98                       (forall (?k - item)
99                         (imply (on-cart ?k ?c)
100                                (or (= ?k ?i) (= ?k ?j)))))
101   :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
102                (not (cart-at ?c ?from)) (cart-at ?c ?to)
103                (not (item-at ?i ?from)) (item-at ?i ?to)
104                (not (item-at ?j ?from)) (item-at ?j ?to)))
```

Listado 6.1: Una acción por caso: empujar un carro que lleva exactamente dos artículos (de `courier-cart-cases-domain.pddl`).

La acción debe saber que `?i` y `?j` son *todo* lo que hay en el carro: si hubiera un tercer artículo, se quedaría atrás. Ese es el trabajo del `forall` de las líneas 98 a 100, una precondición universal del capítulo anterior. El modelo funciona. También tiene tres defectos que crecen con el problema:

- **Tiene un límite que nadie ha pedido.** Con cuatro acciones, el carro se mueve con hasta tres artículos. Pon un cuarto y no se aplica ningún empujón: el carro se queda atascado, y nada en el hospital explica por qué.
- **Es grande.** Hay dos parámetros de artículo en `push-cart-2` y tres en `push-cart-3`. Con n artículos y 14 conexiones dirigidas en el mapa, la versión para k artículos tiene $14\,n^k$ acciones instanciadas candidatas. Además, cada par se cuenta dos veces, como `(meds1 meds2)` y como `(meds2 meds1)`: dos acciones que hacen exactamente lo mismo.

- **Se repite.** Las cuatro acciones dicen lo mismo cuatro veces. Un cambio en la regla (un carro que no puede entrar en la UCI, por ejemplo) hay que hacerlo en cuatro sitios.

Un efecto STRIPS es una lista fija. Cuando el número de hechos que cambian, o cuáles cambian, depende del estado, la única salida es una acción por caso, y los casos se multiplican.

Lo que queremos escribir es una sola frase: *el robot y el carro cambian de habitación, y también todo artículo que está en el carro*. Esa frase tiene dos ingredientes que los efectos aún no tienen. «Todo artículo» es un cuantificador, y «que está en el carro» es una condición que se evalúa sobre el estado, dentro del efecto. Las dos secciones siguientes los presentan de uno en uno, en un dominio lo bastante pequeño como para ver todos sus estados.

6.2. Efectos condicionales: `when`

Un pasillo tiene lámparas, cada una con un pulsador. Pulsar el botón cambia la lámpara: si estaba apagada se enciende, y si estaba encendida se apaga. En STRIPS esto son dos acciones, `switch-on` con la precondición `(not (lit ?l))` y `switch-off` con la precondición `(lit ?l)`. Pero hay un solo botón, y una sola cosa que un agente puede hacer con él. El Listado 6.2 modela el botón tal como es: una acción, `toggle`, cuyo efecto depende del estado. (La segunda acción del fichero pertenece a la sección siguiente.)

```
;; Lamps with one push button each: pressing it changes the lamp.
;; What the action does depends on the state of the lamp.
(define (domain lights)
  (:requirements :strips :typing :negative-preconditions
                 :conditional-effects)

  (:types lamp)

  (:predicates
    (lit ?l - lamp))          ; the lamp is on

  ; One action for both cases: on becomes off, off becomes on.
  (:action toggle
    :parameters (?l - lamp)
    :effect (and (when (lit ?l) (not (lit ?l)))
                 (when (not (lit ?l)) (lit ?l))))

  ; The master switch: every lamp goes off, whatever it was.
  (:action all-off
    :parameters ()
    :effect (forall (?l - lamp) (not (lit ?l)))))
```

Listado 6.2: Las lámparas: `toggle` tiene dos efectos condicionales, `all-off` uno universal (`lights-domain.pddl`).

Un **efecto condicional** tiene la forma

```
(when <condition> <effect>)
```

y se lee «si la condición se cumple, el efecto también ocurre». La condición es una fórmula del misma clase que una precondición: puede usar `and`, `or`, `not`, la igualdad y los cuantificadores, siempre que se declaren los requisitos de los capítulos anteriores. El efecto es un literal o un `and` de literales. Los efectos condicionales necesitan el requisito `:conditional-effects`. La acción `toggle` declara además `:negative-preconditions`, porque la condición de su segundo `when` es una negación.

Fíjate en lo que `toggle` no tiene: una precondición. Una acción sin `:precondition` es aplicable en todos los estados. El botón siempre se puede pulsar; el estado solo decide qué hace pulsarlo.

Qué significa un efecto condicional

El significado de una acción con efectos condicionales lo dan tres reglas. Son breves, y la mayoría de los errores con estos efectos vienen de olvidar una de ellas.

1. **Las condiciones no deciden si la acción se puede aplicar.** Eso solo lo decide la precondición. Un `when` cuya condición es falsa se omite, y el resto de la acción ocurre como siempre.
2. **Todas las condiciones se evalúan en el estado anterior a la acción.** Ninguna condición ve los cambios que hace otro efecto de la misma acción.
3. **Todos los efectos que se disparan se aplican a la vez**: primero los borrados, después las adiciones. Si una acción borra y añade el mismo hecho, este acaba siendo cierto.

La Figura 6.2 dibuja las reglas para `(toggle l1)` sobre una lámpara encendida. En el estado anterior, `(lit l1)` es cierto: el primer `when` se dispara y el segundo no. El resultado es una lámpara apagada. La regla 2 es la que hace correcta la acción. Si los dos efectos se ejecutaran uno tras otro, como dos sentencias `if` de un programa, el primero apagaría la lámpara y el segundo, al encontrarla apagada, la volvería a encender.

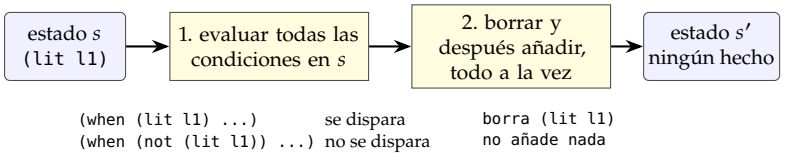


Figura 6.2.: Aplicación de `(toggle l1)` a una lámpara encendida. Las dos condiciones se leen en el estado anterior a la acción; después, los efectos que se disparan se aplican juntos. Ninguna condición se evalúa a medio camino.

Un when no bloquea la acción: selecciona parte de su efecto. Todas las condiciones se leen en el estado anterior a la acción, y todos los efectos que se disparan se aplican juntos.

A ejecutarlo

El primer problema tiene tres lámparas, de las que solo `l2` está encendida, y pide que esté encendida exactamente `l1`. El objetivo usa literales negativos, como enseñó el capítulo anterior: `(and (lit l1) (not (lit l2)) (not (lit l3)))`.

```
$ kairos check lights-domain.pddl lights-p01.pddl
valid, plannable
  typing: full
  negative-preconditions: full
  conditional-effects: full
$ kairos lights-domain.pddl lights-p01.pddl
; cost = 2
(toggle l1)
(toggle l2)
```

La misma acción aparece dos veces en el plan y hace cosas opuestas: enciende `l1` y apaga `l2`. Un plan con efectos condicionales no se puede leer acción por acción sin tener el estado presente. Para saber qué hizo un paso, hay que saber dónde se aplicó.

Una condición en el efecto o en la precondición

La misma fórmula puede ir en dos sitios de una acción y significar dos cosas distintas (Tabla 6.1). En la precondición dice cuándo la acción es *posible*. En un `when` dice lo que la acción *hace*. La pregunta que hay que hacerle al mundo es sencilla: cuando la condición es falsa, ¿puede el agente hacerlo igualmente? Una lámpara encendida no impide pulsar el botón, así que el estado de la lámpara va en el efecto. Una puerta cerrada sí impide atravesarla, así que el estado de la puerta va en la precondición.

	En la precondición	En un `when`
Decide	si la acción se aplica	qué hechos cambia la acción
Si es falsa	la acción no se puede usar	la acción se usa, sin ese efecto
Se evalúa en	el estado anterior	el estado anterior
El planificador la usa	para podar acciones	para predecir el resultado

Tabla 6.1.: La misma condición en los dos sitios de una acción donde se puede escribir.

Las dos acciones `switch-on` y `switch-off` se conservan en el fichero `lights-strips-domain.pddl`, que resuelve el mismo problema con el mismo número de pasos y que, para una lámpara, es un modelo perfectamente válido: dos casos, dos acciones. La diferencia aparece cuando los casos se combinan. Un botón que cambia tres lámparas tiene 2^3 casos, uno por cada combinación de lámparas encendidas, y necesitaría ocho acciones STRIPS. Con efectos condicionales sigue siendo una sola acción, como mostrará la Sección 6.4.

6.3. Efectos universales: `forall`

El pasillo tiene también un interruptor general que apaga todas las lámparas. ¿Cuántas lámparas hay? El dominio no lo sabe: tres en un problema, seis en el siguiente. La acción `all-off` del Listado 6.2 lo dice sin contar:

```
18  ; The master switch: every lamp goes off, whatever it was.
19  (:action all-off
20    :parameters ()
21    :effect (forall (?l - lamp) (not (lit ?l)))))
```

Un **efecto universal** tiene la forma

```
(forall (<typed variables>) <effect>)
```

y aplica el efecto una vez por cada combinación de objetos que pueden tomar las variables. El efecto interior puede mencionar las variables nuevas y también los parámetros de la acción. Un efecto universal es la imagen especular de la precondición universal del capítulo anterior: aquella le *preguntaba* algo a cada objeto; este le *hace* algo a cada objeto.

No hace falta ningún requisito nuevo. En la gramática de PDDL, `forall` en un efecto pertenece a `:conditional-effects`, como `when`. Kairos lo confirma: al comprobar un dominio cuyo único efecto rico es un `forall`, informa de `conditional-effects` entre las características en uso.

El requisito `:universal-preconditions` se refiere a `forall` en condiciones y objetivos. Ningún requisito se llama «efectos universales».

El conjunto de objetos es finito y se conoce al leer el problema, así que un efecto universal es una abreviatura. Con las lámparas `l1` a `l6`, `all-off` es una acción instanciada con seis efectos de borrado. No se decide nada al planificar; el planificador expande el `forall` cuando instancia la

acción. Lo que gana el modelador es un dominio que sirve para cualquier problema.

El segundo problema tiene seis lámparas, cinco de ellas encendidas, y quiere encendida solo `l2`:

```
$ kairos lights-domain.pddl lights-p02.pddl
; cost = 6
(toggle l1)
(toggle l2)
(toggle l3)
(toggle l4)
(toggle l5)
(toggle l6)
; better plan 2
; cost = 2
(all-off)
(toggle l2)
```

El primer plan pulsa los seis botones: cinco lámparas se apagan y `l2` se enciende. La búsqueda continúa y encuentra el plan que elegiría una persona: el interruptor general y después un botón. Observa que `all-off` borra `(lit l2)`, que ya era falso. Borrar un hecho que no está no es un error. Simplemente no cambia nada, y eso es lo que permite escribir un efecto universal sin conocer el estado.

Un forall en un efecto lo aplica a cada objeto del tipo. Se expande cuando se instancia la acción y necesita :conditional-effects.

Un `forall` incondicional es una herramienta tosca: *todas* las lámparas, *todas* las puertas. Encaja en efectos que reinician algo (un interruptor general, una alarma que abre todas las puertas, un corte de luz). Usaremos uno en el carro: cuando el robot coge un artículo, el artículo deja de estar en cualquier carro, fuera cual fuera el carro en el que estaba, y `(forall (?c - cart) (not (on-cart ?i ?c)))` lo dice sin preguntar cuál. Pero la mayoría de las veces el efecto va dirigido a *algunos* de los objetos, y el `forall` lleva un `when` dentro.

6.4. Los dos juntos: para todo objeto tal que

Las dos construcciones se combinan en la forma que las hace útiles:

```
(forall (?x - type)
  (when <condition on ?x>
    <effect on ?x>))
```

Se lee «para todo `?x` tal que se cumple la condición, el efecto ocurre». El `forall` hace entrar a todos los objetos; el `when` se queda con los afectados. Es la frase que la Sección 6.1 no podía escribir. La veremos tres veces: en un rompecabezas, en un clásico y en el carro.

Lights Out

En el rompecabezas Lights Out cada botón está conectado a varias lámparas y las cambia todas a la vez. El objetivo es apagar todas las lámparas. Qué lámparas cambia un botón es parte del tablero, así que es un dato del problema: un predicado estático `(wired ?b ?l)`.

```
;; Lights Out: each button changes the lamps wired to it.
;; A universal effect with conditions: every lamp, when wired.
(define (domain lightsout)
  (:requirements :strips :typing :negative-preconditions
                 :universal-preconditions :conditional-effects)

  (:types button lamp)

  (:predicates
    (wired ?b - button ?l - lamp)   ; static: ?b changes ?l
    (lit ?l - lamp))

  (:action press
    :parameters (?b - button)
    :effect (forall (?l - lamp)
              (and (when (and (wired ?b ?l) (lit ?l))
                     (not (lit ?l)))
                   (when (and (wired ?b ?l) (not (lit ?l)))
                     (lit ?l))))))
```

Listado 6.3: Lights Out: para cada lámpara conectada al botón, la encendida se apaga y la apagada se enciende (`lightsout-domain.pddl`).

La acción `press` tiene un parámetro y cambia cualquier número de lámparas. Dentro del `forall`, los dos `when` son la conmutación de la Sección 6.2 con un término más en la conjunción, `(wired ?b ?l)`. Una lámpara que no está conectada al botón no cumple ninguna de las dos condiciones y se queda como está. Es la regla del marco del Capítulo 2 actuando dentro de un efecto condicional: lo que ningún efecto menciona no cambia.

El dominio declara `:universal-preconditions` por el objetivo de sus problemas (todas las lámparas apagadas), no por el efecto.

El primer problema es una fila de cinco lámparas. Cada botón cambia su propia lámpara y las vecinas. Están encendidas las lámparas 1, 3 y 4 (Figura 6.3).

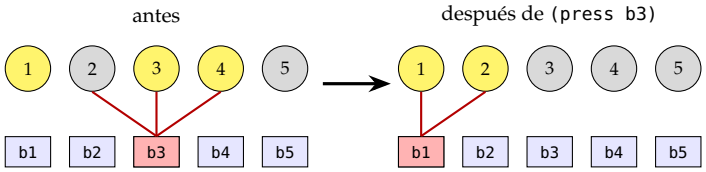


Figura 6.3.: Lights Out en una fila de cinco lámparas (en amarillo, las encendidas). A la izquierda, el estado inicial; las líneas rojas son las lámparas conectadas a `b3`. Pulsarlo cambia las lámparas 2, 3 y 4 y deja encendidas las lámparas 1 y 2, que `b1` (a la derecha) apaga.

```
$ kairos lightsout-domain.pddl lightsout-p01.pddl
; cost = 4
(press b2)
(press b3)
(press b2)
(press b1)
; better plan 2
; cost = 2
(press b3)
(press b1)
```

El primer plan pulsa `b2` dos veces, lo que se anula a sí mismo; el segundo es el de la figura. El segundo problema difiere en una lámpara del estado inicial (encendidas las lámparas 1, 2 y 4) y no tiene solución:

```
$ kairos lightsout-domain.pddl lightsout-p02.pddl
note: no plan: every state reachable from the initial one was
     searched (16 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

Cinco lámparas tienen $2^5 = 32$ estados, y el planificador visitó 16. En este tablero, los botones alcanzan desde cualquier estado solo la mitad de los estados, y «todas apagadas» está en la otra mitad. El planificador no conoce el álgebra que hay detrás de ese hecho. Recorrió todos los estados alcanzables, y su respuesta es una demostración.

El maletín

El ejemplo más antiguo de estos efectos es un maletín. Se pueden meter cosas en él y sacarlas, y cuando se lleva el maletín a otro lugar, lo que hay dentro va con él.

```
;; The briefcase: what is inside moves with it.
(define (domain briefcase)
  (:requirements :strips :typing :negative-preconditions
                 :conditional-effects)

  (:types place thing)

  (:predicates
    (case-at ?p - place)            ; where the briefcase is
    (at ?t - thing ?p - place)      ; where a thing is
    (in ?t - thing))                ; ?t is in the briefcase

  ; Carry the briefcase: every thing in it changes place too.
  (:action carry
    :parameters (?from ?to - place)
    :precondition (case-at ?from)
    :effect (and (not (case-at ?from)) (case-at ?to)
                 (forall (?t - thing)
                   (when (in ?t)
                     (and (not (at ?t ?from)) (at ?t ?to))))))

  (:action put-in
    :parameters (?t - thing ?p - place)
    :precondition (and (case-at ?p) (at ?t ?p)
                       (not (in ?t)))
    :effect (in ?t))

  (:action take-out
    :parameters (?t - thing)
    :precondition (in ?t)
    :effect (not (in ?t))))
```

Listado 6.4: El maletín: llevarlo mueve todas las cosas que hay dentro (`briefcase-domain.pddl`).

Una cosa que está en el maletín conserva su hecho `at`, y `carry` lo mantiene al día: las líneas 18 a 20 mueven todas las cosas que están `in`. En el problema, el maletín está en casa con las llaves dentro. El informe, también en casa, tiene que llegar a la oficina, y las llaves tienen que quedarse en casa.

```
$ kairos briefcase-domain.pddl briefcase-p01.pddl
; cost = 3
(put-in report home)
(take-out keys)
(carry home office)
```

Ningún objetivo pide sacar las llaves. El planificador lo hace porque ha entendido el efecto universal: llevar el maletín con las llaves dentro haría falso `(at keys home)`. El segundo paso está ahí para *evitar* un efecto. Veremos más casos así en la Sección 6.5.

ADL y el maletín

En 1989 Edwin Pednault presentó ADL, el *Action Description Language*, en la primera conferencia sobre *Principles of Knowledge Representation and Reasoning* (Pednault, 1989). STRIPS facilitaba planificar y se quedaba corto para modelar; el cálculo de situaciones de la lógica era lo contrario. ADL tomó una posición intermedia: las acciones conservan su forma de adiciones y borrados, pero las condiciones pueden ser cualquier fórmula de primer orden y los efectos pueden ser condicionales y cuantificados. El ejemplo fue un maletín cuyo contenido se mueve con él. Cuando se definió PDDL para la primera competición de planificación, en 1998 (McDermott et al., 1998), adoptó estas características, cada una con su propio indicador de requisito, y un indicador para todas ellas: `:adl`.

El carro

Ya podemos escribir la frase que necesitaba el carro. Primero el método: lo que el carro añade a las tablas del mundo del mensajero.

Pasos O y D: el carro

Ficha del modelo, lo que cambia: la Historia gana un carro que un robot carga y empuja; el Alcance incluye ahora dónde está el carro y qué hay en él, y sigue dejando fuera cuánto cabe en él (cualquier número de artículos). Supuesto: un robot que empuja un carro no tiene ninguna mano libre.

Tabla de estado, filas nuevas:

Propiedad	De qué	Clase
`(cart-at ?c ?l)`	un carro: la habitación donde está	dinámico
`(on-cart ?i ?c)`	un artículo y un carro: el artículo está encima	dinámico
`(pushing ?r ?c)`	un robot y un carro: el robot sujeta su asa	dinámico

Tabla de acciones, filas nuevas y modificadas. La última columna es nueva: lo que la acción cambia *según el estado*.

Acción	Siempre cambia	También cambia
`pick`	como antes	el artículo deja *todos* los carros
`load`	la mano queda libre, el artículo está en el carro	
`grab-cart`	el robot sujeta el carro, mano ocupada	
`release-cart`	lo contrario	
`push-cart`	robot y carro cambian de habitación	*todo* artículo, *si* está en el carro, cambia de habitación

El Listado 6.5 es el dominio. Parte del dominio tipado del mensajero del Capítulo 4 y conserva sus nombres.

```
;; The courier world with a cart: pushing the cart moves
;; everything that is on it.
(define (domain courier-cart)
  (:requirements :strips :typing :negative-preconditions
                 :existential-preconditions :conditional-effects)

  (:types robot location item cart)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
```

```
14      (free ?r - robot)
15      (cart-at ?c - cart ?l - location)
16      (on-cart ?i - item ?c - cart)     ; ?i lies on the cart
17      (pushing ?r - robot ?c - cart))   ; ?r holds the handle
18
19    ; A robot that holds a cart cannot walk away from it.
20    (:action move
21      :parameters (?r - robot ?from ?to - location)
22      :precondition (and (robot-at ?r ?from) (connected ?from ?to)
23                         (not (exists (?c - cart) (pushing ?r ?c))))
24      :effect (and (not (robot-at ?r ?from))
25                   (robot-at ?r ?to)))
26
27    ; Take an item from the floor or from a cart: whatever cart
28    ; it was on, it is on none afterwards.
29    (:action pick
30      :parameters (?r - robot ?i - item ?l - location)
31      :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
32      :effect (and (not (item-at ?i ?l)) (not (free ?r))
33                   (carrying ?r ?i)
34                   (forall (?c - cart) (not (on-cart ?i ?c)))))
35
36    (:action drop
37      :parameters (?r - robot ?i - item ?l - location)
38      :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
39      :effect (and (not (carrying ?r ?i))
40                   (item-at ?i ?l) (free ?r)))
41
42    ; Put the carried item on a cart that is in the same room.
43    (:action load
44      :parameters (?r - robot ?i - item ?c - cart ?l - location)
45      :precondition (and (robot-at ?r ?l) (cart-at ?c ?l)
46                         (carrying ?r ?i))
47      :effect (and (not (carrying ?r ?i)) (free ?r)
48                   (item-at ?i ?l) (on-cart ?i ?c)))
49
50    (:action grab-cart
51      :parameters (?r - robot ?c - cart ?l - location)
52      :precondition (and (robot-at ?r ?l) (cart-at ?c ?l) (free ?r))
53      :effect (and (not (free ?r)) (pushing ?r ?c)))
54
55    (:action release-cart
56      :parameters (?r - robot ?c - cart)
57      :precondition (pushing ?r ?c)
58      :effect (and (not (pushing ?r ?c)) (free ?r)))
59
60    ; The robot, the cart and every item on the cart change room.
61    (:action push-cart
62      :parameters (?r - robot ?c - cart ?from ?to - location)
63      :precondition (and (pushing ?r ?c) (robot-at ?r ?from)
64                         (connected ?from ?to))
65      :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
66                   (not (cart-at ?c ?from)) (cart-at ?c ?to)
67                   (forall (?i - item)
68                     (when (on-cart ?i ?c)
69                       (and (not (item-at ?i ?from))
70                            (item-at ?i ?to)))))))
```

Listado 6.5: El mundo del mensajero con un carro (`courier-cart-domain.pddl`).

Cuatro puntos del listado merecen un comentario.

- `push-cart` (líneas 61 a 70) es la frase de la Sección 6.1, casi palabra por palabra. Su efecto condicional universal tiene la misma forma que el del maletín. Con el carro vacío no se cumple ninguna condición y solo se mueven el robot y el carro.
- El efecto condicional contiene un borrado *y* una adición. Un artículo que se mueve debe salir de la habitación antigua además de entrar en la nueva, exactamente igual que en un `move` normal.
- `pick` (línea 34) coge un artículo del suelo o de un carro con la misma acción. El `forall` incondicional borra `on-cart` para todos

los carros, lo que no cambia nada si el artículo no estaba en ninguno.
- `move` (línea 23) prohíbe marcharse sujetando el asa, con una precondición existencial negada. `grab-cart` quita `(free ?r)`, así que un robot que empuja tampoco puede coger nada.

El primer problema es el de la Figura 6.1, salvo que las cajas empiezan en el suelo de la farmacia: el robot está en el base de carga, y el objetivo es `(and (item-at meds1 icu) (item-at meds2 icu))`.

```
$ kairos courier-cart-domain.pddl courier-cart-p01.pddl
; cost = 18
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
...
(drop r1 meds2 icu)
; better plan 2
; cost = 17
...
; better plan 3
; cost = 11
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(load r1 meds1 cart1 pharmacy)
(pick r1 meds2 pharmacy)
(load r1 meds2 cart1 pharmacy)
(grab-cart r1 cart1 pharmacy)
(push-cart r1 cart1 pharmacy lobby)
(push-cart r1 cart1 lobby corridor)
(push-cart r1 cart1 corridor ward-b)
(push-cart r1 cart1 ward-b icu)
```

El primer plan ignora el carro y lleva las cajas de una en una: 18 pasos, de los que 12 son desplazamientos a pie. El último, que Kairos demuestra óptimo, carga las dos cajas y empuja cuatro veces. El objetivo se cumple en el cuarto empujón, como efecto de una acción cuyos parámetros no mencionan ninguna caja. El plan termina ahí, con las cajas en el carro en la UCI: el objetivo preguntaba dónde están las cajas, no que estuvieran en el suelo.

El segundo problema tiene tres artículos y añade un objetivo: el carro debe volver a la farmacia. El mejor plan encontrado, de 25 pasos, termina así:

```
$ kairos courier-cart-domain.pddl courier-cart-p02.pddl
...
(push-cart r1 cart1 ward-b icu)
(release-cart r1 cart1)
(pick r1 meal1 icu)
(drop r1 meal1 icu)
(pick r1 meds2 icu)
(drop r1 meds2 icu)
(pick r1 meds1 icu)
(drop r1 meds1 icu)
(grab-cart r1 cart1 icu)
(push-cart r1 cart1 icu ward-b)
(push-cart r1 cart1 ward-b corridor)
(push-cart r1 cart1 corridor lobby)
(push-cart r1 cart1 lobby pharmacy)
```

Antes de que el carro vuelva, el robot suelta el asa, quita cada artículo del carro y lo deja en el suelo. Nada en el objetivo menciona `on-cart`. El planificador descarga porque lo que se quede en el carro volvería con él

a la farmacia. Como con las llaves del maletín, parte del plan existe solo por lo que, de otro modo, haría un efecto universal.

El forall hace entrar a todos los objetos y el when se queda con los afectados. Así, una acción cambia tantos hechos como exige el estado, con parámetros que no los nombran.

6.5. Efectos que hay que evitar: el laboratorio estéril

Hasta ahora, un efecto condicional era algo que el plan buscaba (las cajas llegan) o que tenía que esquivar (las llaves se quedan). El laboratorio trae un efecto condicional que es claramente malo. El laboratorio es una sala estéril. Su puerta es una esclusa que rocía con desinfectante todo lo que entra. Los medicamentos y las comidas pasan sin daño. Una muestra en una bolsa sellada, también. Una muestra que no está sellada se estropea con el rociado: queda contaminada e inservible para el análisis. El robot puede sellar en una bolsa la muestra que lleva, en cualquier momento.

En la ficha del modelo, el Alcance gana dos propiedades de las muestras y una de las habitaciones, con los nombres del vocabulario común: `(sealed ?i)`, `(contaminated ?i)` y el estático `(sterile ?l)`. Se añaden dos supuestos: un sello no se vuelve a abrir nunca, y una muestra contaminada sigue contaminada. Los artículos tienen ahora subtipos, ya que solo las muestras se pueden sellar o contaminar. La acción nueva y la modificada están en el Listado 6.6.

```
21  ; The move is always possible. What it changes, besides the
22  ; position, depends on the room entered and on what is carried.
23  (:action move
24    :parameters (?r - robot ?from ?to - location)
25    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
26    :effect (and (not (robot-at ?r ?from))
27                 (robot-at ?r ?to)
28                 (forall (?i - sample)
29                   (when (and (sterile ?to) (carrying ?r ?i)
30                              (not (sealed ?i)))
31                     (contaminated ?i)))))
32  ; Put the carried sample in a sealed bag.
33  (:action seal
34    :parameters (?r - robot ?i - sample)
35    :precondition (and (carrying ?r ?i) (not (sealed ?i)))
36    :effect (sealed ?i)))
```

Listado 6.6: El movimiento a través de una esclusa y el sellado (de `courier-sterile-domain.pddl`; `pick` y `drop` no cambian).

La esclusa no detiene a nadie, así que no se ha añadido nada a la precondición de `move`. Lo que entrar le hace a una muestra está en el efecto, y la Figura 6.4 despliega su condición. El `forall` está ahí por una razón de sintaxis: `move` no tiene un parámetro para el artículo, y no puede tenerlo, porque el robot puede no llevar nada. «Toda muestra que lleva el robot» cubre los casos de ninguna y de una con el mismo texto.

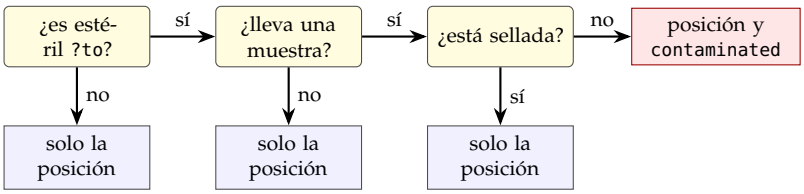


Figura 6.4.: Lo que cambia `(move ?r ?from ?to)`. La acción es aplicable en los cuatro casos; tres condiciones, todas leídas en el estado anterior al movimiento, seleccionan aquel en el que la muestra queda además contaminada.

Ensayar el modelo

Un efecto que depende del estado necesita una batería de pruebas que recorra sus casos. El paso E del método MODEL pide cuatro problemas, y

aquí cada uno tiene un cometido preciso.

Paso E: una batería de pruebas para un efecto condicional

Trivial (`p01`): una caja de medicamentos al laboratorio. La condición del efecto nunca es cierta; el plan debe ser el del antiguo mundo del mensajero.

Típica (`p02`): la muestra de la sala A al laboratorio, con el objetivo `(not (contaminated sample1))`. El plan debe sellarla antes de la esclusa.

Sin plan (`p03`): el laboratorio quiere la muestra abierta *y* utilizable. Es imposible en este mundo, así que el planificador debe decirlo.

Trampa (`p04`): el objetivo solo pide la muestra en el laboratorio. Esperamos que el planificador tome el atajo, y queremos verlo.

El problema trivial da un plan de siete pasos, como la entrega del Capítulo 4. El típico:

```
$ kairos courier-sterile-domain.pddl courier-sterile-p02.pddl
; cost = 8
(move r1 dock lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(pick r1 sample1 ward-a)
(move r1 ward-a corridor)
(seal r1 sample1)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

El robot sella la muestra en el pasillo, un paso antes de la esclusa. Valdría cualquier momento posterior a cogerla; lo que importa es que `(sealed sample1)` sea cierto en el estado en el que se aplica `(move r1 corridor lab)`. El tercer problema termina con `kairos: unsolvable (exhausted)` tras 71 estados, como debe. El cuarto:

```
$ kairos courier-sterile-domain.pddl courier-sterile-p04.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(pick r1 sample1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

Un paso más corto, y la muestra llega contaminada. El dominio no tiene ningún error: describe lo que hace la esclusa, y un robot puede, en efecto, entrar en ella con una muestra abierta. El problema es el objetivo, que olvidó decir lo que importa. Un planificador no evita un efecto malo porque sea malo. Lo evita cuando un objetivo (o una precondición posterior) necesita lo contrario. El validador hace visible la diferencia: el plan anterior, guardado en un fichero, es válido para `p04` y se rechaza para `p02`.

```
$ kairos validate courier-sterile-domain.pddl courier-sterile-p02.
    pddl courier-sterile-noseal.plan
courier-sterile-noseal.plan: error: the goal does not hold at the
    end of the plan: (contaminated sample1) is true [validate.
```

```
    goal_not_reached]
plan invalid
```

Un efecto condicional puede ser un peligro. El planificador lo evita solo si el objetivo o una precondición posterior necesitan lo que destruye: todo daño que importe debe nombrarse en el objetivo.

Una acción por caso

El movimiento a través de la esclusa tiene pocos casos, así que aquí es razonable la alternativa de la Sección 6.1. Es la que sigue el fichero `courier-sterile-cases-domain.pddl`, que divide `move` en tres:

```
; Case 1: the room entered has no airlock.
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (not (sterile ?to)))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))

; Case 2: airlock, and no unsealed sample is carried.
(:action enter-clean
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (sterile ?to)
                     (forall (?i - sample)
                       (imply (carrying ?r ?i) (sealed ?i))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))

; Case 3: airlock, and the unsealed sample ?i is carried.
(:action enter-exposed
  :parameters (?r - robot ?i - sample ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (sterile ?to)
                     (carrying ?r ?i) (not (sealed ?i)))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (contaminated ?i)))
```

Listado 6.7: La esclusa sin efectos condicionales: tres acciones para los tres casos de un movimiento (de `courier-sterile-cases-domain.pddl`).

Las precondiciones de las tres acciones se excluyen entre sí y cubren todos los estados, así que exactamente una de ellas se aplica allí donde se aplicaba el antiguo `move`. El plan del problema típico es el mismo, con `(enter-clean r1 corridor lab)` en lugar del movimiento hacia el laboratorio. El plan dice ahora en su propio texto que la entrada fue limpia, lo que se lee mejor que un `move` que puede haber contaminado algo o no. Los costes son la repetición (el efecto sobre la posición se escribe tres veces) y lo que ocurre cuando la regla crece. Un segundo factor independiente, por ejemplo un robot que tuviera que desinfectarse a sí mismo salvo que lleve una funda, duplicaría los casos. Con efectos condicionales sería un `when` más.

Cabe una tercera lectura de la historia: una esclusa que se *niega* a abrirse ante una muestra sin sellar. Eso es una precondición, y un mundo distinto: en él ningún plan contamina nunca nada, diga lo que diga el objetivo. Cuál de los dos es el correcto es una pregunta para el paso M, sobre la esclusa real. El Ejercicio 6.5.2 construye ese modelo y compara las baterías de pruebas.

6.6. El requisito `:adl`

Con los efectos condicionales hemos visto todos los requisitos del PDDL clásico. Como se suelen usar juntos, PDDL tiene una abreviatura: `:adl` equivale a los siete requisitos de la Tabla 6.2.

Requisito	Qué permite	Capítulo
`:strips`	efectos de adición y de borrado	Capítulo 2
`:typing`	tipos de objetos y de parámetros	Capítulo 4
`:equality`	= entre objetos	Capítulo 4
`:negative-preconditions`	`not` en condiciones y objetivos	Capítulo 5
`:disjunctive-preconditions`	`or` e `imply`	Capítulo 5
`:quantified-preconditions`	`exists` y `forall` en condiciones	Capítulo 5
`:conditional-effects`	`when` y `forall` en efectos	este

Tabla 6.2.: Los siete requisitos que abrevia `:adl`.

Un dominio puede escribir `(:requirements :adl)` y usar cualquiera de ellos. Este libro sigue enumerando los requisitos que cada fichero usa de verdad. La lista es un resumen del fichero para el lector, y es lo primero que hay que mirar cuando un planificador rechaza un dominio: un planificador que solo admite algunos de los siete puede aceptar un dominio que declara lo que usa y tiene que rechazar uno que declara `:adl` para usar una sola negación.

Aquí termina la planificación clásica

Los tipos, la igualdad, la negación, la disyunción, los cuantificadores y los efectos condicionales no cambian lo que *es* un problema de planificación. Los estados siguen siendo conjuntos de hechos, las acciones siguen llevando de un estado a otro y un plan sigue siendo una secuencia de acciones. Todo modelo ADL con un conjunto fijo de objetos se puede reescribir como un modelo STRIPS que tiene los mismos planes, o planes que se corresponden uno a uno (la Sección 6.7 muestra cómo para los efectos). ADL es un lenguaje mejor para la misma clase de problemas. Las partes del libro que vienen después son distintas: los números, el tiempo y las preferencias cambian el propio problema.

6.7. Qué hacen los planificadores con los efectos condicionales

El Capítulo 3 explicó que un planificador instancia el modelo antes de buscar. Los efectos ricos se tratan en esa fase, en hasta tres pasos. Conocerlos le dice al modelador cuánto cuesta cada construcción.

Los efectos universales se expanden. Un `forall` se convierte en un efecto por cada objeto. Las condiciones sobre predicados estáticos se deciden en este momento y desaparecen. En Lights Out, la acción `(press b3)` empieza con dos efectos condicionales por cada una de las cinco lámparas; `(wired b3 l1)` es falso para siempre, así que los efectos de `l1` y `l5` se descartan, y la acción instanciada se queda con seis: dos por cada una de las tres lámparas conectadas al botón. Los datos estáticos en la condición de un efecto no cuestan nada durante la búsqueda, igual que en una precondición.

Los efectos condicionales se conservan o se eliminan compilándolos. Un planificador cuya búsqueda trabaja con efectos condicionales los conserva dentro de la acción instanciada. Un planificador que solo conoce acciones STRIPS tiene que reescribir antes el modelo, y hay dos formas de hacerlo (Figura 6.5).

- **Una acción por combinación.** Una acción con k efectos condicionales se sustituye por 2^k acciones, una por cada elección de qué condiciones se cumplen, con esas elecciones añadidas a la precondición. Los planes conservan su longitud. El modelo puede crecer exponencialmente: `(press b3)`, con tres lámparas, se convierte en ocho acciones.
- **Acciones auxiliares.** La acción se corta en una secuencia: primero la parte incondicional, después un pequeño paso por cada efecto condicional, que lo aplica o lo omite, y por último un paso que cierra la secuencia. Hechos adicionales obligan a dar los pasos en orden, y hay que recordar las condiciones para que se sigan leyendo tal como eran antes de la acción. El modelo crece poco; los planes se alargan.

No hay una tercera vía que evite los dos costes. Bernhard Nebel demostró en 2000 Nebel, 2000 que los efectos condicionales no se pueden compilar a STRIPS puro conservando a la vez un modelo pequeño y unos planes esencialmente igual de cortos. Por eso los efectos condicionales cuentan como una ganancia real de poder expresivo, y no como una comodidad de notación.

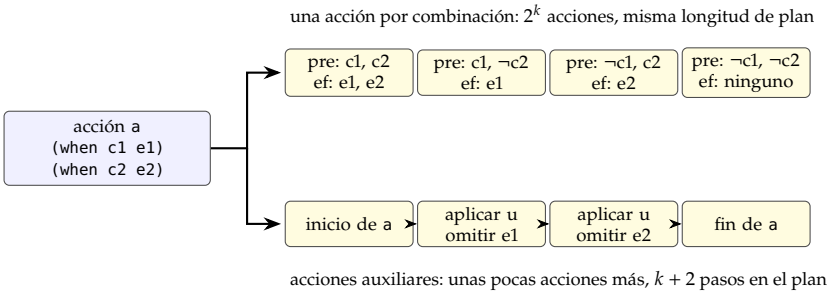


Figura 6.5.: Dos formas de eliminar los efectos condicionales de una acción. Arriba crece el modelo; abajo, los planes.

La heurística mira cada efecto por separado. La relajación del Capítulo 3 ignora los borrados. Con efectos condicionales, los planificadores de la tradición de FF tratan cada efecto condicional como una pequeña acción cuya precondición es la precondición de la acción más su propia condición (Hoffmann & Nebel, 2001). La estimación puede contar varias veces una misma acción real, una por cada uno de sus efectos que necesita, pero sigue siendo barata de calcular.

Kairos conserva los efectos condicionales en sus acciones instanciadas. Los dos modelos de las lámparas lo muestran:

```
$ kairos lights-domain.pddl lights-p01.pddl -v 2>&1 | grep ground
[   0.001s] [ground] grounded in 0.1 ms: 3 atoms, 0 numeric fluents,
      4 actions, 0 durative actions, 0 axioms (10 Datalog facts, 2
     rules, 0 instances pruned)
$ kairos lights-strips-domain.pddl lights-strips-p01.pddl -v 2>&1 |
     grep ground
[   0.000s] [ground] grounded in 0.1 ms: 3 atoms, 0 numeric fluents,
      6 actions, 0 durative actions, 0 axioms (12 Datalog facts, 3
     rules, 0 instances pruned)
```

Tres lámparas dan cuatro acciones instanciadas con efectos condicionales (tres `toggle` y `all-off`) y seis sin ellos (tres `switch-on` y tres `switch-off`). La conmutación no se ha dividido.

No todos los planificadores llegan tan lejos, y este es el punto débil de la característica. En las pruebas que hay detrás de la tabla de compatibilidad de la documentación de Kairos, con un dominio pequeño por característica, solo Kairos, ENHSP y LAMA devolvieron un plan válido para el dominio con efectos condicionales; OPTIC, POPF, TFD, LPG-td y Metric-FF no. Un modelo que tenga que funcionar en varios planificadores, en particular en los temporales, paga por cada `when`.

Los efectos universales se expanden en la instanciación. Los efectos condicionales los conservan los planificadores que los admiten; eliminarlos compilándolos cuesta o un número exponencial de acciones o planes más largos.

6.8. El método MODEL: efectos que dependen de la situación

El paso D del método describe la dinámica: para cada acción, cuándo se aplica y qué cambia. Con efectos STRIPS, la segunda parte era una lista. Este capítulo le añade dos preguntas.

Paso D: dos preguntas nuevas

Para cada acción de la tabla de acciones:

1. **¿Depende de la situación lo que cambia la acción?** Descríbete la acción con palabras. Si la descripción tiene un «si» o un «salvo que» que no impide actuar al agente (*siempre puede pulsar, entrar, empujar; lo que ocurre depende*), la acción tiene un efecto condicional. Si el «si» sí impide actuar al agente, es una precondición.
2. **¿Cambia muchos objetos a la vez?** Si la descripción dice «cada», «todos» o «lo que esté dentro, encima o cerca», la acción tiene un efecto universal, normalmente con una condición que selecciona los objetos.

Escribe las respuestas en una tercera columna de la tabla de acciones: *siempre cambia* y *también cambia, si*.

Y para el paso E: para cada efecto condicional, un problema de prueba en el que su condición se cumple, otro en el que no, y una prueba de la trampa cuyo objetivo se pueda alcanzar más barato desencadenando (o no desencadenando) el efecto.

Cuando la respuesta a una de las preguntas es sí, un efecto condicional es la codificación directa. No es la única, y el paso L pide mirar las demás.

Alternativas comparadas

Comparamos tres codificaciones del carro. Las tres mantienen la decisión de que un artículo que está en el carro tiene un hecho `item-at`, y se diferencian en cómo lo mantiene al día un empujón.

A. Efecto condicional. El dominio del Listado 6.5: `push-cart` con un `forall` y un `when`.

B. Una acción por caso. El dominio del Listado 6.1, con sus cuatro acciones de empujar, cada una de las cuales nombra los artículos que se mueven.

C. Una acción auxiliar. El empujón mueve solo el robot y el carro. Los artículos lo siguen de uno en uno con una acción adicional, y una tercera acción cierra el movimiento (Listado 6.8). Es la segunda compilación de la Figura 6.5, hecha a mano.

```
; Step 1: the robot and the cart change room; the items wait.
(:action push-cart
  :parameters (?r - robot ?c - cart ?from ?to - location)
  :precondition (and (not (unsettled))
                     (pushing ?r ?c) (robot-at ?r ?from)
                     (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (cart-at ?c ?from)) (cart-at ?c ?to)
               (rolled ?c ?from ?to) (unsettled)))

; Step 2, once for each item on the cart: it follows the cart.
(:action bring-along
  :parameters (?i - item ?c - cart ?from ?to - location)
  :precondition (and (rolled ?c ?from ?to) (on-cart ?i ?c)
                     (item-at ?i ?from))
  :effect (and (not (item-at ?i ?from)) (item-at ?i ?to)))

; Step 3: when no item is left behind, normal life goes on.
(:action settle
  :parameters (?c - cart ?from ?to - location)
  :precondition (and (rolled ?c ?from ?to)
                     (forall (?i - item)
                       (imply (on-cart ?i ?c)
                              (item-at ?i ?to))))
  :effect (and (not (rolled ?c ?from ?to))
               (not (unsettled)))))
```

Listado 6.8: Codificación C: el empujón en tres clases de pasos (de `courier-cart-settle-domain.pddl`). Todas las demás acciones del dominio tienen la precondición `(not (unsettled))`.

En la codificación C, `push-cart` levanta un indicador, `(unsettled)`, que bloquea todas las acciones normales. Mientras está levantado, solo se aplican `bring-along` y `settle`; `settle` exige que ningún artículo del carro siga en la habitación antigua, y baja el indicador. Un empujón con dos cajas en el carro ocupa cuatro pasos del plan:

```
$ kairos courier-cart-settle-domain.pddl courier-cart-settle-p04.
    pddl
; cost = 11
...
(grab-cart r1 cart1 pharmacy)
(push-cart r1 cart1 pharmacy lobby)
(bring-along meds2 cart1 pharmacy lobby)
(bring-along meds1 cart1 pharmacy lobby)
(settle cart1 pharmacy lobby)
```

La Tabla 6.3 compara las tres según los cinco criterios, con las cifras que da `kairos -v` para el problema de tres artículos y un carro que debe volver (`p02`).

	A. Efecto condicional	B. Acción por caso	C. Acción auxiliar
Fidelidad	exacta	un carro con cuatro artículos no se puede mover	exacta solo si todo objetivo pide también `(not (unsettled))`
Tamaño	109 acciones instanciadas	343 acciones instanciadas; crece como n^k	189 acciones instanciadas
Guía	mejor plan: 25 pasos, con el carro	25 pasos, con el carro	un empujón cuesta de 2 a 5 pasos: el mejor plan (28 pasos) evita el carro
Claridad	una acción, una frase	cuatro acciones que dicen lo mismo	tres acciones y un indicador en cada precondición
Portabilidad	necesita `:conditional-effects`	necesita precondiciones cuantificadas	necesita precondiciones cuantificadas

Tabla 6.3.: Tres codificaciones del empujón de un carro, comparadas en el problema `p02` (tres artículos, el carro vuelve).

Dos filas necesitan un comentario. En Guía, la codificación C cambia lo que significa la longitud de un plan. Sus pasos ya no son cosas que hace el robot, y un planificador que busca planes cortos ve el carro como caro: para `p02` prefiere llevar los tres artículos a mano. El Capítulo 9 da el remedio, un coste cero para las acciones auxiliares. En Fidelidad, el indicador abre un agujero. En `courier-cart-settle-p03` el objetivo es el carro cargado en el vestíbulo, y el plan se detiene justo después del empujón, con el carro en el vestíbulo y sus cajas, según `item-at`, todavía en la farmacia. Un modelo con acciones auxiliares tiene estados que no significan nada en el mundo, y el objetivo debe excluirlos.

En Portabilidad, ni B ni C son STRIPS puro tal como están escritos aquí: los dos usan una precondición universal para decir «no hay nada más en el carro». Se puede eliminar, contando los artículos del carro con las técnicas del Capítulo 7, y alarga los dos dominios.

Una cuarta vía: no almacenar lo que se puede deducir

Las tres codificaciones se esfuerzan por mantener al día un hecho redundante. La posición de un artículo que está en el carro ya se conoce: es donde está el carro. Si `load` borra `item-at` y el artículo solo recupera una posición cuando un robot lo quita del carro y lo deja en el suelo, el empujón no tiene nada que actualizar:

```
; On the cart the item loses its own position...
(:action load
  :parameters (?r - robot ?i - item ?c - cart ?l - location)
  :precondition (and (robot-at ?r ?l) (cart-at ?c ?l)
                     (carrying ?r ?i))
  :effect (and (not (carrying ?r ?i)) (free ?r)
               (on-cart ?i ?c)))

; ...and gets none back until a robot drops it somewhere.
(:action unload
  :parameters (?r - robot ?i - item ?c - cart ?l - location)
  :precondition (and (robot-at ?r ?l) (cart-at ?c ?l)
                     (on-cart ?i ?c) (free ?r))
  :effect (and (not (on-cart ?i ?c)) (not (free ?r))
               (carrying ?r ?i)))
```

Listado 6.9: Codificación D: un artículo en el carro no tiene posición propia (de `courier-cart-implied-domain.pddl`). El `push-cart` de este dominio es la acción STRIPS simple que mueve el robot y el carro.

Así ha tratado el mundo del mensajero, desde el principio, un artículo transportado: `pick` borra `item-at`, y nadie lo echó de menos. En `p02` la codificación tiene 133 acciones instanciadas y el mismo mejor plan de 25 pasos. Sus estados son además más sencillos para el planificador. Con

`-v`, Kairos informa de 13 variables de estado para D y 37 para A, porque en D puede demostrar que cada artículo está exactamente en un lugar (en el suelo de una habitación, en una mano o en un carro) y empaquetar esos hechos juntos.

El precio se paga en Fidelidad, por el lado de las preguntas que el modelo puede responder. En D, `(item-at meds1 icu)` es falso mientras la caja está en el carro en la UCI. El primer problema del carro, cuyo objetivo era solo ese hecho, necesita ahora 16 pasos en lugar de 11: el robot tiene que descargar y dejar en el suelo cada caja. Si «en el carro en la UCI» cuenta como entregado, el objetivo tiene que decirlo con una disyunción y un cuantificador, y lo mismo toda precondición que pregunte dónde está un artículo.

Veredicto.

- Usa un **efecto condicional** (A) cuando la acción es una sola cosa en el mundo y su resultado depende del estado, sobre todo cuando afecta a un número de objetos que el dominio no puede acotar. Es la codificación más fiel y la más clara. Su coste es la portabilidad.
- Usa **una acción por caso** (B) cuando los casos son pocos y fijos (dos o tres), como en la esclusa, o cuando el plan debe decir qué caso se dio. Evítala cuando los casos son combinaciones de objetos.
- Usa una **acción auxiliar** (C) solo cuando el planificador disponible no admite efectos condicionales y los casos no se pueden enumerar. Pon en cada objetivo la condición de que las acciones auxiliares han terminado, y recuerda que la longitud del plan ya no mide el esfuerzo.
- Antes de cualquiera de las tres, pregúntate si el hecho que actualiza el efecto tiene que almacenarse siquiera (D). Si se puede deducir de otro, almacenar solo el otro elimina la necesidad del efecto. La condición es que nada más en el modelo necesite leerlo directamente.

6.9. Errores comunes

Leer los efectos como un programa

Los efectos de una acción no se ejecutan en orden. Una conmutación escrita como «apágala si está encendida, y enciéndela» pierde la condición de su segunda mitad:

```
    :effect (and (when (lit ?l) (not (lit ?l)))
                 (lit ?l))))
```

Sobre una lámpara encendida, esta acción borra y añade `(lit ?l)`. Las adiciones se aplican después de los borrados, así que la lámpara sigue encendida, y la lámpara de `lights-clash-p01.pddl` no se puede apagar nunca:

```
$ kairos lights-clash-domain.pddl lights-clash-p01.pddl
note: no plan: every state reachable from the initial one was
     searched (1 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Un solo estado recorrido: la acción existe y no lleva a ninguna parte.

Solución: da a cada caso su propio `when`, con condiciones que se excluyan entre sí, y recuerda que todas se leen en el estado anterior a la acción.

El `when` olvidado

Un efecto universal sin su condición se aplica a todos los objetos. Este empujón olvidó preguntar qué artículos están en el carro:

```
                (forall (?i - item)
                  (and (not (item-at ?i ?from))
                       (item-at ?i ?to))))))
```

El fichero es válido y el planificador está encantado con él. Para las dos cajas que tienen que llegar a la UCI:

```
$ kairos courier-cart-teleport-domain.pddl courier-cart-teleport
    -p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(grab-cart r1 cart1 pharmacy)
(push-cart r1 cart1 pharmacy lobby)
(push-cart r1 cart1 lobby corridor)
(push-cart r1 cart1 corridor ward-b)
(push-cart r1 cart1 ward-b icu)
```

Nadie ha cargado nada: todos los artículos del hospital siguen al carro. **Síntoma:** un plan demasiado corto que nunca toca los objetos del objetivo. **Solución:** el `when`, y una prueba de la trampa con un artículo que *no* esté en el carro.

Un `when` en una precondición

`when` solo existe en los efectos. En una condición, «si la lámpara está encendida, no debe estar rota» se escribe con `imply`. Kairos no se detiene ante este error; avisa de que ha leído `when` como el nombre de un predicado que nadie ha declarado:

```
$ kairos check lights-bad-when-domain.pddl
lights-bad-when-domain.pddl:14:19: warning: 'when' is not
    declared: never true [dialect.undeclared_predicate]
 14 |     :precondition (when (lit ?l) (not (broken ?l)))
    |                   ^
valid, plannable
  typing: full
```

«Never true» significa que la acción no se puede aplicar nunca. Otros planificadores rechazan el fichero. **Solución:** `(imply (lit ?l) (not (broken ?l)))`, con `:disjunctive-preconditions`. Lee los avisos de `check` aunque el veredicto sea `valid`.

El requisito que falta

Un dominio que usa `when` o un efecto `forall` sin declarar `:conditional-effects` recibe un aviso de Kairos, que planifica con él de todos modos:

```
$ kairos check lights-noreq-domain.pddl
lights-noreq-domain.pddl:4:18: warning: uses conditional-effects
      but :requirements does not include :conditional-effects [
     semantic.undeclared_requirement]
 4 |   (:requirements :strips :typing :negative-preconditions)
   |                  ^
valid, plannable
  typing: full
  negative-preconditions: full
  conditional-effects: full
```

Un planificador más estricto rechaza el fichero o, peor, ignora el efecto. **Solución:** declara lo que usa el fichero; la lista que imprime `check` dice qué características son.

Un `when` dentro de un `when`

El efecto de un `when` es un literal o un `and` de literales. La gramática de PDDL no tiene `when` dentro de un `when`, ni tampoco `forall` dentro de uno. Kairos acepta la forma anidada (`lights-nested-domain.pddl` se comprueba como válido); un planificador que sigue la gramática al pie de la letra, no. **Solución:** une las condiciones. `(when a (when b e))` es `(when (and a b) e)`, y un `forall` va fuera del `when`.

Un daño que el objetivo no vigila

El plan de `courier-sterile-p04` entregó una muestra contaminada, y el plan de `courier-cart-settle-p03` dejó atrás las cajas. Los dos modelos eran correctos y los dos objetivos estaban incompletos. **Síntoma:** un plan más corto de lo esperado en el que falta un paso que creías necesario (el sellado, la descarga). **Solución:** para cada efecto condicional que pueda estropear algo, decide si el objetivo debe protegerlo, y mantén la prueba de la trampa en la batería de pruebas.

6.10. Resumen

Los efectos ya pueden depender del estado y alcanzar cualquier número de objetos. La Tabla 6.4 reúne las construcciones del capítulo.

Construcción	Significado	Se usa para
`(when c e)`	`e` ocurre si `c` se cumple en el estado anterior a la acción	la conmutación; la esclusa
`(forall (?x - t) e)`	`e` ocurre para cada objeto de tipo `t`	el interruptor general; dejar todos los carros
`(forall (?x - t) (when c e))`	`e` ocurre para cada objeto que cumple `c`	Lights Out; el maletín; el carro
`:conditional-effects`	el requisito de las tres formas	
`:adl`	los siete requisitos clásicos a la vez	

Tabla 6.4.: Las construcciones de este capítulo.

Las ideas que hay que retener:

- Una precondición dice si la acción se puede hacer; la condición de un `when` dice lo que hace. Las dos se leen en el estado anterior a la acción, y todos los efectos se aplican juntos, primero los borrados.

- Un efecto universal se expande sobre los objetos del problema cuando se instancia la acción. Permite que un mismo dominio sirva para problemas de cualquier tamaño.
- Un efecto condicional puede ser algo que hay que evitar. El planificador lo esquiva solo cuando un objetivo o una precondición necesitan lo que destruiría.
- El paso D hace dos preguntas más: si lo que cambia la acción depende de la situación, y si cambia muchos objetos a la vez. El paso E prueba cada efecto condicional con su condición cierta, con ella falsa y con un objetivo que tienta.
- Un efecto condicional compite con una acción por caso, con las acciones auxiliares y con no almacenar el hecho en absoluto. El primero es el más claro y el menos portable.

6.11. Lecturas complementarias

- Pednault (1989) es el artículo que definió ADL, con la semántica de los efectos condicionales y cuantificados que heredó PDDL.
- McDermott et al. (1998), el manual del primer PDDL, introduce los indicadores de requisito de la Tabla 6.2; la gramática de los efectos tal como está hoy se encuentra en Kovacs (2011).
- Haslum et al. (2019) tratan los efectos condicionales y universales en su presentación del PDDL clásico, con una discusión de cómo los admiten los planificadores.
- Hoffmann y Nebel (2001) describen cómo trata FF el ADL: cuantificadores expandidos en la instanciación, y efectos condicionales conservados y vistos uno a uno por la heurística.
- Helmert (2009) explica la traducción de una tarea PDDL con cuantificadores y efectos condicionales a las variables de estado con las que buscan Fast Downward y muchos planificadores posteriores.
- Ghallab et al. (2004) analizan, en su capítulo sobre representaciones clásicas, cómo se relacionan las extensiones de este capítulo con STRIPS puro.

6.12. Ejercicios

Ejercicios de la Sección 6.1 — Efectos que dependen de la situación

Ejercicio 6.1.1. Un hospital tiene 5 artículos y el mapa del mundo del mensajero (14 conexiones dirigidas), con un robot y un carro. Con la codificación de una acción por caso, ¿cuántas acciones instanciadas candidatas tienen en total de `push-cart-0` a `push-cart-3`, antes de que el planificador pode ninguna? ¿Cuántas añadiría `push-cart-4`? Después ejecuta `kairos -v` sobre `courier-cart-cases-p02.pddl` y explica por qué el número de acciones instanciadas que da es mucho menor que tu cuenta para 3 artículos.

Ejercicio 6.1.2. Para cada frase, di si una acción STRIPS con una lista fija de efectos puede modelarla y, si no, qué ingrediente falta (una condición en el efecto, un cuantificador o ambos): (a) «cerrar la válvula detiene el flujo en su tubería»; (b) «cerrar la válvula principal detiene el flujo en todas las tuberías»; (c) «el ascensor sube un piso, con quien esté dentro»; (d) «pulsar el botón de llamada lo enciende,

salvo que el ascensor ya esté en el piso»; (e) «el robot deja el artículo que lleva».

Ejercicios de la Sección 6.2 — Efectos condicionales: `when`

Ejercicio 6.2.1. Una acción tiene el efecto `(and (when (p) (q)) (when (q) (r)) (when (r) (not (p))))`, donde `p`, `q` y `r` son hechos sin argumentos. Se aplica al estado {`(p)`}, y luego otra vez al resultado, cuatro veces en total. Escribe los cuatro estados. ¿Qué estado daría la primera aplicación si los tres efectos se ejecutaran en orden, como un programa?

Ejercicio 6.2.2. Una lámpara con regulador tiene tres niveles (apagada, baja, alta) y un botón que pasa de apagada a baja, de baja a alta y de alta a apagada. Escribe el dominio `dimmer` con los predicados `(low ?l)` y `(high ?l)` y una sola acción `press`, y un problema con dos lámparas en el que una debe pasar de baja a apagada y la otra de apagada a alta. ¿Cuántos pasos tiene el plan?

Ejercicio 6.2.3. En `courier-sterile-domain.pddl`, la acción `seal` tiene la precondición `(not (sealed ?i))`. Quítala y vuelve a ejecutar los cuatro problemas de prueba. ¿Cambia algún plan? ¿Cambia el número de acciones instanciadas? Da un argumento para conservar la precondición y otro para quitarla.

Ejercicios de la Sección 6.3 — Efectos universales: `forall`

Ejercicio 6.3.1. En una planta con puertas (predicados `joins`, `open` y `locked` como en el Capítulo 5), un pulsador de alarma de incendios desbloquea y abre todas las puertas. Escribe un dominio `alarm` con las acciones `move` (a través de una puerta abierta), `open-door` (solo si no está bloqueada) y `raise-alarm` (el robot debe estar en una habitación con pulsador), y un problema en el que la única forma de entrar en una habitación bloqueada sea dar la alarma.

Ejercicio 6.3.2. El objetivo `(forall (?l - lamp) (not (lit ?l)))` y el efecto `(forall (?l - lamp) (not (lit ?l)))` se escriben con el mismo texto. Explica qué significa cada uno, qué requisito necesita cada uno y qué hace un planificador con cada uno cuando el problema tiene seis lámparas.

Ejercicios de la Sección 6.4 — Los dos juntos: para todo objeto tal que

Ejercicio 6.4.1. Modifica `lightsout-p01.pddl` para que las cinco lámparas empiecen encendidas. Antes de ejecutar el planificador, encuentra a mano un conjunto de botones que las apague todas. ¿Importa el orden de las pulsaciones? ¿Sirve de algo pulsar dos veces un botón? Comprueba tus respuestas con Kairos.

Ejercicio 6.4.2. Un robot que sujeta el asa de un carro y camina está empujando el carro. Reescribe `courier-cart-domain.pddl` sin `push-cart`: un único `move` que mueve también el carro que sujeta el robot, si lo hay, y lo que hay en él. Resuelve con él `courier-cart-p02` y compara el plan y el número de acciones instanciadas con los del capítulo.

Ejercicio 6.4.3. El dominio del carro se probó con un robot. Escribe un problema con dos robots, `r1` y `r2`, y un carro, cuyo objetivo sea `(and (cart-at cart1 icu) (cart-at cart1 lab))`. No debería

existir ningún plan. Ejecútalo, lee el plan que devuelve Kairos, encuentra el fallo del dominio y arréglalo.

Ejercicios de la Sección 6.5 — Efectos que hay que evitar: el laboratorio estéril

Ejercicio 6.5.1. Une el carro y el laboratorio estéril en un dominio (llámalo `courier-cart-sterile`). Decide qué hace la esclusa con las muestras que entran en un carro, y escribe un problema en el que dos muestras de la sala A deban llegar al laboratorio, utilizables. Lee el plan: ¿dónde las sella el robot?

Ejercicio 6.5.2. Escribe la variante del laboratorio estéril en la que la esclusa no se abre para un robot que lleva una muestra sin sellar (una precondición, ningún efecto condicional). Ejecuta con ella los problemas «sin plan» y «trampa» del capítulo. ¿Qué resultados cambian? ¿En qué clase de hospital sería fiel cada uno de los dos modelos?

Ejercicios de la Sección 6.6 — El requisito `:adl`

Ejercicio 6.6.1. Sustituye los requisitos de `lights-domain.pddl` por `:adl` y comprueba el fichero. ¿Qué características en uso indica `kairos check`, y son todas las que anuncia `:adl`? Explica por qué un planificador sin precondiciones disyuntivas podría ejecutar el fichero original y podría rechazar el nuevo.

Ejercicios de la Sección 6.7 — Qué hacen los planificadores con los efectos condicionales

Ejercicio 6.7.1. Toma el problema `lightsout-p01.pddl` y compila a mano su acción instanciada `(press b1)` en acciones STRIPS con precondiciones negativas, una por cada combinación de sus condiciones. ¿Cuántas son? ¿Cuántas necesitaría `(press b3)`, y cuántas los cinco botones juntos?

Ejercicio 6.7.2. Ejecuta `kairos -v` sobre `courier-sterile-p02` con el dominio del capítulo y con `courier-sterile-cases-domain.pddl` (problema `courier-sterile-cases-p02`). Compara el número de acciones instanciadas y de estados recorridos. ¿A cuál de las dos compilaciones de la Figura 6.5 corresponde el segundo dominio?

Ejercicios de la Sección 6.8 — El método MODEL: efectos que dependen de la situación

Ejercicio 6.8.1. Un pintor pinta habitaciones con una pistola de pintura. Pintar una habitación pinta sus paredes y además mancha todos los muebles que están en la habitación y no están cubiertos con una sábana. El pintor puede ir entre habitaciones unidas por una puerta, cubrir un mueble y llevar un mueble a la habitación de al lado. Aplica los pasos O y D del método (tabla de estado, tabla de acciones con la columna de este capítulo), escribe el dominio `paint` y pruébalo con un problema en el que haya que pintar el dormitorio sin manchar ni la cama ni la lámpara.

Ejercicio 6.8.2. Aplica el paso L al maletín. Escribe la codificación en la que una cosa que está en el maletín no tiene hecho `at` (la codificación D de este capítulo), resuelve con ella el mismo problema y

compárala con el dominio del Listado 6.4 según los cinco criterios.
¿Qué requisito ha costado el efecto condicional, y qué ha comprado?

Modelar problemas clásicos: patrones y decisiones

7.

Cada uno de los tres últimos capítulos añadió una pieza del lenguaje: tipos, condiciones más ricas, efectos más ricos. Con ellas podemos escribir cualquier modelo clásico. Otra cosa es que sepamos escribir uno *bueno*. Dale la misma historia a tres personas (un robot camarero lleva platos en una bandeja con dos sitios, y cada plato del menú llega después del anterior) y te devolverán tres dominios distintos. Uno tiene un predicado para cada sitio de la bandeja, otro cuenta los platos y el tercero ni siquiera tiene bandeja. Los tres pasan `kairos check`. Se diferencian en qué planes permiten, en cuántas acciones instanciadas producen y en lo bien que se las arregla un planificador con ellos. Saber la sintaxis no decidía nada de eso.

Este capítulo trata de esas decisiones. No trae PDDL nuevo. Toma el método MODEL, que ha ido creciendo pregunta a pregunta desde el Capítulo 2, y lo aplica entero a un problema nuevo, escribiendo los cinco artefactos. Después pone nombre a las soluciones a las que los modeladores recurren una y otra vez: once **patrones**, cada uno con la situación que lo pide, un ejemplo probado y lo que cuesta, y cuatro **antipatrones**, cada uno con el síntoma que lo delata. Los capítulos posteriores y el catálogo del Capítulo 19 se refieren a ellos por estos nombres.

El capítulo empieza con el camarero, de la historia a un modelo probado. Siguen tres secciones con los patrones: para el estado, para el control y para las acciones. Después vienen los antipatrones y un procedimiento para los dos malos resultados del paso E, un modelo sin plan y un modelo con un plan erróneo. La sección del método reúne en una lista de comprobación las preguntas de la Parte II, hace un índice de los patrones y cierra el caso del camarero con el paso L.

7.1 Un problema a partir de una historia: el camarero 145
Paso M: mapear la situación 146
Paso O: objetos y estado . 146
Paso D: dinámica 147
Paso E: ensayar el modelo 147
7.2 Patrones para el estado . 148
Tabla estática 149
Hechos espejo 150
Invariante de a lo sumo uno 150
Cadena de sucesores . . . 151
Huecos 152
7.3 Patrones para el control . 152
Testigo de fase 153
Testigo de turno 153
Cerrojo de recurso 155
7.4 Patrones para las acciones 155
Par inverso y acción sin vuelta atrás 155
Acción fusionada 157
7.5 Antipatrones 158
Borrado olvidado 158
Objetivo en el estado inicial 158
Gemelos desacompasados 159
Parámetro ocioso 159
7.6 Depurar un modelo . . . 160
Sin plan 160
Un plan erróneo 161
7.7 El método MODEL: el método clásico, consolidado 161
Alternativas comparadas 163
7.8 Errores comunes 164
7.9 Resumen 165
7.10 Lecturas complementarias 166
7.11 Ejercicios 166

7.1. Un problema a partir de una historia: el camarero

Un pequeño restaurante quiere un robot que atienda sus mesas. El dueño describe el trabajo:

> *El cocinero deja los platos terminados en el pase, el mostrador que separa la cocina del comedor. El camarero los lleva a las mesas en una bandeja. La bandeja tiene dos sitios. Cada plato es para una mesa, y en cada mesa los platos llegan en orden: nadie recibe el pescado antes que la sopa. La noche ha ido bien cuando cada plato que se ha pedido está en su mesa.*

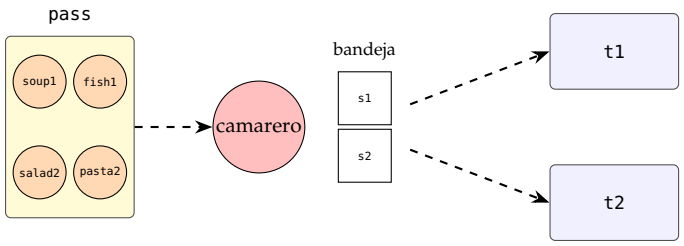


Figura 7.1.: El restaurante de la historia. Cuatro platos esperan en el pase; la bandeja tiene dos sitios; la sopa y el pescado son para la mesa 1, y la ensalada y la pasta, para la mesa 2.

Paso M: mapear la situación

La historia es corta y ya deja cosas sin decir. El paso M las hace explícitas antes de escribir una sola línea de PDDL. Cada línea de la ficha del modelo es una decisión que alguien podría haber tomado de otra manera.

Paso M: la ficha del modelo del camarero

Historia. Un robot camarero lleva platos del pase a las mesas, dos a la vez como mucho, respetando el orden de los platos en cada mesa.

Agentes. Un camarero. El cocinero y los clientes no son agentes: no hacen nada que decida el planificador.

Éxito. Cada plato pedido está en su mesa.

Alcance. *Dentro*: dónde está el camarero, qué hay en la bandeja, qué platos están servidos, qué plato va antes de cuál. *Fuera*: cocinar (los platos están listos cuando empieza el problema), la ruta entre lugares (la sala es diáfana y la navegación del robot encuentra el camino), el tiempo, comer, recoger las mesas, el pago.

Supuestos. Un plato servido sigue servido. Un plato que sale del pase no vuelve a él. El camarero nunca tira nada.

Dos líneas merecen una segunda mirada. Dejar fuera la ruta significa que el modelo no tendrá mapa: ir de cualquier lugar a cualquier otro es una sola acción. Es una decisión de **abstracción**, y es acertada si otro (el software de navegación) resuelve lo que dejamos fuera. El segundo supuesto, que un plato no vuelve al pase, parece inofensivo. Veremos en la Sección 7.4 que crea callejones sin salida.

Paso O: objetos y estado

¿Qué cosas existen? Lugares (el pase y las mesas), platos y los sitios de la bandeja, a los que llamamos huecos. El camarero es uno y siempre el mismo, así que no necesita objeto: su posición es un predicado con un argumento. El pase existe en todos los problemas y las acciones hablan de él, así que es una constante.

Propiedad	Sobre qué	Clase	Patrón
`(waiter-at ?p)`	el lugar donde está el camarero	dinámico	a lo sumo uno
`(ready ?d)`	un plato espera en el pase	dinámico	
`(on-tray ?d ?s)`	un plato está en un hueco de la bandeja	dinámico	huecos
`(empty ?s)`	un hueco no contiene nada	dinámico	huecos
`(served ?d)`	un plato está en su mesa	dinámico	
`(ordered ?d ?t)`	la mesa que pidió un plato	estático	tabla estática
`(before ?a ?b)`	el plato `?a` se sirve antes que `?b`	estático	tabla estática

Tabla 7.1.: La tabla de estado del camarero. La última columna nombra el patrón que hay detrás de cada fila; las secciones siguientes los definen.

La fila interesante es la del orden de los platos. Podríamos tener los tipos `starter` y `main` y una regla en el dominio. En lugar de eso, el orden es un dato: el problema enumera qué plato va antes de cuál, y el dominio solo sabe que existen esos pares. Así, un postre, o una mesa que se salta el primer plato, no necesitarán ningún cambio en el dominio.

Paso D: dinámica

Acción	Se aplica cuando	Cambia
`move ?from ?to`	el camarero está en `?from`; los lugares son distintos	el camarero está en `?to`
`load ?d ?s`	el camarero está en el pase; `?d` está listo; `?s` está vacío	`?d` está en `?s` y ya no está listo; `?s` ya no está vacío
`serve ?d ?s ?t`	el camarero está en `?t`; `?d` está en `?s` y lo pidió `?t`; todo plato anterior a `?d` está servido	`?d` está servido; `?s` vuelve a estar vacío

Tabla 7.2.: La tabla de acciones del camarero. Ningún efecto depende del estado, así que la tercera columna del Capítulo 6 no hace falta.

Las dos tablas se traducen al Listado 7.1 casi línea a línea. Las características vienen de los tres capítulos anteriores: una constante y una desigualdad (Capítulo 4), y una precondición universal con `imply` para «todo plato anterior a este» (Capítulo 5).

```
;; The waiter robot: dishes go from the kitchen pass to the
;; tables on a tray with places, each course in its turn.
(define (domain waiter)
  (:requirements :strips :typing :equality
                 :negative-preconditions
                 :disjunctive-preconditions
                 :universal-preconditions)

  (:types place dish slot)

  (:constants pass - place)         ; where the cook leaves dishes

  (:predicates
    (waiter-at ?p - place)
    (ready ?d - dish)               ; the dish waits on the pass
    (on-tray ?d - dish ?s - slot)
    (empty ?s - slot)               ; a free place of the tray
    (served ?d - dish)              ; the dish is on its table
    (ordered ?d - dish ?t - place)  ; static: the table of ?d
    (before ?a ?b - dish))          ; static: ?a goes before ?b

  ; The dining room is open: any place is one move away.
  (:action move
    :parameters (?from ?to - place)
    :precondition (and (waiter-at ?from) (not (= ?from ?to)))
    :effect (and (not (waiter-at ?from)) (waiter-at ?to)))

  (:action load
    :parameters (?d - dish ?s - slot)
    :precondition (and (waiter-at pass) (ready ?d) (empty ?s))
    :effect (and (not (ready ?d)) (not (empty ?s))
                 (on-tray ?d ?s)))

  ; A dish is served at its table, after the dishes before it.
  (:action serve
    :parameters (?d - dish ?s - slot ?t - place)
    :precondition (and (waiter-at ?t) (on-tray ?d ?s)
                       (ordered ?d ?t)
                       (forall (?e - dish)
                         (imply (before ?e ?d) (served ?e))))
    :effect (and (not (on-tray ?d ?s)) (empty ?s)
                 (served ?d))))
```

Listado 7.1: El camarero (`waiter-domain.pddl`).

Paso E: ensayar el modelo

La batería de pruebas tiene los cuatro problemas que pide el método. Todos usan dos huecos.

Trivial (`p01`): una mesa, un plato. Lo esperado: cargar, moverse, servir.
Típica (`p02`): los cuatro platos de la Figura 7.1.
Sin plan (`p03`): la sopa de la mesa 1 nunca salió de la cocina (no está `ready`), y el objetivo pide el pescado.
Trampa (`p04`): el objetivo pide solo los dos segundos platos. El atajo es servirlos y saltarse los primeros.

El problema trivial da los tres pasos esperados. Para el típico, Kairos imprime tres planes, de 16, 13 y 11 pasos, y demuestra que el último es óptimo:

```
$ kairos waiter-domain.pddl waiter-p02.pddl
...
; better plan 3
; cost = 11
(load fish1 s1)
(load soup1 s2)
(move pass t1)
(serve soup1 s2 t1)
(serve fish1 s1 t1)
(move t1 pass)
(load pasta2 s1)
(load salad2 s2)
(move pass t2)
(serve salad2 s2 t2)
(serve pasta2 s1 t2)
```

El plan lleva los dos platos de cada mesa en un solo viaje y sirve antes el primer plato. Es lo que hace un buen camarero, y nadie le ha hablado de viajes al modelo: se sigue de los dos huecos y del coste de moverse. El tercer problema termina con `kairos: unsolvable (exhausted)`. La prueba de la trampa devuelve un plan de 11 pasos que sirve los cuatro platos, aunque el objetivo solo nombra dos. Para estar seguros de que el atajo es imposible, y no solo que se ha pasado por alto, lo escribimos a mano y se lo damos al validador:

```
$ kairos validate waiter-domain.pddl waiter-p04.pddl waiter-cheat.
    plan
waiter-cheat.plan:5: error: step 4 (serve fish1 s1 t1) at 3.000:
    precondition false: for ?e = soup1: (imply (before ?e ?d) (
    served ?e)) with ?d = fish1 ?s = s1 ?t = t1 ?e = soup1 is false
     [validate.precondition_failed]
 5 | (serve fish1 s1 t1)
plan invalid
```

Una prueba de la trampa tiene dos mitades: el planificador no toma el atajo, y el validador rechaza el atajo escrito a mano, por la razón que esperamos.

Cuatro de los cinco artefactos están ya sobre la mesa: la ficha del modelo, la tabla de estado, la tabla de acciones y la batería de pruebas. El quinto, la comparativa del paso L, necesita alternativas, y las alternativas están hechas de patrones. Volveremos al camarero en la Sección 7.7.

7.2. Patrones para el estado

Un **patrón** es una forma, con nombre propio, de codificar algo que muchos problemas tienen en común. No es una regla. Cada patrón es una respuesta a una pregunta del paso O o del paso D, y suele tener un rival. Para cada uno decimos cuándo usarlo, lo mostramos en un fichero que se ejecuta y decimos lo que cuesta.

Tabla estática

Tabla estática (*static table*): una relación que ninguna acción cambia, escrita como hechos del estado inicial y leída por las precondiciones. El mapa del mundo del mensajero es una (`connected`); también lo son `ordered` y `before` en el camarero.

Cuándo usarlo. Siempre que una regla del mundo pueda ser distinta en otro problema: qué habitaciones están unidas, qué plato va primero, quién puede hacer qué. La prueba consiste en preguntarse qué cambiaría en otra instancia. Si la respuesta es «una línea de una acción», el conocimiento está en el sitio equivocado.

Ejemplo. Un granjero debe cruzar un río con un lobo, una cabra y una col en una barca en la que solo caben él y una cosa más. Si se quedan solos, el lobo se come a la cabra y la cabra se come la col. Un modelo directo tiene una acción para cada cosa, con los peligros escritos en las precondiciones. El Listado 7.2 no tiene nada de eso: quién se come a quién es una tabla estática, `(eats ?a ?b)`.

```
;; The farmer, the wolf, the goat and the cabbage. The boat is
;; where the farmer is; who eats whom is data of the problem.
(define (domain river)
  (:requirements :strips :typing :equality
                 :negative-preconditions
                 :existential-preconditions)

  (:types bank thing)

  (:predicates
    (boat-at ?b - bank)             ; the farmer and the boat
    (at ?x - thing ?b - bank)
    (eats ?a ?b - thing))           ; static: ?a eats ?b if alone

  ; The farmer crosses alone: nothing may be eaten behind him.
  (:action cross-alone
    :parameters (?from ?to - bank)
    :precondition (and (boat-at ?from) (not (= ?from ?to))
      (not (exists (?a ?b - thing)
             (and (eats ?a ?b) (at ?a ?from) (at ?b ?from)))))
    :effect (and (not (boat-at ?from)) (boat-at ?to)))

  ; The farmer crosses with ?x: nothing may be eaten among
  ; the things that stay.
  (:action cross-with
    :parameters (?x - thing ?from ?to - bank)
    :precondition (and (boat-at ?from) (at ?x ?from)
                       (not (= ?from ?to))
      (not (exists (?a ?b - thing)
             (and (eats ?a ?b) (at ?a ?from) (at ?b ?from)
                  (not (= ?a ?x)) (not (= ?b ?x))))))
    :effect (and (not (boat-at ?from)) (boat-at ?to)
                 (not (at ?x ?from)) (at ?x ?to))))
```

Listado 7.2: El acertijo del río, con sus reglas como datos (`river-domain.pddl`).

Las precondiciones dicen: tras la travesía, ningún par de la tabla `eats` se queda junto en la orilla que deja el granjero. Kairos encuentra las siete travesías de la solución clásica:

```
$ kairos river-domain.pddl river-p01.pddl
; cost = 7
(cross-with goat left right)
(cross-alone right left)
(cross-with wolf left right)
(cross-with goat right left)
(cross-with cabbage left right)
```

```
(cross-alone right left)
(cross-with goat left right)
```

El segundo problema añade un hecho, `(eats wolf cabbage)`, y no tiene plan. Un acertijo nuevo ha costado una línea de datos.

Coste. Casi ninguno durante la búsqueda: las precondiciones estáticas se evalúan al instanciar el modelo y podan acciones instanciadas (Capítulo 3). El coste está en los ficheros de problema, que crecen, y en los errores que se cometan en ellos, que ninguna comprobación de tipos detecta: un hecho que falta no es un error, es un mundo distinto.

Hechos espejo

Hechos espejo (*mirrored facts*): una relación simétrica escrita dos veces en la tabla estática, una en cada sentido, para que las acciones la lean en un solo sentido. Todos los problemas del mundo del mensajero lo hacen con su mapa: `(connected lobby dock)` y `(connected dock lobby)`.

Cuándo usarlo. Para cualquier relación que se cumpla en los dos sentidos (adyacente, conectado, al lado de, compatible), salvo que los ficheros de problema se escriban a mano y sean muy grandes.

Ejemplo y rival. El rival escribe cada par una sola vez y deja que la acción lo lea en los dos sentidos:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from)
                     (or (connected ?from ?to)
                         (connected ?to ?from)))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

Con él, el mapa de `courier-sym-p01.pddl` tiene siete hechos en lugar de catorce. El plan es el mismo, y también el número de acciones instanciadas, 30 en las dos versiones.

Coste. Los hechos espejo duplican la tabla y pueden escribirse de forma incoherente: olvida un sentido y el robot entrará en una habitación de la que no puede salir. El `or` mantiene la tabla corta y coherente, necesita `:disjunctive-preconditions` y pierde algo: ya no se puede expresar una puerta de un solo sentido. Los hechos espejo son la opción por defecto porque son STRIPS puro y porque una relación simétrica deja a menudo de serlo cuando el modelo crece.

Invariante de a lo sumo uno

Un **invariante** es una propiedad que se cumple en todo estado alcanzable. El más común dice que algo tiene un solo valor a la vez: el robot está en una habitación, el camarero en un lugar, cada cosa en una orilla. PDDL no tiene forma de declararlo. Se *mantiene* por la manera de escribir las acciones.

Invariante de a lo sumo uno (*at-most-one invariant*): toda acción que añade un valor de la propiedad borra el antiguo en el mismo efecto, y el estado inicial tiene exactamente uno. El `move` del mensajero es el modelo que seguir: `(not (robot-at ?r ?from))` junto a `(robot-at ?r ?to)`, con el

valor antiguo nombrado en la precondición para que la acción sepa qué borrar.

Cuándo usarlo. Para toda propiedad que el paso O describió con un artículo determinado o un posesivo: *la* posición, *su* estado, *el* que lo tiene.

Ejemplo. El planificador encuentra estos invariantes por sí solo, y lo dice. Con `-v`, Kairos informa de cómo agrupa los hechos en variables de estado:

```
$ kairos courier-domain.pddl courier-p03.pddl -v 2>&1 | grep sas
[   0.001s] [sas] SAS+ in 0.1 ms: 3 variables (2 from 3 mutex groups,
      1 binary), 3 invariants from 13 candidates, 0 axiom layers
```

Un **grupo mutex** es un conjunto de hechos de los que como mucho uno es verdadero a la vez. Los ocho hechos `robot-at` forman uno, y Kairos los guarda como una sola variable con ocho valores. Dieciocho hechos se han convertido en tres variables.

Coste. Ninguno. El invariante es gratis y hace más pequeña la búsqueda. También es frágil: basta una acción que olvide su borrado para romperlo, como muestra la Sección 7.5. El Capítulo 16 introduce una construcción, el fluente de objeto, que hace del valor único una parte del lenguaje.

Cadena de sucesores

El PDDL clásico no tiene números, y aun así los modelos necesitan contar: platos en una bandeja, plantas de un edificio, unidades de combustible. **Cadena de sucesores** (*successor chain*): los números son objetos, `n0`, `n1`, `n2`..., una tabla estática `(next ?n ?m)` dice que a cada uno le sigue el siguiente, y un hecho guarda la cuenta actual. Sumar uno sube ese hecho un paso por la cadena; restar uno lo baja un paso.

Cuándo usarlo. Para una cantidad pequeña y acotada de la que solo importa cuánto vale.

Ejemplo. La bandeja del camarero como contador (Listado 7.3).

```
28  ; One more dish: the count goes one step up the chain. With
29  ; the tray full there is no next count, and no load.
30  (:action load
31    :parameters (?d - dish ?n ?m - count)
32    :precondition (and (waiter-at pass) (ready ?d)
33                       (tray-count ?n) (next ?n ?m))
34    :effect (and (not (ready ?d)) (on-tray ?d)
35                 (not (tray-count ?n)) (tray-count ?m)))
36
37  ; One dish less: one step down.
38  (:action serve
39    :parameters (?d - dish ?t - place ?n ?m - count)
40    :precondition (and (waiter-at ?t) (on-tray ?d)
41                       (ordered ?d ?t)
42                       (tray-count ?m) (next ?n ?m)
43                       (forall (?e - dish)
44                         (imply (before ?e ?d) (served ?e))))
45    :effect (and (not (on-tray ?d)) (served ?d)
46                 (not (tray-count ?m)) (tray-count ?n))))
```

Listado 7.3: Una bandeja que cuenta (de `waiter-count-domain.pddl`).

La capacidad no aparece en ningún sitio del dominio. Es la longitud de la cadena en el problema: `(next n0 n1) (next n1 n2)` es una bandeja para dos. Con la cuenta en `n2`, no existe ningún hecho `(next n2 ?m)` y `load`

no se puede aplicar: el límite lo impone la ausencia de datos. El mismo problema de prueba da los mismos 11 pasos:

```
$ kairos waiter-count-domain.pddl waiter-count-p02.pddl
...
; better plan 2
; cost = 11
(load soup1 n0 n1)
(load fish1 n1 n2)
(move pass t1)
(serve soup1 t1 n1 n2)
(serve fish1 t1 n0 n1)
...
```

Coste. Dos parámetros más en cada acción que cuenta, y planes más difíciles de leer. Solo «más uno» y «menos uno» salen gratis; sumar dos cantidades exige una tabla con todas las sumas. A partir de unas pocas decenas de valores, la cadena es la herramienta equivocada, y el Capítulo 8 trae números de verdad.

Huecos

Huecos (*slots*): la capacidad es un conjunto de objetos, uno por cada unidad de espacio, con un hecho que dice si está libre. Coger algo ocupa un hueco libre; soltarlo libera el hueco. El camarero del Listado 7.1 es el ejemplo: `(empty ?s)` y `(on-tray ?d ?s)`. Las dos pinzas del robot del dominio gripper también son huecos (Figura 7.2).

Cuándo usarlo. Cuando las unidades de capacidad son cosas distintas en el mundo, o cuando importa *dónde* está cada artículo: un estante con posiciones numeradas, un aparcamiento, dos manos.

Coste. La simetría. Si los huecos son intercambiables, cada plan existe en tantas versiones como formas haya de asignar artículos a huecos, y el planificador puede explorarlas todas. Las cifras del camarero lo muestran. Con huecos, la búsqueda para `p02` visita 220 estados; con la cadena, que no dice dónde está cada plato, 123.

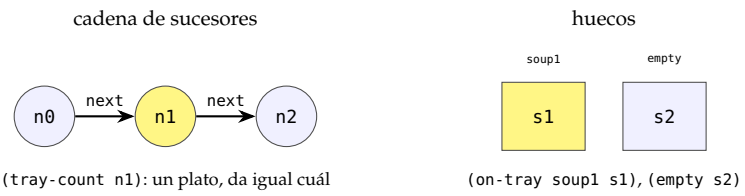


Figura 7.2.: Dos codificaciones de una bandeja para dos que lleva un plato. La cadena guarda un solo hecho que se desplaza a lo largo de `next`; los huecos guardan un hecho por cada sitio, y la misma situación tiene un gemelo con `soup1` en `s2`.

Cadena o huecos: cuenta cuando solo importa cuántos hay; usa huecos cuando los sitios son distintos o importa cuál contiene qué.

7.3. Patrones para el control

Un plan clásico es una secuencia libre: cualquier acción aplicable puede venir a continuación. Muchos problemas no son tan libres. Las cosas ocurren por etapas, los agentes se alternan, una herramienta está en una sola mano a la vez. Los tres patrones de esta sección imponen esa disciplina con el mismo recurso, un **testigo**: un hecho que dice quién o qué tiene permiso ahora, que las acciones exigen y que algunas acciones pasan a otro.

Testigo de fase

Testigo de fase (*phase token*): un hecho nombra la fase actual del proceso. Cada acción exige el testigo de su fase, y una acción especial lleva el testigo a la fase siguiente, nunca hacia atrás.

Cuándo usarlo. Cuando el trabajo tiene etapas que no deben mezclarse: todo el corte antes de cualquier encolado (el serrín estropea la cola), la puesta a punto antes de la operación, la carga antes de cerrar el horno.

Ejemplo. Un taller construye un taburete con tres piezas. Cada pieza se corta, se ensambla y se pinta, y el taller hace primero todo el corte, después todo el ensamblado y por último toda la pintura.

```
  (:action cut
    :parameters (?x - part)
    :precondition (and (now cutting) (not (cut ?x)))
    :effect (cut ?x))
  ; The token moves forward and never comes back.
  (:action advance
    :parameters (?a ?b - phase)
    :precondition (and (now ?a) (follows ?b ?a))
    :effect (and (not (now ?a)) (now ?b))))
```

```
$ kairos workshop-domain.pddl workshop-p01.pddl
; cost = 11
(cut seat)
(cut leg1)
(cut leg2)
(advance cutting joining)
(join seat)
...
(advance joining painting)
(paint seat)
(paint leg1)
(paint leg2)
```

Listado 7.4: Fases con un testigo (de `workshop-domain.pddl`). Las fases son constantes; `follows` es una tabla estática del problema.

Las precondiciones de cada pieza (`join` necesita `cut`) ya ordenan el trabajo *de una pieza*. El testigo ordena el trabajo *del taller*. Un plan escrito a mano que termina el asiento antes de tocar las patas lo rechaza el validador en su segundo paso, con `(now joining) is false`.

Coste. Una acción más en el plan por cada cambio de fase, y callejones sin salida: avanza demasiado pronto y una pieza que no se cortó ya no se cortará nunca. A cambio, la búsqueda es mucho más pequeña, porque desaparecen los entrelazados de las tres piezas entre fases. El testigo debe existir exactamente una vez en el estado inicial (Sección 7.8).

Testigo de turno

Testigo de turno (*turn token*): un hecho dice a quién le toca mover. Las acciones de cada agente exigen su turno y pasan el turno al agente siguiente.

Cuándo usarlo. Cuando los agentes deben alternarse y, sobre todo, para meter en un modelo clásico un mundo que se mueve solo de una forma conocida: tras cada paso del robot, la cinta transportadora avanza, el semáforo cambia, el vigilante sigue su ronda. El «otro agente» es entonces el mundo, con exactamente una acción disponible en su turno.

Ejemplo. Un robot debe cruzar una sala mientras una máquina de limpieza da vueltas por ella siguiendo una ruta fija (Figura 7.3). Se mueven por turnos, una celda cada uno, y nunca deben compartir celda.

```
; The robot's turn: one step, never into the machine.
(:action move
  :parameters (?from ?to - cell)
  :precondition (and (turn robot) (robot-at ?from)
                     (connected ?from ?to)
                     (not (machine-at ?to)))
  :effect (and (not (robot-at ?from)) (robot-at ?to)
               (not (turn robot)) (turn machine)))

; The robot may also let its turn pass.
(:action wait
  :parameters ()
  :precondition (turn robot)
  :effect (and (not (turn robot)) (turn machine)))

; The machine's turn: the only thing that can happen. If the
; robot stands in its way, nothing can, and the plan dies.
(:action machine-step
  :parameters (?from ?to - cell)
  :precondition (and (turn machine) (machine-at ?from)
                     (route ?from ?to) (not (robot-at ?to)))
  :effect (and (not (machine-at ?from)) (machine-at ?to)
               (not (turn machine)) (turn robot))))
```

Listado 7.5: El robot y la máquina se turnan (de `patrol-domain.pddl`).

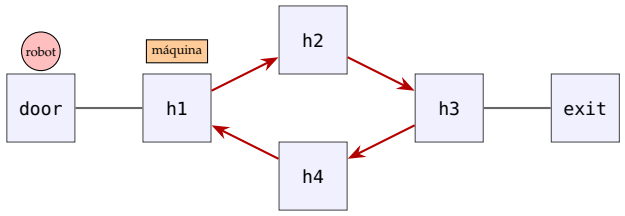


Figura 7.3.: La sala de `patrol-p01`. Las flechas rojas son la ronda de la máquina, que empieza en `h1`; el robot empieza en la puerta y quiere llegar a la salida.

```
$ kairos patrol-domain.pddl patrol-p01.pddl
...
; better plan 2
; cost = 9
(wait)
(machine-step h1 h2)
(move door h1)
(machine-step h2 h3)
(move h1 h2)
(machine-step h3 h4)
(move h2 h3)
(machine-step h4 h1)
(move h3 exit)
```

El robot espera un turno y luego sigue a la máquina alrededor del anillo, siempre una celda por detrás. Dos detalles hacen que el patrón funcione. La máquina no tiene elección: en su turno hay exactamente un `machine-step` aplicable, así que el planificador solo lo «elige» de nombre. Y si el robot estuviera en la siguiente celda de la máquina, ningún `machine-step` sería aplicable, el turno no volvería nunca y esa rama de la búsqueda moriría. Un paso del mundo que no puede ocurrir es la forma que tiene este patrón de decir «prohibido».

Coste. Planes el doble de largos, la mitad de ellos pasos que nadie ejecuta, y el tiempo contado en turnos. Si el robot y la máquina tuvieran velocidades distintas, o el mundo cambiara con el reloj hiciera lo que hiciera el robot,

los turnos encajarían mal; las Partes IV y VI tienen las herramientas adecuadas.

Cerrojo de recurso

Cerrojo de recurso (*resource lock*): una cosa que solo un agente puede usar a la vez se representa con un hecho que dice que está disponible. Adquirir el recurso borra ese hecho, y liberarlo lo vuelve a añadir. Nadie puede adquirir lo que no está disponible.

Cuándo usarlo. Para herramientas, llaves, cargadores, habitaciones estrechas, el asa de un carro: cualquier recurso compartido por varios agentes, o por varias actividades de uno solo.

Ejemplo. El hospital tiene una sola tarjeta de acceso para sus habitaciones restringidas. Cuelga en el vestíbulo.

```
39  ; Acquire: the card leaves its hook, so nobody else can
40  ; take it.
41  (:action take-keycard
42    :parameters (?r - robot ?l - location)
43    :precondition (and (robot-at ?r ?l) (keycard-at ?l))
44    :effect (and (not (keycard-at ?l)) (has-keycard ?r)))
45
46  ; Release: the card is available again, here.
47  (:action leave-keycard
48    :parameters (?r - robot ?l - location)
49    :precondition (and (robot-at ?r ?l) (has-keycard ?r))
50    :effect (and (not (has-keycard ?r)) (keycard-at ?l))))
```

Listado 7.6: Una tarjeta para dos robots (de `courier-keycard-domain.pddl`). `move` exige que la habitación no sea restringida o que el robot tenga la tarjeta.

El hecho de disponibilidad es `(keycard-at ?l)`. Como `take-keycard` lo borra, la tarjeta nunca puede estar en dos manos. La prueba de la trampa pide justo eso, `(and (has-keycard r1) (has-keycard r2))`, y es irresoluble. En el problema típico, con medicamentos en la farmacia restringida y una muestra en la UCI restringida, un robot coge la tarjeta y hace las dos entregas en 16 pasos: pasarse la tarjeta costaría más pasos de los que ahorra.

Coste. Un cerrojo que se adquiere y no se libera nunca bloquea a todos los demás durante el resto del plan, y el planificador no lo liberará salvo que algo lo necesite. Cuando el recurso deba acabar libre, dilo en el objetivo. En los planes secuenciales, un cerrojo solo importa para los recursos que se retienen *a lo largo de* varias acciones. Con acciones concurrentes (Capítulo 10) se vuelve imprescindible.

El carro del Capítulo 6 tenía este defecto: dos robots podían agarrar la misma asa, porque `grab-cart` no borraba ningún hecho de disponibilidad. Lo que faltaba en el Ejercicio 6.4.3 era un cerrojo.

7.4. Patrones para las acciones

Par inverso y acción sin vuelta atrás

Para cada acción, el paso D debería preguntar: ¿se puede deshacer? La respuesta es una decisión de modelado con dos resultados, cada uno con su nombre.

Par inverso (*inverse pair*): dos acciones tales que cada una restaura lo que la otra cambió. Coger y soltar un artículo, abrir y cerrar una puerta, coger

la tarjeta y dejarla son pares inversos. **Acción sin vuelta atrás** (*one-way action*): una acción sin inversa, cuyo efecto es permanente: `seal`, `paint`, `serve`, el `advance` de una fase.

Cuándo usar cada uno. Sigue al mundo. Si el mundo tiene la inversa y omitirla solo ahorra escritura, ponla: un modelo sin ella tiene **callejones sin salida**, estados desde los que no se puede alcanzar el objetivo y que el mundo no tiene. Si el mundo no tiene inversa, no te la inventes: el planificador la usaría para escapar de errores que son definitivos.

Ejemplo de acción sin vuelta atrás. En Sokoban el jugador empuja cajas y no puede tirar de ellas (Listado 7.7). La cuadrícula es una tabla estática con una dirección en cada hecho, para poder decir «tres celdas en línea»; las cajas no tienen nombre, porque una caja vale tanto como otra.

```
(:action walk
  :parameters (?from ?to - cell ?d - dir)
  :precondition (and (player-at ?from) (adjacent ?from ?to ?d)
                     (not (box-at ?to)))
  :effect (and (not (player-at ?from)) (player-at ?to)))

; Three cells in a line: the player, the box, the cell behind.
(:action push
  :parameters (?p ?from ?to - cell ?d - dir)
  :precondition (and (player-at ?p) (box-at ?from)
                     (adjacent ?p ?from ?d)
                     (adjacent ?from ?to ?d)
                     (not (box-at ?to)))
  :effect (and (not (player-at ?p)) (player-at ?from)
               (not (box-at ?from)) (box-at ?to))))
```

Listado 7.7: Sokoban: `walk` es su propia inversa, `push` no tiene ninguna (de `sokoban-domain.pddl`).

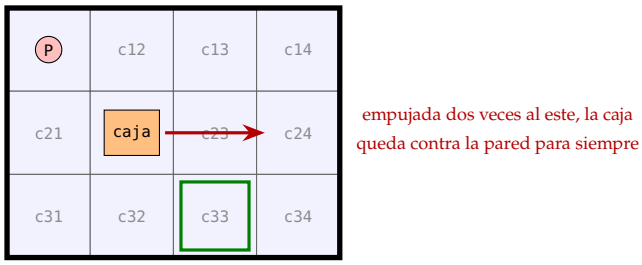


Figura 7.4.: La habitación de `sokoban-p01`: el jugador `P`, la caja y la celda objetivo (en verde). Dos empujones hacia el este dejan la caja contra la pared derecha, desde donde ningún empujón la devuelve a la columna 3: un callejón sin salida.

```
$ kairos sokoban-domain.pddl sokoban-p01.pddl
; cost = 5
(walk c11 c12 east)
(push c12 c22 c32 south)
(walk c22 c21 west)
(walk c21 c31 south)
(push c31 c32 c33 east)
```

En `sokoban-p02` la caja empieza en una esquina, y Kairos explica el fallo sin buscar: `(box-at c22) can never be reached: push adds it, but is never applicable`.

Ejemplo de inversa que falta. El camarero no puede devolver un plato al pase. Si la bandeja está llena de segundos platos mientras los primeros esperan (`waiter-p05`), no se puede servir nada ni cargar nada: no hay plan. ¿Es fiel? Un camarero de verdad devolvería un plato al pase. La ficha del modelo supuso lo contrario, y aquí ese supuesto cuesta soluciones. El arreglo es la inversa de `load`:

```
; The inverse of load: same places, effects turned around.
(:action unload
```

```
    :parameters (?d - dish ?s - slot)
    :precondition (and (waiter-at pass) (on-tray ?d ?s))
    :effect (and (ready ?d) (empty ?s)
                 (not (on-tray ?d ?s))))
```

Una inversa tiene los efectos de la original dados la vuelta, adiciones en lugar de borrados y borrados en lugar de adiciones, y como precondición lo que la original dejó verdadero. Con ella, la misma situación tiene un plan de 11 pasos que empieza con `(unload fish1 s1)`.

Coste. Los pares inversos eliminan callejones sin salida y agrandan el espacio de estados: el planificador puede ahora ir y volver indefinidamente, y confía en su memoria de estados visitados para no hacerlo. Las acciones sin vuelta atrás mantienen pequeño el espacio y hacen que importe el orden de las decisiones. Las heurísticas basadas en la relajación de borrados del Capítulo 3 no ven la mayoría de los callejones sin salida, así que un modelo lleno de ellos es difícil por razones que el número de acciones instanciadas no muestra.

Acción fusionada

¿Qué parte del mundo cabe en una acción? «Entregar la sopa» puede ser un paso o cinco. La **granularidad** de un modelo es esa elección. **Acción fusionada** (*merged action*): varias cosas que siempre ocurren juntas, en el mismo orden y sin ninguna decisión entre medias, se convierten en una sola acción. Su opuesto, la *acción dividida* (*split action*), parte una acción en dos allí donde aparece una decisión.

Cuándo usarlo. Fusiona cuando el planificador no tiene nada que decidir entre los pasos, o cuando quien ejecuta el plan lo decide mejor. Divide cuando entre medias puede ocurrir algo que importa: la acción de otro agente, la elección de un recurso, un fallo del que recuperarse.

Ejemplo. Supongamos que los viajes del camarero no son asunto del planificador: el controlador del robot agrupa los platos en la bandeja por su cuenta. Entonces el modelo puede fusionar `load`, `move` y `serve`:

```
  (:action deliver
    :parameters (?d - dish ?t - place)
    :precondition (and (ready ?d) (ordered ?d ?t)
                       (forall (?e - dish)
                         (imply (before ?e ?d) (served ?e))))
    :effect (and (not (ready ?d)) (served ?d))))
```

Sin bandeja, sin posición, cuatro acciones instanciadas en lugar de 22, y un plan de cuatro pasos para el problema que necesitaba 11. El orden de los platos se sigue respetando.

Coste. Una acción fusionada oculta lo que pasa dentro de ella. El camarero de grano grueso no puede contar viajes, así que no puede preferir los planes que hacen menos, y aceptaría una bandeja con diez platos. Lo que un modelo no representa, el planificador no puede ni optimizarlo ni prohibirlo. En el otro extremo, cada división multiplica la longitud de los planes y la profundidad de la búsqueda. El nivel al que hay que apuntar es aquel en el que se toman las decisiones de la historia.

Modela al nivel de las decisiones que debe tomar el planificador. Fusiona lo que no implica ninguna elección; divide donde pueda interponerse una elección, otro agente o un fallo.

7.5. Antipatrones

Un **antipatrón** es un error lo bastante común como para merecer un nombre. Los cuatro de esta sección tienen en común algo muy desagradable: el fichero es PDDL válido y `kairos check` no protesta. A cada uno solo se le descubre por su síntoma, así que el síntoma es lo que hay que recordar.

Borrado olvidado

Borrado olvidado (*forgotten delete*): una acción añade el nuevo valor de una propiedad y no borra el antiguo. El invariante de a lo sumo uno se rompe.

```
  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
    :effect (robot-at ?r ?to))
```

El robot de este dominio está en todas las habitaciones que ha visitado. Si se le pide llevar los medicamentos a la sala A y volver a la base de carga (`p03`), el dominio correcto necesita 10 pasos. Este necesita 6:

```
$ kairos courier-nodelete-domain.pddl courier-nodelete-p03.pddl
; cost = 6
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Tras recoger los medicamentos en la farmacia, el robot sale *desde el vestíbulo*, donde también sigue estando, y nunca vuelve a la base de carga, porque nunca se fue de ella.

Síntomas. Un plan demasiado corto; una acción aplicada desde un lugar donde el agente ya no debería estar. Y una señal que no necesita ningún plan: la línea `[sas]` de `kairos -v` informa de `10 variables (1 from 2 mutex groups, 9 binary)`, cuando el dominio correcto tenía 3 variables. Si una propiedad que debería tener un solo valor no aparece como grupo mutex, alguna acción no está borrando.

Objetivo en el estado inicial

Objetivo en el estado inicial (*goal in the initial state*): el objetivo ya es verdadero antes de cualquier acción. Viene de copiar líneas del objetivo en `:init`, de un objetivo demasiado débil o de un objetivo negativo sobre un hecho que nunca fue verdadero.

```
$ kairos waiter-domain.pddl waiter-done-p01.pddl
; cost = 0
```

Síntoma. Un plan de coste 0 sin ningún paso. Kairos informa de éxito, y tiene razón: el problema, tal como está escrito, está resuelto. El plan vacío es fácil de ver. La variante peligrosa es el objetivo que solo es *en*

parte verdadero al principio: el plan parece normal y se salta parte del trabajo. Lee el objetivo frente al estado inicial antes de ejecutar nada.

Gemelos desacompasados

Los modelos suelen mantener dos predicados que dicen lo mismo de dos maneras: `open` y `closed`, `empty` y `on-tray`, `free` y `carrying`. Esos gemelos son legítimos. Evitan las precondiciones negativas y hacen legibles las acciones. **Gemelos desacompasados** (*out-of-step twins*): una acción actualiza un gemelo y se olvida del otro, y entonces el estado dice dos cosas que no pueden ser verdad a la vez.

El fichero `waiter-bug-domain.pddl` es el camarero con un hecho de menos en el efecto de `serve`:

```
41    :effect (and (not (on-tray ?d ?s))
42                 (served ?d))))
```

El plato sale del hueco y el hueco no se declara `empty`. Tras dos platos, la bandeja no lleva nada y no tiene ningún sitio libre.

Síntoma. La prueba trivial pasa y la típica no tiene plan. Averiguar por qué es el tema de la Sección 7.6.

Prevención. Para cada pareja de gemelos, escribe el invariante («un hueco está vacío exactamente cuando no hay ningún plato en él») y comprueba contra él cada acción que toque a uno de los dos. O elimina uno de los gemelos y usa una negación, al precio de un requisito. El Capítulo 13 ofrece una tercera vía: definir un gemelo como predicado derivado del otro, de modo que no pueda desacompasarse.

Parámetro ocioso

Parámetro ocioso (*idle parameter*): un parámetro que ninguna precondición restringe y ningún efecto usa, o cuyo valor no cambia lo que hace la acción.

```
28  (:action load
29    :parameters (?d - dish ?s - slot ?t - place)
30    :precondition (and (waiter-at pass) (ready ?d) (empty ?s))
```

Alguien pensó que cargar un plato debía mencionar su mesa. El parámetro `?t` está libre, así que cada carga existe una vez por cada lugar: `(load fish1 s2 pass)` y `(load fish1 s2 t1)` son acciones distintas que hacen lo mismo.

Síntoma. Planes con argumentos que no tienen sentido, y un recuento: `kairos -v` informa de 38 acciones instanciadas para `p02`, cuando el dominio correcto tiene 22. Con tres mesas el daño es pequeño. Un parámetro libre de un tipo con mil objetos multiplica la acción por mil, y dos parámetros así, por un millón.

La prueba para cada parámetro: ¿lo restringe alguna precondición, y aparece en un efecto o decide qué efecto se aplica? Un parámetro puede hacer falta solo para consultar algo en una tabla estática, como `?t` en `serve`, donde `(ordered ?d ?t)` deja un único valor posible. Ese no es ocioso, y no cuesta nada tras la instanciación.

7.6. Depurar un modelo

El paso E puede acabar mal de dos maneras: el planificador no encuentra plan para un problema que lo tiene, o encuentra un plan que el mundo no aceptaría. En ninguno de los dos casos ayuda mucho el planificador. Dice «irresoluble», pero no qué línea tiene la culpa; imprime un plan erróneo con la misma seguridad que uno correcto. Hace falta un procedimiento (Figura 7.5).

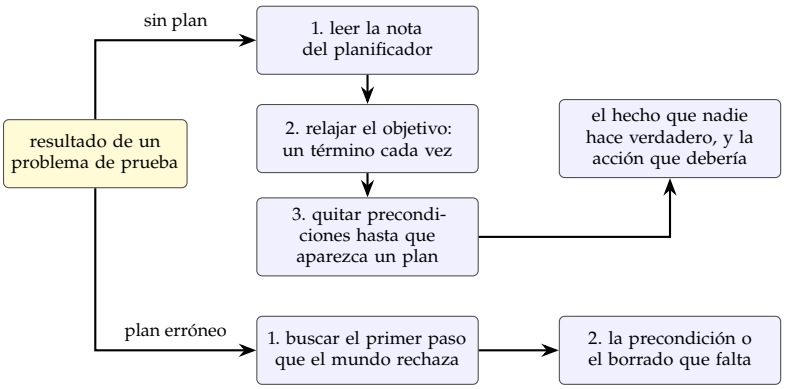


Figura 7.5.: El procedimiento de depuración del paso E. Arriba, para un problema que debería tener plan y no lo tiene; abajo, para un plan que no debería existir.

Sin plan

1. Lee lo que dice el planificador. Cuando el objetivo necesita un hecho que ninguna secuencia de acciones puede producir, ni siquiera ignorando los borrados, Kairos dice cuál y por qué. El error de la Sección 7.8 con un mapa de un solo sentido se diagnostica así, sin ninguna búsqueda. Cuando la nota solo dice que se ha explorado todo estado alcanzable, el modelo puede producir cada hecho del objetivo, y algo más sutil impide producirlos juntos. Pasa al punto siguiente.

2. Relaja el objetivo. Sustituye el objetivo por uno de sus términos, luego por dos, y así sucesivamente. Lo que se busca es el objetivo más pequeño que no tiene plan. Para el camarero con los gemelos desacompasados, `p02` pide cuatro platos y falla. Escribimos copias del problema con objetivos más débiles:

```
$ kairos waiter-bug-domain.pddl waiter-bug-p06.pddl      # (served
     soup1)
; cost = 3
(load soup1 s1)
(move pass t1)
(serve soup1 s1 t1)
$ kairos waiter-bug-domain.pddl waiter-bug-p07.pddl      # the two
     starters
; cost = 6
(load salad2 s1)
(load soup1 s2)
(move pass t1)
(serve soup1 s2 t1)
(move t1 t2)
(serve salad2 s1 t2)
$ kairos waiter-bug-domain.pddl waiter-bug-p08.pddl      # two
     starters and the fish
note: no plan: every state reachable from the initial one was
     searched (63 states) and none satisfies the goal. [orchestrator.
     why]
```

```
kairos: unsolvable (exhausted)
```

Se puede servir un plato, y dos. Tres no, aunque el pescado por sí solo no plantea ninguna dificultad una vez servida su sopa. El modelo falla en *el tercer plato*, sea cual sea. Eso apunta a la bandeja.

3. Quita precondiciones. Toma la acción que debería haberse aplicado, aquí la tercera `load`, y quítale las precondiciones de una en una en una copia del dominio, hasta que aparezca un plan. La precondición cuya eliminación devuelve el plan nombra el hecho que el modelo no consigue producir. En `waiter-bug-relaxed-domain.pddl`, `load` ha perdido `(empty ?s)`:

```
$ kairos waiter-bug-relaxed-domain.pddl waiter-bug-relaxed-p02.pddl
; cost = 10
(load soup1 s1)
(load salad2 s1)
(load pasta2 s1)
(load fish1 s1)
(move pass t1)
...
```

El plan es absurdo (cuatro platos en un mismo hueco), y eso no importa: existe, así que `(empty ?s)` era el obstáculo. La última pregunta se le hace al dominio: *¿qué acción añade `(empty ?s)`?* Ninguna. Solo es verdadero en el estado inicial. Al `serve` que debería restaurarlo se le ha perdido una línea.

Sin plan: busca el objetivo más pequeño que falla, después la precondición cuya eliminación devuelve un plan, y por último pregúntate qué acción debería haber hecho verdadero ese hecho.

Un plan erróneo

Un plan es erróneo cuando el mundo no lo permitiría. Léelo paso a paso frente a la historia, no frente al PDDL, y detente en el *primer* paso que el mundo rechazaría. En el plan del borrado olvidado es el cuarto: el robot sale del vestíbulo mientras está en la farmacia. Después pregúntate por qué el modelo permitió ese paso. Hay dos respuestas posibles.

- **Falta una precondición**: a la acción le falta la condición que la historia da por supuesta. Añádela.
- La precondición está y era verdadera cuando debería haber sido falsa: **una acción anterior dejó un hecho erróneo**, casi siempre por un borrado olvidado. Escribe a mano el estado anterior al paso, a partir del estado inicial y de los efectos, y busca el hecho que no debería estar ahí.

Cuando el plan erróneo viene de una prueba de la trampa, convierte el caso reparado en una prueba de regresión: el atajo escrito a mano como fichero de plan, con una línea `validate` en `tests.txt` que espera que se rechace. Las baterías de pruebas de este libro están llenas de ellas.

7.7. El método MODEL: el método clásico, consolidado

Desde el Capítulo 2, cada capítulo ha añadido preguntas a los cinco pasos. Esta sección reúne en un solo sitio la parte clásica del método.

A partir de aquí el libro le añade números, tiempo y preferencias; el esqueleto no cambia.

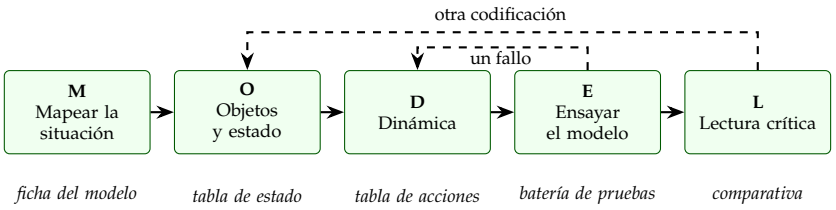


Figura 7.6.: Los cinco pasos y el artefacto que produce cada uno. Las flechas discontinuas son el bucle: una prueba fallida devuelve a la dinámica, y una alternativa mejor, al estado.

La lista de comprobación clásica

M. Mapear la situación. ¿Cuál es la historia? ¿Quién actúa y quién es solo parte del escenario? ¿Qué cuenta como resuelto? ¿Qué se deja fuera y quién se ocupa de ello? ¿Qué supuestos hacen clásico el problema (nada cambia por sí solo, las acciones son instantáneas y nunca fallan)?

O. Objetos y estado. ¿Qué clases de cosas existen, hay una jerarquía, pertenece algún objeto a todos los problemas (Capítulo 4)? Para cada propiedad: ¿es estática o dinámica (Capítulo 2)? ¿Habla de una ausencia, de algún objeto, de todos los objetos (Capítulo 5)? ¿Tiene un solo valor a la vez? ¿Es una cantidad, y solo importa cuánto vale? ¿Se puede deducir de otra y, por tanto, no hace falta guardarla (Capítulo 6)?

D. Dinámica. Para cada acción: ¿cuándo se aplica y qué cambia, diciendo todo lo que cambia y nada de lo que no cambia (Capítulo 2)? ¿Pueden dos parámetros ser el mismo objeto (Capítulo 4)? ¿Depende de la situación lo que cambia, cambia muchos objetos a la vez (Capítulo 6)? ¿Se puede deshacer? ¿Es una decisión o varias? ¿Hace falta cada parámetro?

E. Ensayar el modelo. Estima el tamaño antes de ejecutar (Capítulo 3). Ejecuta los problemas trivial, típico, sin plan y de la trampa; lee y valida cada plan; escribe las trampas a mano y comprueba que se rechazan. Si algo falla, sigue el procedimiento de la Sección 7.6.

L. Lectura crítica. ¿Qué otras codificaciones existen? El catálogo de patrones las sugiere: todo patrón tiene un rival. Compara dos o tres según la Fidelidad, el Tamaño, la Guía, la Claridad y la Portabilidad, con cifras, y decide.

Las preguntas de la lista que no llevan referencia a ningún capítulo son las que añade este capítulo, y cada una de ellas conduce a un patrón. La Tabla 7.3 es el índice.

Patrón	Úsalo cuando	Lo que cuesta
Tabla estática	una regla puede variar entre problemas	ficheros de problema más largos
Hechos espejo	una relación se cumple en los dos sentidos	una tabla el doble de larga
Invariante de a lo sumo uno	una propiedad tiene un solo valor	nada; un borrado olvidado lo rompe
Cadena de sucesores	una cantidad pequeña; solo importa cuánto vale	dos parámetros por acción
Huecos	las unidades de capacidad son sitios distintos	simetría entre los huecos
Testigo de fase	las etapas no deben mezclarse	callejones sin salida; una acción por etapa
Testigo de turno	se alternan los agentes, o el agente y el mundo	planes el doble de largos
Cerrojo de recurso	un solo usuario a la vez	hay que pedir que se libere
Par inverso	el mundo puede deshacerlo	un espacio de estados mayor
Acción sin vuelta atrás	el mundo no puede deshacerlo	callejones sin salida
Acción fusionada	ninguna decisión entre los pasos	lo de dentro no se puede optimizar

Tabla 7.3.: El catálogo de patrones de la planificación clásica.

Alternativas comparadas

El paso L del camarero sigue abierto. Los patrones dan tres codificaciones de la misma historia, todas escritas y ejecutadas en este capítulo: la bandeja como **huecos** (Listado 7.1), la bandeja como **cadena de sucesores** (Listado 7.3) y la **acción fusionada** sin bandeja. La Tabla 7.4 las compara en el problema típico.

	Huecos	Cadena de sucesores	Acción fusionada
Fidelidad	exacta; sabe qué plato está en qué sitio de la bandeja	exacta para la historia; desaparecen los sitios de la bandeja	no representa la capacidad ni los viajes: cualquier número de platos «a la vez»
Tamaño	22 acciones instanciadas	22 acciones instanciadas	4 acciones instanciadas
Guía	220 estados explorados; cada plan tiene un gemelo con los huecos intercambiados	123 estados explorados; sin simetría	9 estados explorados
Claridad	los planes se leen bien: `(load soup1 s2)`	las cuentas enturbian el plan: `(load soup1 n0 n1)`	el dominio y los planes más cortos
Portabilidad	precondición universal para el orden de los platos	lo mismo	lo mismo

Tabla 7.4.: Tres codificaciones del camarero en `p02` (cuatro platos, dos mesas). El mejor plan tiene 11 pasos en las dos primeras y 4 en la tercera.

Veredicto. La decisión depende de una pregunta que pertenece al paso M: *¿quién decide los viajes?*

- Si el planificador debe decidir qué va en la bandeja y en qué orden se visitan las mesas, la acción fusionada queda descartada: no puede representar esa decisión. De las otras dos, la **cadena de sucesores** es el mejor modelo de esta historia. Los sitios de la bandeja son intercambiables, nada pregunta dónde está cada plato y la cadena explora más o menos la mitad de estados. Una bandeja más grande necesita una cadena más larga en el problema y ningún cambio en el dominio.

- Los **huecos** ganan en cuanto los sitios son distintos: una bandeja con un sitio hondo para los cuencos de sopa, o un robot a cuyo controlador hay que decirle qué brazo lleva qué plato.
- Si es el propio controlador del robot el que agrupa los platos, la **acción fusionada** es el modelo adecuado. Es el más pequeño, y al planificador le deja la única decisión que es realmente suya, el orden de los platos.

El modelo que escribimos primero no era el mejor para la historia tal como se contó. Ese es el resultado normal del paso L, y la razón por la que el método es un bucle.

7.8. Errores comunes

Los antipatrones de la Sección 7.5 son errores en las acciones. Los de esta sección vienen de usar un patrón al que le falta una pieza, normalmente en el fichero de problema.

La mitad de los hechos espejo

Un mapa escrito con un solo hecho por conexión, para un dominio cuyo `move` lee un solo sentido. En `courier-p02.pddl` el robot está en la base de carga, y el único hecho sobre la base es `(connected lobby dock)`:

```
$ kairos courier-domain.pddl courier-p02.pddl
note: the goal needs atoms that no action can reach
(item-at meds1 ward-a) can never be reached:
  drop adds it, but needs (robot-at ?r - robot ward-a), which
     never holds
    move adds it, but is never applicable (types, negative or
     numeric conditions) [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

La nota recorre el camino hacia atrás desde el objetivo: el artículo necesita un `drop`, que necesita al robot en la sala, lo que necesita un `move`, y ningún movimiento es aplicable. **Corrección:** escribe los dos sentidos, o genera el mapa con un programa. Un caso más leve, en el que solo falta un sentido, da un robot que entra en una habitación y no puede salir: los planes existen y dan rodeos extraños.

Dos testigos

Un testigo funciona porque existe una sola vez. En `workshop-p02.pddl` el estado inicial tiene `(now cutting)`, `(now joining)` y `(now painting)`:

```
$ kairos workshop-domain.pddl workshop-p02.pddl
; cost = 9
(cut seat)
(cut leg1)
(cut leg2)
(join seat)
...
```

Nueve pasos en lugar de once, y ningún `advance`: todas las fases están activas, así que las fases no ordenan nada. El plan de arriba resulta ordenado por casualidad; el validador también acepta, para este problema, el plan que termina el asiento antes de cortar las patas.

Síntoma: un plan sin las acciones que pasan el testigo. **Corrección:** exactamente un testigo en `:init`; comprueba que toda acción que añade un testigo borra otro.

Una cadena con un eslabón roto

En una cadena de sucesores el límite es la *ausencia* de un hecho `next`, así que un hecho omitido por error es un límite que nadie quería poner. Una bandeja declarada con `n0 n1 n2` y solo `(next n0 n1)` lleva un plato. Nada falla: el camarero hace más viajes, y el plan para `p02` pasa de 11 pasos a 15. **Síntoma:** planes más largos de lo esperado, con una cuenta que nunca pasa de cierto valor. **Corrección:** revisa la cadena con el mismo cuidado que el mapa, e incluye en la batería de pruebas un problema que necesite toda la capacidad.

Un cerrojo que nadie devuelve

Con un cerrojo de recurso, el planificador solo libera el recurso si alguien lo necesita después. En el plan de `courier-keycard-p01` el robot `r1` termina con la tarjeta encima, allí donde lo haya dejado su última entrega. Si el turno siguiente espera encontrar la tarjeta en su gancho, el modelo es correcto y el problema está incompleto. **Corrección:** pon en el objetivo el estado final de todo recurso compartido, aquí `(keycard-at lobby)`.

7.9. Resumen

Este capítulo no ha añadido sintaxis. Ha añadido vocabulario: nombres para las decisiones que toma todo modelo clásico.

- Un modelo empieza con una **ficha del modelo**. Lo que deja fuera y lo que supone deciden más sobre el PDDL final que cualquier paso posterior.
- Once **patrones** (Tabla 7.3) responden a las preguntas recurrentes de los pasos O y D. Cada uno tiene un rival y un coste; ninguno es siempre el adecuado.
- Cuatro **antipatrones** producen ficheros válidos y modelos erróneos. Sus síntomas: un plan demasiado corto (borrado olvidado), un plan vacío (objetivo en el estado inicial), una prueba pequeña que pasa mientras otra mayor no tiene plan (gemelos desacompasados), demasiadas acciones instanciadas (parámetro ocioso).
- La **depuración** es un procedimiento. Sin plan: lee la nota, relaja el objetivo, quita precondiciones y pregúntate quién debería añadir el hecho que falta. Plan erróneo: busca el primer paso que el mundo rechaza y, después, la precondición o el borrado que falta.
- La salida de `-v` es una herramienta de diagnóstico además de un informe de progreso: el número de acciones instanciadas y los grupos mutex muestran qué ha entendido el planificador del modelo.
- El paso L no es un trámite. En el camarero sustituyó el primer modelo por uno mejor, y mostró que la elección depende de una pregunta del paso M.

Aquí termina la Parte II. Los modelos de la parte siguiente conservan todo lo de esta y añaden lo que la planificación clásica no sabe contar: cantidades.

7.10. Lecturas complementarias

- Haslum et al. (2019) tratan las decisiones de modelado a lo largo de toda su presentación de PDDL, con muchos de los dominios de *benchmark* de las competiciones de planificación.
- McDermott (2000) da cuenta de la primera competición de planificación, celebrada en 1998, cuyos dominios (gripper y logistics, entre otros) se convirtieron en el banco de pruebas común del área y en el origen de varios patrones de este capítulo.
- Helmert (2009) explica cómo encuentra un planificador los grupos mutex y cómo los convierte en variables de estado multivaluadas: la maquinaria que hay detrás de la línea `[sas]` con la que hemos comprobado el invariante de a lo sumo uno.
- Ghallab et al. (2004) presentan la planificación clásica desde el lado de los algoritmos; leerlos después de este capítulo muestra por qué los callejones sin salida y las simetrías hacen difícil un modelo.
- Geffner y Bonet (2013) cubren el mismo terreno en menos páginas, con los modelos que hay detrás de los algoritmos.
- Hoffmann y Nebel (2001) es el origen de la heurística de relajación de borrados en la que se apoyan casi todos los argumentos de Guía de este capítulo.

7.11. Ejercicios

Ejercicios de la Sección 7.1 — Un problema a partir de una historia: el camarero

Ejercicio 7.1.1. Escribe la ficha del modelo (Historia, Agentes, Éxito, Alcance, Supuestos) para esta historia: «*Una cafetera de una oficina hace café solo y café con leche. Necesita agua, café en grano y, para el café con leche, leche; cada depósito da para unas pocas tazas y alguien tiene que rellenarlo. Debe haber una taza bajo la boquilla. La gente hace cola y cada persona quiere una bebida.*» Enumera al menos tres cosas que dejes fuera y di quién o qué se ocupa de cada una.

Ejercicio 7.1.2. Un cliente de la mesa 1 pide sopa, pescado y tarta, en ese orden. Escribe el problema para el dominio del camarero del capítulo. ¿Ha necesitado el dominio algún cambio? ¿Cuántos viajes desde el pase hace el mejor plan, y por qué no menos?

Ejercicio 7.1.3. En el dominio del camarero, nada en `load` dice que el plato se vaya a servir después. ¿Hay aquí una trampa? Piensa en un objetivo que pida solo `(served soup1)` en el problema de cuatro platos, y di qué hace el planificador con los otros tres platos y si el mundo pondría alguna objeción.

Ejercicios de la Sección 7.2 — Patrones para el estado

Ejercicio 7.2.1. Un granjero tiene un zorro, una gallina, un saco de grano y un perro. El zorro se come la gallina y la gallina se come el grano; el perro no se come nada y nadie se lo come a él. Escribe el problema para el dominio del río. ¿Cuántas travesías hacen falta? Después explica qué habrías tenido que cambiar en un dominio con una acción por animal.

Ejercicio 7.2.2. El hospital hace que la conexión entre el vestíbulo y el pasillo sea de un solo sentido para los robots (solo del vestíbulo al pasillo). ¿Cuál de las dos codificaciones del mapa, los hechos

espejo o el `or` en `move`, puede expresarlo, y cómo? ¿Qué pasa con la entrega de `courier-p03`?

Ejercicio 7.2.3. Dale al camarero una bandeja para tres platos, primero en la codificación con huecos y luego con la cadena de sucesores. ¿Qué cambia en cada dominio y en cada problema? Ejecuta `p02` con las dos y compara el número de estados explorados (`kairos -v`). Explica la tendencia.

Ejercicio 7.2.4. Para cada efecto, di si mantiene el invariante de a lo sumo uno de la propiedad que cambia, suponiendo que el invariante se cumplía antes: (a) `(and (not (robot-at ?r ?from)) (robot-at ?r ?to))` con `(robot-at ?r ?from)` en la precondición; (b) el mismo efecto sin esa precondición; (c) `(and (not (now ?a)) (now ?b))` con `(now ?a)` en la precondición y `?a` igual a `?b`; (d) `(carrying ?r ?i)` con `(free ?r)` en la precondición y sin ningún borrado.

Ejercicios de la Sección 7.3 — Patrones para el control

Ejercicio 7.3.1. En el taller, el testigo obliga a hacer todo el corte antes de cualquier ensamblado. Escribe a mano un plan para `workshop-p01` que avance a `joining` después de cortar solo el asiento y una pata, y continúalo hasta donde puedas. ¿Dónde se atasca, y qué dice eso del coste del patrón?

Ejercicio 7.3.2. Quita la acción `wait` del dominio de la patrulla y vuelve a resolver `patrol-p01`. Explica el resultado. Después di qué modela `wait` en el mundo, y por qué olvidarla hace perder soluciones.

Ejercicio 7.3.3. Algunas habitaciones del hospital son estrechas: solo cabe un robot. Amplía el dominio tipado del mensajero con un cerrojo de recurso para las habitaciones estrechas (un hecho estático `(narrow ?l)` y un hecho dinámico que diga que la habitación está ocupada). Pruébalo con dos robots que deban recoger cada uno un artículo de la farmacia, que es estrecha, y con una prueba de la trampa que quiera a los dos robots dentro.

Ejercicios de la Sección 7.4 — Patrones para las acciones

Ejercicio 7.4.1. Clasifica las acciones del dominio del mensajero con carro del Capítulo 6 (`move`, `pick`, `drop`, `load`, `grab-cart`, `release-cart`, `push-cart`) en pares inversos y acciones sin vuelta atrás. Para cada acción, nombra su inversa o explica por qué no la tiene. ¿Tiene el dominio callejones sin salida?

Ejercicio 7.4.2. El mundo de los bloques del Capítulo 2 tiene cuatro acciones que pasan por la mano: coger, dejar, apilar y desapilar. Escribe una versión con acciones fusionadas, en la que un bloque va de donde está a donde debe estar en un solo paso y la mano no se modela. Resuelve con ella la inversión de una torre de tres bloques. ¿Qué puede expresar el modelo de cuatro acciones que este no puede?

Ejercicios de la Sección 7.5 — Antipatrones

Ejercicio 7.5.1. El fichero `gripper-buggy-domain.pddl` modela un robot con dos pinzas que lleva bolas entre dos habitaciones. Pasa `kairos check` y resuelve `gripper-buggy-p01.pddl`. Contiene tres antipatrones de este capítulo. Encuéntralos, primero leyendo y después con `kairos -v` (acciones instanciadas y la línea `[sas]`), y escribe el dominio reparado.

Ejercicio 7.5.2. Un estudiante escribe este objetivo para el laboratorio estéril del Capítulo 6:

```
(and (not (contaminated sample1))
     (not (item-at sample1 corridor)))
```

El planificador responde con un plan de coste 0. Explica qué ha pasado y de qué antipatrón se trata. ¿Por qué los objetivos negativos son especialmente propensos a él?

Ejercicios de la Sección 7.6 — Depurar un modelo

Ejercicio 7.6.1. El problema `sokoban-p03.pddl` se tecleó con prisas: la caja de `c22` debe empujarse hacia arriba hasta `c12`, y Kairos no encuentra plan. Aplica el procedimiento de la sección. ¿Qué dice la nota del planificador, y qué hechos faltan?

Ejercicio 7.6.2. En la sesión de depuración del capítulo, el paso 2 usó tres objetivos relajados. Supón que, en cambio, el objetivo `(served soup1)` por sí solo no tuviera plan. ¿Qué precondiciones quitarías primero, y en qué orden? Describe la estrategia general para elegir la acción y la precondición que relajar.

Ejercicios de la Sección 7.7 — El método MODEL: el método clásico, consolidado

Ejercicio 7.7.1. Aplica el método completo a esta historia y escribe los cinco artefactos: «*Un transbordador lleva coches de una orilla a otra de un río. Caben dos coches como mucho. Hay coches esperando en las dos orillas, y cada uno quiere llegar al otro lado.*» Nombra los patrones que uses. Para el paso L, compara al menos dos codificaciones de la capacidad del transbordador.

Ejercicio 7.7.2. Toma cualquier dominio que hayas escrito para los ejercicios de los tres últimos capítulos. Recorre la Tabla 7.3 y nombra los patrones que usa, el rival de cada uno y un antipatrón al que se acerca. Cambia una decisión y mide el efecto con `kairos -v`.

Parte III. Números: PDDL 2.1

8. Fluentes numéricos

El mensajero de los capítulos anteriores nunca se cansa. Cruza la planta del hospital tantas veces como el planificador le pida, porque nada en el modelo dice que moverse gaste algo. Un robot real funciona con batería: cada metro cuesta carga, una ronda larga tiene que interrumpirse para pasar por la base de carga, y un robot que lo olvida se queda parado en mitad de un pasillo. El Capítulo 7 mostró una forma de contar sin números, con una cadena de objetos de nivel enlazados por `next`. Sirve para «como mucho tres artículos» o «quedan cinco movimientos». No sirve para «el pasillo mide 10 metros, la sala está 12 metros más allá y el robot gasta una unidad y media de carga por metro»: la cadena necesitaría un nivel por cada media unidad, y habría que escribir la aritmética hecho a hecho.

Este capítulo introduce los números en el estado. Con PDDL 2.1 un modelo puede declarar *funciones*: propiedades cuyo valor es un número en lugar de verdadero o falso. Las acciones pueden comparar esos números en sus precondiciones y cambiarlos en sus efectos, con aritmética de por medio. El mundo del mensajero gana una batería que cada movimiento gasta y la base de carga repone, distancias entre habitaciones y una capacidad de carga medida en kilos. A su alrededor, las jarras de agua, un camión con su depósito de combustible, un almacén que cuenta unidades y un laboratorio que diluye muestras muestran las mismas construcciones en otros escenarios.

El capítulo empieza por lo que un hecho no puede decir y cómo lo dice una función. Después, tres secciones construyen el mensajero numérico: la batería, con comparaciones y efectos; las distancias y los pesos, con aritmética sobre datos que nunca cambian; y las demás formas de cambiar un número, con los otros dominios. Dos secciones más cortas estudian qué falla cuando un valor no se ha dado nunca y por qué los números dificultan el trabajo del planificador, hasta el punto de que ningún algoritmo puede hacerlo en general. La sección del método añade las preguntas que hay que hacerle a cualquier magnitud y compara los números con las dos formas de prescindir de ellos.

8.1 Lo que un hecho no puede decir 170
8.2 Funciones: números en el estado 171
8.3 Comparar y cambiar números: la batería . . . 172
Condiciones numéricas . 173
Efectos numéricos 173
Ejecutarlo 174
Ensayar el modelo 177
8.4 Números que nunca cambian: distancias, pesos y capacidad 178
Una capacidad de carga . 179
Otro dominio: un camión y su combustible 180
8.5 Asignar, escalar y preguntar: jarras, diluciones y existencias 181
Las jarras de agua 182
Escalar: una dilución . . 183
Contar: existencias en un almacén 184
8.6 Valores que nunca se dieron 185
8.7 Por qué los números dificultan la planificación 187
8.8 El método MODEL: magnitudes 188
Alternativas comparadas 189
8.9 Errores comunes 191
8.10 Resumen 193
8.11 Lecturas complementarias 194
8.12 Ejercicios 194

8.1. Lo que un hecho no puede decir

Todos los modelos vistos hasta ahora describen un estado como un conjunto de hechos. Un hecho es cierto o no lo es: `(robot-at r1 lab)`, `(open d1)`, `(carrying r1 meds1)`. Es la herramienta adecuada para propiedades con pocos valores posibles. El robot está en una de ocho habitaciones; una puerta está abierta, cerrada o cerrada con llave.

Una batería no encaja. Su carga es una **magnitud**: una propiedad que se mide con un número, que toma muchos valores y sobre la que el mundo hace aritmética. Recorrer 12 metros con un consumo de 1,5 unidades por metro quita 18 unidades, fuera cual fuera la carga anterior. Para decirlo con hechos necesitaríamos un hecho por cada carga posible y, para cada par de cargas que se diferencian en 18, una forma de pasar de una a otra. El modelo sería una tabla de restas escrita a mano.

Hay tres situaciones que piden una magnitud, y el mundo del mensajero las tiene todas.

- **Un recurso que se consume y se repone.** La batería: los movimientos le quitan carga y la base de carga se la devuelve. El combustible, el dinero, el agua de un depósito y el tiempo que queda se comportan igual.
- **Una capacidad que comparten varias cosas.** El robot puede llevar 5 kg. Dos cajas de medicamentos de 2 kg caben juntas; una comida de 4 kg solo cabe sola. Lo que importa es una suma.
- **Datos del problema que son números.** El vestíbulo y el pasillo están a 10 metros. Nada cambia ese valor, pero las acciones lo necesitan para calcular cuánto consumen.

Una magnitud es una propiedad que se mide con un número y sobre la que el mundo hace aritmética: un recurso, una capacidad, una medida. Los hechos solo pueden imitarla con un objeto por valor.

No todo número de una historia es una magnitud para el modelo. «La sala A tiene el número 12» es un nombre. «El robot está en la planta 2» es un lugar entre unos pocos. Un número pasa a ser una magnitud del modelo cuando el planificador tiene que sumarlo, restarlo o compararlo para decidir qué hacer. La sección del método de este capítulo convierte esa observación en preguntas.

Los números llegan con PDDL 2.1

La primera definición de PDDL ya tenía un requisito llamado `:fluents` McDermott et al., 1998, pero no estaba claro qué podían hacer con él los planificadores. Los números entraron a formar parte del lenguaje común con PDDL 2.1, definido por Maria Fox y Derek Long para la tercera Competición Internacional de Planificación, celebrada en 2002 Fox y Long, 2003. PDDL 2.1 se organizaba en niveles: el nivel 1 era el lenguaje clásico de los capítulos anteriores, el nivel 2 añadía los fluentes numéricos y las métricas del plan, y los niveles 3 y 4 añadían el tiempo, tema de la Parte IV. Este capítulo y el siguiente cubren el nivel 2.

8.2. Funciones: números en el estado

Un predicado, aplicado a objetos, da un hecho que es verdadero o falso. Una **función**, aplicada a objetos, da un número. Las funciones se declaran en el dominio, en una sección propia, con la misma sintaxis que los predicados:

```
(:functions
  (battery ?r - robot)             ; charge left, 0 to 100
  (battery-capacity ?r - robot)    ; the most it can hold
  (consumption ?r - robot)         ; charge used per meter
  (distance ?from ?to - location)) ; meters between rooms
```

La declaración dice que cada robot tiene una carga de batería, una capacidad y un consumo, y que cada par ordenado de lugares puede tener una distancia. Una función aplicada a objetos, como `(battery r1)` o `(distance lobby corridor)`, es un **fluente numérico**: una variable del estado que guarda un número. La palabra *fluente* viene de la literatura clásica sobre el razonamiento acerca de acciones, donde designa cualquier cosa cuyo valor puede cambiar de una situación a la siguiente.

Los parámetros de una función llevan tipo como los de un predicado. PDDL 3.1 permite además que una función declare el tipo de su valor, `- number`, como en el Listado 8.3; si se omite, se entiende `number`.

Un estado son ahora dos cosas: el conjunto de hechos que son ciertos, como antes, y un valor para cada fluente numérico. El problema da

los valores iniciales en `:init`, junto a los hechos iniciales, con un signo igual:

```
(= (battery r1) 100)
(= (battery-capacity r1) 100)
(= (consumption r1) 1)
```

`(= (battery r1) 100)` se lee «la batería de `r1` empieza en 100». Los valores pueden tener decimales (`1.5`) y pueden ser negativos. No hay declaración de unidades ni de rango: que la batería sea un porcentaje entre 0 y 100 es intención nuestra, escrita en un comentario, y mantenerlo cierto es trabajo de las acciones.

Una función aplicada a objetos es un fluente numérico, un número en el estado. El dominio declara las funciones; el problema da a cada fluente su valor inicial.

Un dominio que declara funciones debe indicarlo en sus requisitos, con `:numeric-fluents`. PDDL 2.1 llamaba a este requisito `:fluents`, y muchos dominios que circulan siguen usando el nombre antiguo; desde PDDL 3.1, `:fluents` equivale a los fluentes numéricos y los fluentes de objeto (Capítulo 16) juntos. Kairos acepta los dos nombres. Si falta el requisito, planifica igualmente y avisa:

```
warning: uses numeric-fluents but :requirements does not include :
    numeric-fluents [semantic.undeclared_requirement]
```

Las funciones, como los predicados, son de dos clases que el lenguaje no distingue pero el modelador sí debe distinguir (Capítulo 2). Una función **estática** no la cambia ninguna acción: la distancia entre dos habitaciones, la capacidad de la batería, el peso de un artículo. Es un dato del problema. Una función **dinámica** la cambia alguna acción: la carga de la batería. Solo las dinámicas agrandan el estado; las estáticas son una tabla que el planificador lee una vez.

8.3. Comparar y cambiar números: la batería

Partimos del mensajero con tipos del Capítulo 4 y le damos al robot una batería. La ficha del modelo cambia en tres puntos.

Paso M: lo que la batería cambia en la ficha del modelo

Historia. Moverse gasta carga de la batería, en proporción a los metros recorridos. El robot puede recargar en la base de carga, y una recarga llena la batería.
Alcance. Dentro: la carga de cada robot, la longitud de cada conexión, el cargador. Fuera: cuánto tarda una recarga (el tiempo llega en el Capítulo 10), la carga que gastan recoger y dejar, el desgaste de la batería.
Supuestos. El consumo por metro es el mismo lleve lo que lleve el robot. Un robot nunca empieza un movimiento que no puede terminar.

El último supuesto es una decisión sobre qué es un plan válido. Un robot varado entre dos habitaciones no es un estado de nuestro modelo, así que el modelo debe prohibir el movimiento que lo dejaría varado. Eso es una precondición sobre un número.

Condiciones numéricas

Una **condición numérica** compara dos expresiones numéricas con uno de cinco operadores: `<`, `<=`, `=`, `>=`, `>`. Como todo en PDDL, el operador va primero. El movimiento del mensajero pide carga suficiente para cubrir el tramo:

```
      (>= (battery ?r)
          (* (distance ?from ?to) (consumption ?r))))
```

El segundo argumento es una **expresión aritmética**: un número, un fluente numérico o un operador aplicado a dos expresiones. Los operadores son `+`, `-`, `*` y `/`, de nuevo en forma prefija: `(* (distance ?from ?to) (consumption ?r))` es la distancia por el consumo. Una condición numérica puede aparecer en cualquier sitio donde pueda aparecer un hecho: bajo `and`, `or`, `not`, dentro de un cuantificador o de un `when`, y en el objetivo. No hay operador «distinto de»; `(not (= (load ?r) 0))` hace ese papel, y necesita `:negative-preconditions`.

Un signo menos con un solo argumento, `(- (load ?r))`, cambia el signo. Algunos planificadores aceptan sumas y productos de más de dos argumentos; escrita con dos, una expresión se acepta en todas partes.

Efectos numéricos

Un **efecto numérico** cambia el valor de un fluente numérico. Tres operadores cubren casi todos los modelos:

- `(increase f e)` suma el valor de la expresión `e` al fluente `f`;
- `(decrease f e)` lo resta;
- `(assign f e)` sustituye el valor de `f` por el valor de `e`.

El movimiento resta a la batería lo que consume el tramo; la recarga le asigna la capacidad. El Listado 8.1 es el dominio completo. Aparte de las funciones y de esos dos cambios, es el mensajero con tipos, con un predicado más para decir dónde hay un cargador.

```
;; The courier world with a battery: every move consumes
;; charge in proportion to the distance, the dock restores it.
(define (domain courier-battery)
  (:requirements :strips :typing :numeric-fluents)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (has-charger ?l - location))

  (:functions
    (battery ?r - robot)             ; charge left, 0 to 100
    (battery-capacity ?r - robot)    ; the most it can hold
    (consumption ?r - robot)         ; charge used per meter
    (distance ?from ?to - location)) ; meters between rooms

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and
      (robot-at ?r ?from) (connected ?from ?to)
      ; enough charge for the whole stretch
      (>= (battery ?r)
          (* (distance ?from ?to) (consumption ?r))))
    :effect (and
      (not (robot-at ?r ?from)) (robot-at ?r ?to)
      (decrease (battery ?r)
```

```
                (* (distance ?from ?to) (consumption ?r)))))

  (:action charge
    :parameters (?r - robot ?l - location)
    :precondition (and
      (robot-at ?r ?l) (has-charger ?l)
      (< (battery ?r) (battery-capacity ?r)))
    :effect (and
      (assign (battery ?r) (battery-capacity ?r))))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
    :effect (and (not (item-at ?i ?l)) (not (free ?r))
                 (carrying ?r ?i)))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l) (free ?r))))
```

Listado 8.1: El mensajero con batería. El movimiento comprueba la carga y la consume; la recarga llena la batería.

Tres detalles del listado merecen una segunda mirada.

La precondición y el efecto de `move` usan la misma expresión. La precondición garantiza que la resta no dejará una carga negativa. Nada en PDDL ata una a la otra. Si se olvida la precondición, el efecto se aplica igualmente y la batería baja de cero; la Sección 8.9 muestra el plan que sale.

Las expresiones se evalúan en el estado anterior a la acción. Todos los efectos de una acción ocurren a la vez, y cada expresión que contienen lee los valores que había cuando se aplicó la acción. En `move` esto no cambia nada. Sí lo cambiará en las jarras de agua, donde un efecto lee un fluente que otro efecto de la misma acción pone a cero.

La recarga también tiene una precondición numérica. `(< (battery ?r) (battery-capacity ?r))` prohíbe cargar una batería llena. La acción no cambiaría nada, y una acción que no cambia nada es un regalo de estados inútiles para la búsqueda.

Las condiciones numéricas comparan dos expresiones. Los efectos numéricos cambian un fluente con increase, decrease o assign. Todas las expresiones leen el estado anterior a la acción.

Ejecutarlo

El primer problema (Listado 8.2) es la entrega del Capítulo 4 con los números añadidos: las distancias del mapa en los dos sentidos, el cargador y los tres valores del robot. La Figura 8.1 muestra el mapa con ellos.

```
;; Trivial test: one delivery, the battery is more than enough.
(define (problem courier-battery-p01)
  (:domain courier-battery)
  (:objects
    r1 - robot
    lobby corridor pharmacy lab ward-a ward-b icu dock - location
    meds1 - item)
  (:init
    (connected lobby corridor) (connected corridor lobby)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (connected lobby dock) (connected dock lobby)
    (connected corridor lab) (connected lab corridor)
    (connected corridor ward-a) (connected ward-a corridor)
    (connected corridor ward-b) (connected ward-b corridor)
    (connected ward-b icu) (connected icu ward-b)
    ; distances in meters, one line per pair of rooms
    (= (distance lobby corridor) 10)
    (= (distance corridor lobby) 10)
    (= (distance lobby pharmacy) 6)
    (= (distance pharmacy lobby) 6)
```

```
    (= (distance lobby dock) 4) (= (distance dock lobby) 4)
    (= (distance corridor lab) 8) (= (distance lab corridor) 8)
    (= (distance corridor ward-a) 12)
    (= (distance ward-a corridor) 12)
    (= (distance corridor ward-b) 12)
    (= (distance ward-b corridor) 12)
    (= (distance ward-b icu) 5) (= (distance icu ward-b) 5)
    (has-charger dock)
    (robot-at r1 dock) (free r1)
    (= (battery r1) 100)
    (= (battery-capacity r1) 100)
    (= (consumption r1) 1)
    (item-at meds1 pharmacy))
  (:goal (item-at meds1 ward-a)))
```

Listado 8.2: Una entrega con la batería llena. Las distancias forman parte del estado inicial.

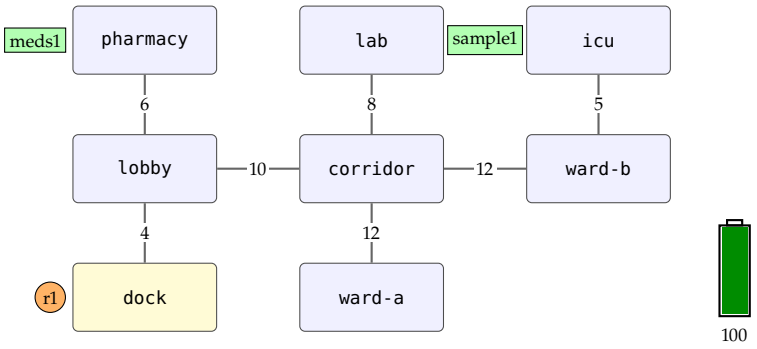


Figura 8.1.: El mundo del mensajero con sus distancias en metros. El número de cada conexión es el valor de `distance` en los dos sentidos. El robot empieza en la base de carga con la batería llena.

Primero comprobamos los ficheros. El informe enumera ahora la característica de este capítulo:

```
$ kairos check courier-battery-domain.pddl courier-battery-p01.pddl
valid, plannable
  typing: full
  numeric-fluents: full
$ kairos courier-battery-domain.pddl courier-battery-p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

El plan es el del Capítulo 4. El robot recorre $4 + 6 + 6 + 10 + 12 = 38$ metros y tiene carga para 100, así que la batería no llega a importar. Es la prueba trivial del paso E: muestra que los números no han roto lo que funcionaba.

El segundo problema hace que la batería importe. Ahora el robot gasta 1,5 unidades por metro y tiene dos entregas: los medicamentos a la sala A y una muestra del laboratorio a la UCI. Además del nuevo objeto `sample1`, cambian tres líneas del problema:

```
    (= (consumption r1) 1.5)
    (item-at meds1 pharmacy) (item-at sample1 lab))
  (:goal (and (item-at meds1 ward-a) (item-at sample1 icu))))
```

```
$ kairos courier-battery-domain.pddl courier-battery-p02.pddl
; cost = 19
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lobby)
(move r1 lobby dock)
(charge r1 dock)
(move r1 dock lobby)
(move r1 lobby corridor)
(move r1 corridor lab)
(pick r1 sample1 lab)
(move r1 lab corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 sample1 icu)
```

Nadie le ha dicho al planificador que recargue. El objetivo solo habla de artículos. La visita a la base de carga en mitad del plan está ahí porque, sin ella, algún movimiento posterior no sería aplicable. Para leer un plan numérico hay que seguir los números, y Kairos puede escribirlos por nosotros: la opción `--dump=plan-trace` escribe en un fichero el estado tras cada paso.

```
$ kairos courier-battery-domain.pddl courier-battery-p02.pddl \
    --no-anytime --dump=plan-trace --dump-dir trace > plan.txt
$ grep -A5 "step 1[01]:" trace/plan-trace.txt
; step 10: (move r1 lobby dock)
  (item-at sample1 lab)
  (item-at meds1 ward-a)
  (robot-at r1 dock)
  (free r1)
  (= (battery r1) 4.000000)
; step 11: (charge r1 dock)
  (item-at sample1 lab)
  (item-at meds1 ward-a)
  (robot-at r1 dock)
  (free r1)
  (= (battery r1) 100.000000)
```

La traza solo enumera lo que puede cambiar; los hechos y valores estáticos se omiten.

La Figura 8.2 dibuja la traza completa. Tras la primera entrega, el robot está en la sala A con 43 unidades. La muestra está a 20 metros y la UCI, 25 más allá: 67,5 unidades. No le llega, así que el robot vuelve. Llega al vestíbulo con 10 unidades y a la base de carga con 4: no habría podido ir a ningún otro sitio. Con la batería llena, la segunda entrega gasta 70,5 unidades y el plan termina con 29,5.

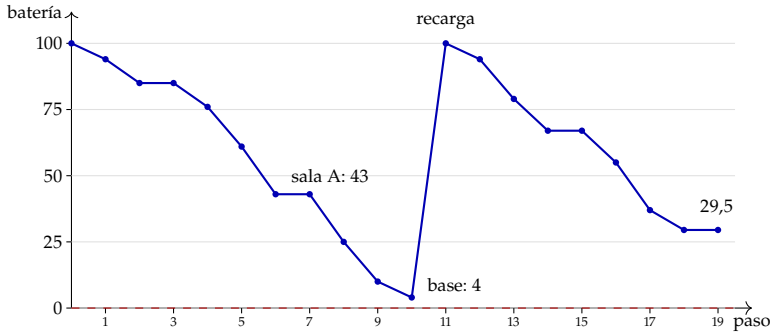


Figura 8.2.: La batería de `r1` a lo largo del plan del segundo problema. Los tramos planos son recogidas y entregas. El robot llega a la base de carga en el paso 10 con 4 unidades; la recarga del paso 11 paga la segunda entrega.

El validador hace la misma aritmética. Si borramos del plan la línea de la recarga y pedimos a Kairos que valide lo que queda, señala el primer movimiento que la batería no puede pagar, con los dos números que ha comparado:

```
$ kairos validate courier-battery-domain.pddl \
    courier-battery-p02.pddl courier-battery-p02-nocharge.plan
courier-battery-p02-nocharge.plan:12: error: step 11 (move r1 dock
     lobby) at 10.000: precondition false: (>= (battery ?r) (* (
     distance ?from ?to) (consumption ?r))): 4 >= 6 is false [
     validate.precondition_failed]
 12 | (move r1 dock lobby)
plan invalid
```

Ensayar el modelo

Dos problemas que funcionan son la mitad del paso E. La otra mitad busca lo que el modelo permite y el mundo no.

Paso E: la batería de pruebas de la carga

Trivial	`p01`: una entrega, carga de sobra. Plan de 7 acciones, sin recarga.
Típica	`p02`: dos entregas. Plan de 19 acciones con una visita a la base de carga, situada donde la traza dice que tiene que estar.
Sin plan	`p03`: el robot está en la UCI con 20 unidades y un artículo para el laboratorio, a 25 metros. No debe haber plan.
Trampa	`p04`: el objetivo es una batería por encima de su capacidad, `(>(battery r1) 100)`. No debe haber plan.

La tercera prueba se supera, y Kairos dice por qué está seguro:

```
$ kairos courier-battery-domain.pddl courier-battery-p03.pddl
note: no plan: every state reachable from the initial one was
     searched (22 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

La cuarta prueba está ahí por un primer borrador de este dominio. En ese borrador, el cargador sumaba una cantidad fija:

```
    :precondition (and (robot-at ?r ?l) (has-charger ?l))
    :effect (and (increase (battery ?r) 50)))
```

Parecía razonable, y los dos problemas de entrega daban planes sensatos. La prueba de la trampa, no:

```
$ kairos courier-battery-naive-domain.pddl \
    courier-battery-naive-p04.pddl
; cost = 1
(charge r1 dock)
```

El robot empezaba con 60, cargaba una vez y tenía 110 unidades en una batería de 100. Con ese dominio, un planificador ante una ronda larga cargaría tres veces seguidas al principio y saldría con 250. Un número no tiene límite superior a menos que una acción se lo imponga. El dominio del Listado 8.1 lo impone asignando la capacidad. Con él, la prueba de la trampa no tiene plan:

```
$ kairos courier-battery-domain.pddl courier-battery-p04.pddl
note: no plan: every state reachable from the initial one was
     searched (681 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Para cada magnitud, prueba sus dos extremos: un problema que necesita más de lo que hay y un objetivo más allá del límite. Los números no tienen cotas propias; solo las precondiciones y assign las mantienen.

8.4. Números que nunca cambian: distancias, pesos y capacidad

Catorce de los diecisiete números del primer problema son distancias. Ninguna acción los cambia: son estáticos, y hacen para los números el papel que `connected` hace para los hechos. La salida detallada de Kairos muestra qué hace con ellos el planificador:

```
$ kairos courier-battery-domain.pddl courier-battery-p02.pddl -v
...
[   0.001s] [ground] grounded in 0.1 ms: 27 atoms, 1 numeric fluents,
      47 actions, 0 durative actions, 0 axioms (97 Datalog facts, 9
     rules, 0 instances pruned)
...
```

Un fluente numérico, no diecisiete. Cuando el planificador instancia la acción `move` para `r1`, el vestíbulo y el pasillo, consulta `(distance lobby corridor)` y `(consumption r1)`, encuentra 10 y 1,5, y se queda con una acción instanciada que necesita 15 unidades y resta 15. Los números estáticos desaparecen dentro de las acciones, como los hechos estáticos en el Capítulo 3. Solo `(battery r1)` queda como variable del estado.

Las funciones estáticas son datos: el planificador las sustituye por sus valores al instanciar. Modelar distancias, pesos o precios como funciones estáticas no cuesta nada durante la búsqueda.

Por eso los datos deben ser datos. Podríamos haber escrito un consumo de 15 en un movimiento especial para el par vestíbulo y pasillo, o una acción `move` por cada longitud de conexión. Con una función estática, el dominio tiene un solo `move`, el mapa está en el problema, y un hospital distinto es un `:init` distinto.

Las distancias se dan en los dos sentidos, con dos líneas por conexión. El lenguaje no tiene forma de decir que una función es simétrica, igual que no la tiene para los predicados (Capítulo 7). Un sentido que falta es una trampa que salta en la Sección 8.6.

Una capacidad de carga

La segunda magnitud del mensajero es lo que transporta. Hasta ahora, el predicado `free` dejaba que el robot llevara un artículo. Un mensajero real tiene una bandeja con un límite de peso. Lo describimos con tres funciones y quitamos `free`: su papel lo asume una comparación. La batería se deja fuera de este dominio para que sea corto; uno de los ejercicios junta las dos cosas.

```
;; The courier world with a load capacity: a robot carries
;; several items at once, up to a total weight in kilograms.
(define (domain courier-load)
  (:requirements :strips :typing :numeric-fluents)

  (:types
    robot location item - object
    medicine sample meal - item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item))

  (:functions
    (load ?r - robot) - number      ; kg it carries now
    (capacity ?r - robot) - number  ; kg it can carry
    (weight ?i - item) - number)    ; kg of one item

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to)))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and
      (robot-at ?r ?l) (item-at ?i ?l)
      ; the item must still fit
      (<= (+ (load ?r) (weight ?i)) (capacity ?r)))
    :effect (and
      (not (item-at ?i ?l)) (carrying ?r ?i)
      (increase (load ?r) (weight ?i))))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and
      (not (carrying ?r ?i)) (item-at ?i ?l)
      (decrease (load ?r) (weight ?i)))))
```

Listado 8.3: El mensajero con capacidad de carga. `pick` comprueba que el artículo todavía cabe; `pick` y `drop` mantienen al día la carga.

La precondición de `pick` compara una suma con una capacidad: `(<= (+ (load ?r) (weight ?i)) (capacity ?r))`. De sus tres fluentes, solo cambia la carga. El peso de cada artículo y la capacidad del robot son estáticos, así que, tras la instanciación, la condición para `meds1` dice «la carga es como mucho 3».

El fluente `(load r1)` es redundante en un sentido preciso: su valor es siempre la suma de los pesos de los artículos que lleva `r1`. PDDL no puede calcular esa suma cuando la necesita, así que el modelo la guarda en un fluente y la actualiza en las dos acciones que cambian lo que lleva el robot. Es la versión numérica de un patrón del Capítulo 7, el hecho que resume otros, y conlleva la misma obligación: toda acción que cambia lo resumido debe actualizar el resumen. Un `increase` en `pick` sin el `decrease` en `drop` da un robot cuya bandeja se llena para siempre.

En el problema del Listado 8.4, tres artículos esperan en el vestíbulo para la sala A. Pesan 8 kg y el robot lleva 5.

```
;; Three items for ward A weigh 8 kg; the robot carries 5 kg.
(define (problem courier-load-p01)
  (:domain courier-load)
  (:objects
    r1 - robot
    lobby corridor pharmacy lab ward-a ward-b icu dock - location
    meds1 meds2 - medicine
    meal1 - meal)
  (:init
    (robot-at r1 lobby)
    (= (load r1) 0) (= (capacity r1) 5)
    (item-at meds1 lobby) (= (weight meds1) 2)
    (item-at meds2 lobby) (= (weight meds2) 2)
    (item-at meal1 lobby) (= (weight meal1) 4))
  (:goal (and (item-at meds1 ward-a) (item-at meds2 ward-a)
              (item-at meal1 ward-a))))
```

```
$ kairos courier-load-domain.pddl courier-load-p01.pddl
; cost = 12
(pick r1 meds2 lobby)
(pick r1 meds1 lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds2 ward-a)
(drop r1 meds1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lobby)
(pick r1 meal1 lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meal1 ward-a)
```

Listado 8.4: Tres artículos, 8 kg en total, para un robot que lleva 5 kg (se omiten las líneas del mapa).

Hacen falta dos viajes, y el planificador ha encontrado la única agrupación con la que bastan dos: las dos cajas de medicamentos juntas (4 kg) y la comida sola. Con `free` habrían sido tres viajes. Si la comida pesa 6 kg en lugar de 4, ningún viaje puede llevarla, y el problema `courier-load-p02.pddl` no tiene plan: Kairos explora los 1296 estados alcanzables y responde `unsolvable`.

Otro dominio: un camión y su combustible

Las mismas dos ideas, un consumo calculado a partir de datos estáticos y un límite mantenido por una precondición, forman el ejemplo numérico más antiguo de la planificación: un vehículo con un depósito de combustible.

```
;; A truck with a fuel tank: driving burns fuel according to
;; the length of the road, stations sell it 10 liters at a time.
(define (domain truck)
  (:requirements :strips :typing :numeric-fluents)

  (:types truck city)

  (:predicates
    (road ?from ?to - city)
    (truck-at ?t - truck ?c - city)
    (has-station ?c - city))

  (:functions
    (fuel ?t - truck)           ; liters in the tank
    (tank ?t - truck)           ; liters the tank holds
```

```
    (km-per-liter ?t - truck)    ; how far one liter takes it
    (distance ?from ?to - city)) ; km of road

  (:action drive
    :parameters (?t - truck ?from ?to - city)
    :precondition (and
      (truck-at ?t ?from) (road ?from ?to)
      (>= (fuel ?t) (/ (distance ?from ?to) (km-per-liter ?t))))
    :effect (and
      (not (truck-at ?t ?from)) (truck-at ?t ?to)
      (decrease (fuel ?t)
                (/ (distance ?from ?to) (km-per-liter ?t)))))

  (:action refuel
    :parameters (?t - truck ?c - city)
    :precondition (and
      (truck-at ?t ?c) (has-station ?c)
      (<= (+ (fuel ?t) 10) (tank ?t)))   ; 10 more must fit
    :effect (increase (fuel ?t) 10)))
```

Listado 8.5: Un camión con depósito de combustible. El combustible se compra de 10 en 10 litros, y solo si cabe.

El consumo es una división: kilómetros entre kilómetros por litro. El repostaje muestra una segunda forma de respetar una capacidad. El cargador del mensajero usaba `assign`: hubiera lo que hubiera, ahora está lleno. La gasolinera suma una cantidad fija, así que necesita una precondición de que esa cantidad cabe: `(<= (+ (fuel ?t) 10) (tank ?t))`. En el problema `truck-p01.pddl` el camión sale de Madrid hacia Valencia pasando por Cuenca, con 170 y 200 km de carretera, 5 litros en un depósito de 60 y 4 km por litro:

```
$ kairos truck-domain.pddl truck-p01.pddl
; cost = 11
(refuel t1 madrid)
(refuel t1 madrid)
(refuel t1 madrid)
(refuel t1 madrid)
(drive t1 madrid cuenca)
(refuel t1 cuenca)
(refuel t1 cuenca)
(refuel t1 cuenca)
(refuel t1 cuenca)
(refuel t1 cuenca)
(drive t1 cuenca valencia)
```

El primer tramo quema 42,5 litros, así que el camión reposta cuatro veces y llega a Cuenca con 2,5. El segundo tramo quema 50: cinco repostajes más, 52,5 litros, y quedan 2,5 en Valencia. Un sexto repostaje en Cuenca no habría cabido en el depósito.

El plan repite una acción muchas veces. Un plan clásico rara vez lo hace, porque hacer lo mismo dos veces sobre los mismos objetos no cambia nada la segunda vez. Con números, cada repetición cambia un poco el estado, y «cuántas veces» se convierte en una cosa más que el planificador tiene que encontrar.

8.5. Asignar, escalar y preguntar: jarras, diluciones y existencias

Quedan por ver en acción dos efectos, `assign` con una expresión y la pareja `scale-up` y `scale-down`, y también los números en el objetivo. Tres dominios pequeños los muestran.

Las jarras de agua

Hay dos jarras sin marcas, de 3 y 5 litros, un grifo y un fregadero. La tarea es medir exactamente 4 litros. Una jarra se puede llenar hasta el borde, vaciar o verter en la otra hasta que la primera quede vacía o la segunda llena. Nada más da una cantidad conocida.

```
;; The water jugs: measure an amount of water with jugs that
;; have no marks. A jug can be filled, emptied or poured.
(define (domain jugs)
  (:requirements :strips :typing :equality
                 :negative-preconditions :numeric-fluents)

  (:types jug)

  (:functions
    (amount ?j - jug)    ; liters in the jug now
    (volume ?j - jug))   ; liters it holds when full

  (:action fill
    :parameters (?j - jug)
    :precondition (< (amount ?j) (volume ?j))
    :effect (assign (amount ?j) (volume ?j)))

  (:action empty
    :parameters (?j - jug)
    :precondition (> (amount ?j) 0)
    :effect (assign (amount ?j) 0))

  ; everything in ?from fits in ?to
  (:action pour-all
    :parameters (?from ?to - jug)
    :precondition (and
      (not (= ?from ?to))
      (> (amount ?from) 0)
      (<= (+ (amount ?from) (amount ?to)) (volume ?to)))
    :effect (and
      (increase (amount ?to) (amount ?from))
      (assign (amount ?from) 0)))

  ; ?to gets full and something stays in ?from
  (:action pour-until-full
    :parameters (?from ?to - jug)
    :precondition (and
      (not (= ?from ?to))
      (< (amount ?to) (volume ?to))
      (> (+ (amount ?from) (amount ?to)) (volume ?to)))
    :effect (and
      (decrease (amount ?from) (- (volume ?to) (amount ?to)))
      (assign (amount ?to) (volume ?to)))))
```

Listado 8.6: Las jarras de agua. Verter son dos acciones, porque lo que ocurre depende de si cabe todo.

Verter tiene dos resultados, y el dominio tiene una acción para cada uno. `pour-all` se aplica cuando el contenido de la jarra de origen cabe en la de destino: el destino lo gana y el origen queda vacío. `pour-until-full` se aplica cuando no cabe: el destino queda lleno, y el origen pierde solo el espacio que quedaba libre, `(- (volume ?to) (amount ?to))`.

Las dos acciones dependen de la regla de que las expresiones leen el estado anterior a la acción. En `pour-all`, el primer efecto suma `(amount ?from)` al destino y el segundo pone `(amount ?from)` a cero. Si los efectos se aplicaran uno tras otro, como las sentencias de un programa, intercambiar las dos líneas sumaría cero. En PDDL, el orden en que se escriben los efectos no significa nada: los dos leen la cantidad antigua.

El objetivo del problema es una condición numérica, `(= (amount big) 4)`:

El acertijo es mucho más antiguo que la planificación, y una versión aparece en una escena de la película *Die Hard with a Vengeance* (1995), con jarras de 3 y 5 galones y una báscula que debe marcar 4.

```
$ kairos jugs-domain.pddl jugs-p01.pddl
; cost = 6
(fill big)
(pour-until-full big small)
(empty small)
(pour-all big small)
(fill big)
(pour-until-full big small)
```

La Figura 8.3 sigue el plan en el espacio de las dos cantidades. El estado no tiene ningún hecho: es un par de números, y el plan es un camino entre los pares que las acciones pueden alcanzar.

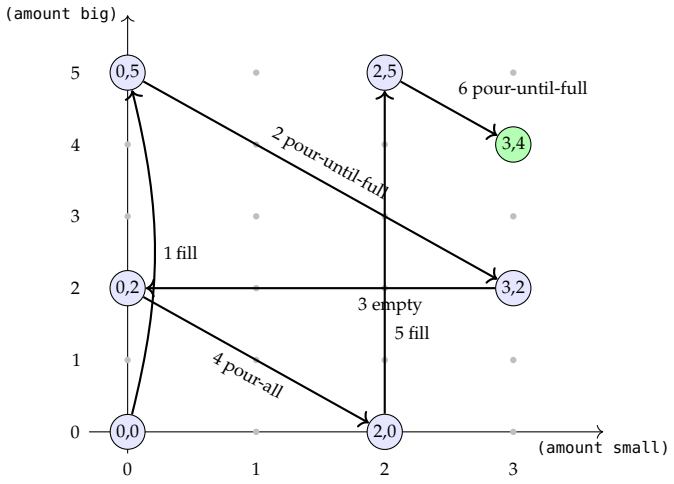


Figura 8.3.: El plan para las jarras de 3 y 5 litros como un camino entre estados. Cada estado es un par (pequeña, grande). Verter mueve en diagonal, porque el total se conserva; llenar y vaciar mueven en línea recta hasta un borde.

Con jarras de 4 y 6 litros y un objetivo de 3, toda cantidad alcanzable es par. Kairos explora los 20 estados de `jugs-p02.pddl`, no encuentra ninguno con 3 litros y responde `unsolvable`. Las cantidades son números, pero solo pueden aparecer unos pocos valores, y una búsqueda que recuerda los estados que ha visto llega a un final. La Sección 8.7 muestra qué ocurre cuando no es así.

Escalar: una dilución

Los dos últimos efectos numéricos multiplican y dividen: `(scale-up f e)` multiplica el fluente `f` por el valor de `e`, y `(scale-down f e)` lo divide. Sirven para cambios que son proporciones: un tipo de interés, una población que se duplica, un descuento. En el laboratorio del hospital, una muestra se diluye añadiendo tanta agua como líquido hay, lo que duplica el volumen y reduce la concentración a la mitad.

```
;; Diluting a sample in the lab: adding as much water as there
;; is liquid doubles the volume and halves the concentration.
(define (domain dilution)
  (:requirements :strips :typing :numeric-fluents)

  (:types tube)

  (:functions
    (concentration ?t - tube)   ; mg per ml
    (volume ?t - tube)          ; ml of liquid in the tube
    (size ?t - tube))           ; ml the tube holds

  (:action dilute
    :parameters (?t - tube)
```

```
    :precondition (<= (* 2 (volume ?t)) (size ?t))
    :effect (and (scale-up (volume ?t) 2)
                 (scale-down (concentration ?t) 2)))

  ; pour half of the liquid away: same concentration, less volume
  (:action discard-half
    :parameters (?t - tube)
    :precondition (>= (volume ?t) 2)
    :effect (scale-down (volume ?t) 2)))
```

Listado 8.7: Diluir una muestra: los dos efectos de escala.

El problema `dilution-p01.pddl` empieza con 2 ml a 80 mg/ml en un tubo de 10 ml y pide una concentración de 10 mg/ml o menos con al menos 4 ml para trabajar. Su objetivo es un par de desigualdades, `(and (<= (concentration t1) 10) (>= (volume t1) 4))`:

```
$ kairos dilution-domain.pddl dilution-p01.pddl
; cost = 4
(dilute t1)
(dilute t1)
(discard-half t1)
(dilute t1)
```

Dos diluciones dan 8 ml a 20 mg/ml. Una tercera necesitaría 16 ml y el tubo admite 10, así que el planificador tira la mitad y vuelve a diluir: 8 ml a 10 mg/ml.

Un objetivo con una desigualdad describe muchos estados a la vez: se acepta cualquier concentración de 10 hacia abajo. Los objetivos numéricos suelen ser de este tipo (que quede al menos tanta carga, que se gaste como mucho tanto). Una igualdad en un objetivo es un blanco mucho más estrecho, y con fracciones puede ser imposible (Sección 8.9).

`(scale-up f 2)` también podría escribirse `(assign f (* f 2))`, y `(increase f e)` como `(assign f (+ f e))`. Los operadores específicos le dicen más al planificador: un `increase` con una constante positiva solo puede hacer crecer el fluente, y las heurísticas lo aprovechan.

Contar: existencias en un almacén

El tercer dominio no tiene robot ni lugares. Solo cuenta. Un almacén prepara un pedido pasando unidades de varios productos de la estantería a una caja.

```
;; Counting in a warehouse: units of each product go from the
;; shelf to a box. Units are counted, they are not objects.
(define (domain warehouse)
  (:requirements :strips :typing :numeric-fluents)

  (:types product)

  (:functions
    (stock ?p - product)    ; units on the shelf
    (batch ?p - product)    ; units that one restocking brings
    (packed ?p - product)   ; units already in the box
    (box-space))            ; units that still fit in the box

  (:action pack
    :parameters (?p - product)
    :precondition (and (>= (stock ?p) 1) (>= (box-space) 1))
    :effect (and (decrease (stock ?p) 1)
                 (increase (packed ?p) 1)
                 (decrease (box-space) 1)))

  ; the supplier only comes when the shelf is empty
  (:action restock
    :parameters (?p - product)
    :precondition (= (stock ?p) 0)
    :effect (increase (stock ?p) (batch ?p))))
```

El dominio tiene un solo tipo, `product`. Las cajas individuales de guantes no son objetos del modelo. Son intercambiables: a ninguna acción le importa qué caja se coge, solo cuántas quedan. Si las hubiéramos hecho objetos, una estantería con 40 cajas daría 40 formas de empaquetar «una caja», todas iguales para el mundo y todas distintas para la búsqueda. Un recuento elimina esa simetría (Capítulo 3) antes de que exista.

Listado 8.8: Un almacén que cuenta. Las unidades de un producto no son objetos: hay un número de ellas.

`(box-space)` es una función sin parámetros: un único número para todo el problema. El pedido del problema `warehouse-p01.pddl` es su objetivo, `(and (= (packed gloves) 3) (>= (packed masks) 2))`, con una unidad de guantes en la estantería, lotes de cuatro y sitio para seis unidades en la caja:

```
$ kairos warehouse-domain.pddl warehouse-p01.pddl
; cost = 6
(pack masks)
(pack masks)
(pack gloves)
(restock gloves)
(pack gloves)
(pack gloves)
```

El planificador empaqueta la única unidad de guantes, llama al proveedor, cosa que el dominio solo permite con la estantería vacía, y empaqueta dos más. Si el pedido pide cuatro unidades de mascarillas además de las tres de guantes, siete unidades no caben en una caja para seis, y `warehouse-p02.pddl` se declara irresoluble tras 66 estados.

Las cosas indistinguibles es mejor contarlas que nombrarlas. Un recuento es un fluente; un conjunto de objetos intercambiables es una simetría que paga la búsqueda.

8.6. Valores que nunca se dieron

Con hechos, lo que el estado inicial no menciona es falso: la hipótesis del mundo cerrado del Capítulo 2. Los números no tienen un valor por defecto así. Un fluente numérico que `:init` no menciona no vale cero. Está **indefinido**: no tiene valor hasta que alguna acción le asigna uno.

PDDL 2.1 es estricto con lo que viene después Fox y Long, 2003: una acción cuya precondición o cuyos efectos necesitan el valor de un fluente indefinido no se puede aplicar. Lo único que se puede hacer con un fluente indefinido es asignarle un valor con `assign`. Así que un valor olvidado no produce un mensaje de error. Produce acciones que, en silencio, nunca se aplican.

Toma el primer problema de la batería y borra la línea que da `(battery r1)`. El comprobador no encuentra nada mal, porque el fichero es PDDL correcto:

```
$ kairos check courier-battery-domain.pddl \
    courier-battery-undefined.pddl
valid, plannable
  typing: full
  numeric-fluents: full
$ kairos courier-battery-domain.pddl courier-battery-undefined.pddl
note: no plan: every state reachable from the initial one was
     searched (2 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

Dos estados: el inicial y aquel en el que el robot recarga, ya que el cargador asigna sin leer. Ningún movimiento llegó a ser aplicable, porque todos comparan una batería que no tiene valor. La respuesta, «irresoluble», es cierta sobre el fichero e inútil sobre el hospital.

En el mapa se esconde un caso peor. Borra una de las catorce distancias del segundo problema, la que va de la sala A de vuelta al pasillo:

```
$ kairos courier-battery-domain.pddl courier-battery-oneway.pddl
; cost = 22
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(drop r1 meds1 corridor)
(move r1 corridor lab)
(pick r1 sample1 lab)
...
(drop r1 sample1 icu)
(move r1 icu ward-b)
(move r1 ward-b corridor)
(pick r1 meds1 corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Hay un plan, tres acciones más largo que antes y extraño de leer: el robot deja los medicamentos en el suelo del pasillo, hace toda la entrega de la muestra y vuelve a por ellos al final. La razón es que la sala A se ha convertido en una habitación sin salida. El hecho `(connected ward-a corridor)` sigue ahí, pero el movimiento por esa conexión necesita una distancia que no tiene valor, así que nunca se aplica. El planificador ha esquivado una puerta de un solo sentido que construimos al olvidar una línea. Ningún mensaje avisa de ello, porque un problema al que le faltan algunas distancias es un problema perfectamente legal.

Un fluente numérico sin valor inicial está indefinido, no vale cero. Las acciones que lo leen nunca se aplican, y nada avisa: el síntoma es un plan que falta o un plan raro.

¿Por qué no cero?

Tomar cero por defecto sería más sencillo y escondería mejor el error. Una distancia cero hace gratis un movimiento; un peso cero hace que un artículo quepa en cualquier parte; un precio cero es una ganga que el planificador encontrará. Un valor indefinido bloquea las acciones que dependen de él, y un modelo que hace demasiado poco se nota más fácilmente que uno que hace demasiado. La regla tiene además una utilidad: un fluente se puede dejar indefinido a propósito, cuando una acción lo va a medir o fijar antes de que nadie lo lea.

Los planificadores difieren en los bordes. Kairos sigue la regla en los modelos clásicos y numéricos, y documenta algunas excepciones para modelos PDDL+ escritos para otros planificadores (Capítulo 17). Un modelo que depende de lo que ocurre con los valores indefinidos no es portable: da un valor a todo fluente que se lea.

Dos hábitos evitan la mayoría de estos errores. El primero es escribir los valores iniciales junto a los hechos con los que van, como hace el problema de la carga con `(item-at meds1 lobby)` y `(= (weight meds1) 2)` en la misma línea, para que un valor que falta se vea como un hueco. El segundo es añadir una prueba de alcanzabilidad a la batería de pruebas del paso E siempre que una función estática proteja una acción: un problema cuyo objetivo sea visitar todas las habitaciones y volver.

8.7. Por qué los números dificultan la planificación

Un problema clásico tiene un número finito de hechos y, por tanto, un número finito de estados: como mucho dos elevado al número de hechos. El Capítulo 3 mostró que ese número es enorme. Pero es finito, y eso da a la búsqueda un suelo firme: un planificador que recuerda los estados que ha visitado, con tiempo y memoria suficientes, o encuentra un plan o se queda sin estados y sabe que no lo hay.

Un fluente numérico puede tomar infinitos valores. Que lo haga o no depende de las acciones. La batería no puede: empieza en 100, pierde una cantidad fija con cada movimiento y vuelve a 100. Las jarras no pueden. El camión del Listado 8.5 tampoco, porque el depósito tiene un tamaño. Si quitamos esa precondición, el combustible puede crecer sin límite:

```
(:action refuel
  :parameters (?t - truck ?c - city)
  :precondition (and (truck-at ?t ?c) (has-station ?c))
  :effect (increase (fuel ?t) 10)))
```

Para ver la diferencia, pedimos a los dos dominios algo imposible. Dos ciudades están unidas por una carretera que quema 40 litros, el camión empieza con 5 y el objetivo es tener exactamente 12 litros. En el depósito solo puede haber 5 más varias veces 10 menos varias veces 40, así que el objetivo es inalcanzable. Con el depósito de 60 litros, el planificador lo demuestra enseguida:

```
$ kairos truck-domain.pddl truck-p03.pddl
note: no plan: every state reachable from the initial one was
    searched (16 states) and none satisfies the goal. [orchestrator.
    why]
kairos: unsolvable (exhausted)
```

Sin el límite, siempre queda un estado más por mirar:

```
$ kairos truck-endless-domain.pddl truck-endless-p03.pddl \
    --timeout 5
note: no plan within the time limit of 5 s (1487655 states searched).
     More time may find one: --timeout (SolveOptions::timeout). [
    orchestrator.why]
kairos: timeout (timeout)
```

Millón y medio de estados en cinco segundos, cada uno con algo más de combustible que otro, y ninguna conclusión. El planificador no puede responder «irresoluble», porque no ha visto todos los estados y nunca los verá. Ni siquiera puede decirnos que más tiempo no servirá de nada. La Figura 8.4 muestra los dos espacios de estados.

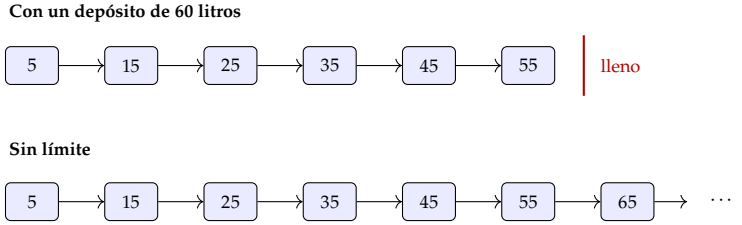


Figura 8.4.: Los valores de combustible que puede alcanzar el repostaje en la primera ciudad (los viajes a la segunda ciudad añaden más estados del mismo tipo). Con depósito, la cadena termina y la búsqueda con ella. Sin límite, siempre hay un estado nuevo.

No es culpa de un planificador concreto.

La planificación numérica es indecidible

En planificación clásica, la pregunta «¿tiene plan este problema?» siempre se puede responder, con un coste que crece muy deprisa (Capítulo 3). Para la planificación con fluentes numéricos, Malte Helmert demostró en 2002 que ningún algoritmo puede responderla en todos los casos Helmert, 2002. La razón es que los fluentes numéricos pueden contar sin cota y, con dos contadores no acotados, una acción que suma uno, una acción que resta uno y una comprobación de cero, un dominio de planificación puede imitar cualquier programa de ordenador. Preguntar si existe un plan equivale entonces a preguntar si un programa termina, la pregunta clásica que ningún algoritmo resuelve.
El mismo artículo traza la frontera: muestra qué combinaciones de comparaciones y efectos mantienen la pregunta decidible y cuáles no, y el lado indecidible empieza con muy poco. Lo que esto significa para un modelador es práctico. Cuando los números de un modelo solo pueden tomar un número finito de valores, como con una batería que tiene capacidad, el planificador pisa terreno clásico y se puede demostrar que un problema es «irresoluble». Cuando pueden crecer para siempre, un planificador puede encontrar un plan, pero su silencio no demuestra nada.

Los valores sin cota son una dificultad. Las otras dos aparecen incluso en modelos acotados.

Más estados. Un hecho duplica el número de estados posibles. Un fluente numérico lo multiplica por el número de valores que puede tomar. El segundo problema de la batería tiene las mismas habitaciones y artículos que uno clásico, y cada situación del robot y los artículos puede darse con muchas cargas distintas. Para demostrar que su plan de 19 acciones es el más corto, Kairos visita varios miles de estados.

Guía más débil. El Capítulo 3 explicó la idea en la que se basan la mayoría de las heurísticas: resolver un problema más fácil en el que los hechos, una vez ciertos, siguen siendo ciertos. Los números necesitan su propia versión de esa relajación, y la habitual deja que un fluente que se puede aumentar crezca tanto como haga falta y olvida que alguna vez se redujo Hoffmann, 2003. En el problema relajado el mensajero nunca se queda sin batería, así que la heurística no ve que hace falta un desvío a la base de carga. Se entera tarde, cuando la búsqueda se topa con movimientos que no son aplicables. Los recursos que se agotan son justo lo que la relajación ignora.

Los números pueden hacer infinito el espacio de estados, y entonces ningún planificador puede demostrar que no hay plan. Las magnitudes acotadas lo evitan, pero aun así multiplican los estados y debilitan las heurísticas.

Para el modelador, las lecciones son estas. Acota toda magnitud que el mundo acota: el depósito tiene un tamaño, la batería una capacidad, la bandeja un límite. Prefiere pocos valores a muchos: una batería que se mueve en pasos de 5 da al planificador menos estados que distinguir que una con dos decimales. Y no descartes prescindir de los números, opción que examina la sección siguiente.

8.8. El método MODEL: magnitudes

Los números entran en el método MODEL en el paso O, cuando enumeramos lo que el planificador debe saber de cada cosa. Hasta ahora, la respuesta para cada propiedad era un predicado, estático o dinámico.

Ahora una propiedad puede ser una magnitud, y tres preguntas deciden cómo codificarla.

Paso O: tres preguntas para cada propiedad en la que hay un número

¿Es una magnitud? ¿Le suma el modelo algo, le resta algo o la compara con otra cantidad? Un número de habitación o una prioridad entre tres es un nombre o un nivel, y basta un predicado o un objeto. Una carga que los movimientos consumen y un peso que se suma son magnitudes.
¿Solo importa un umbral? Si lo único que preguntan las acciones es si hay suficiente para un movimiento más, o si la bandeja está llena, puede que el modelo necesite un hecho (`charged`, `full`) y no el número que hay detrás. Conserva el número cuando acciones distintas consumen cantidades distintas, o cuando un objetivo o una métrica hablan de la cantidad misma.
¿Está acotada? ¿Cuáles son el valor más pequeño y el más grande que permite el mundo, y qué precondición o qué `assign` mantiene el fluente entre ellos? Una cota que el modelo no impone es un plan que el mundo no permite; un fluente sin cota es un espacio de estados sin fin.

La tabla de estado gana una columna para la clase de valor. Para el mensajero de este capítulo:

Propiedad	De qué	Clase	Codificación y cotas
Carga	un robot	dinámica	`(battery ?r)`, de 0 a la capacidad; la mantienen la precondición de `move` y el `assign` de `charge`
Capacidad de batería	un robot	estática	`(battery-capacity ?r)`
Consumo	un robot	estático	`(consumption ?r)`, por metro
Longitud	dos habitaciones	estática	`(distance ?from ?to)`, en los dos sentidos
Carga transportada	un robot	dinámica	`(load ?r)`, de 0 a `(capacity ?r)`; la mantiene `pick`
Peso	un artículo	estático	`(weight ?i)`

Tabla 8.1.: Las filas numéricas de la tabla de estado del mundo del mensajero. Cada magnitud dinámica indica sus cotas y las acciones que las mantienen.

Los pasos D y E heredan una pregunta cada uno. En el paso D, para cada acción y cada magnitud: ¿necesita la acción una parte de ella, y la cambia?; y si la magnitud resume algo, como hace la carga transportada, ¿la actualizan todas las acciones que cambian lo que resume? En el paso E, la batería de pruebas gana, para cada magnitud, las dos pruebas de la Sección 8.3: un problema que necesita más de lo que hay y un objetivo al otro lado de una cota.

Alternativas comparadas

¿Es un número la codificación adecuada para la batería? Para comparar cosas comparables la simplificamos: la batería se cuenta en movimientos. Cada movimiento gasta una unidad sea cual sea su longitud, una batería llena tiene cuatro y la base de carga la rellena. La entrega de la farmacia a la sala A lleva cinco movimientos, así que el robot tiene que recargar por el camino. Lo escribimos de tres maneras.

A, un número. La batería es un fluente numérico. El dominio, en el fichero `moves-number-domain.pddl`, es el del Listado 8.1 con un movimiento más sencillo:

```
   :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                      (>= (battery ?r) 1))
   :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
                (decrease (battery ?r) 1)))
```

B, una cadena de niveles. Es el patrón *cadena de sucesores* del Capítulo 7. Los niveles son objetos, de `n0` a `n4`; unos hechos estáticos `(next n0 n1)` los ordenan, y un predicado dice en qué nivel está la batería de cada robot. Moverse lleva la batería de un nivel al inferior, así que la acción necesita los dos niveles como parámetros (`moves-chain-domain.pddl`):

```
(:action move
  :parameters (?r - robot ?from ?to - location
               ?before ?after - level)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (battery-level ?r ?before)
                     (next ?after ?before))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (battery-level ?r ?before))
               (battery-level ?r ?after)))
```

C, un predicado por nivel. Los niveles se escriben en el dominio como cinco predicados, de `battery-4` a `battery-0`, y unos efectos condicionales (Capítulo 6) pasan de cada uno al siguiente (`moves-flags-domain.pddl`):

```
   :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                      (not (battery-0 ?r)))
   :effect (and
     (not (robot-at ?r ?from)) (robot-at ?r ?to)
     (when (battery-4 ?r)
       (and (not (battery-4 ?r)) (battery-3 ?r)))
     (when (battery-3 ?r)
       (and (not (battery-3 ?r)) (battery-2 ?r)))
```

La Figura 8.5 muestra el mismo estado, con tres unidades restantes, en las tres codificaciones.

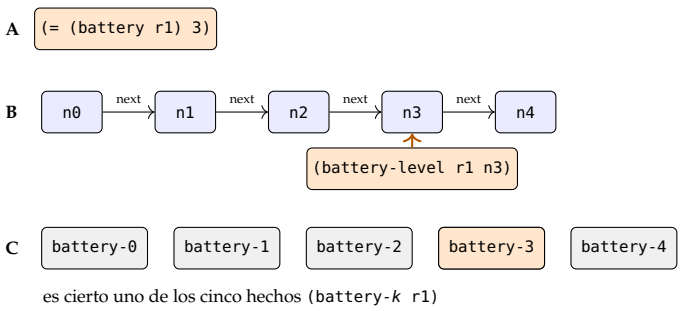


Figura 8.5.: «Quedan tres movimientos» en las tres codificaciones. A es un número. B es un hecho que apunta a uno de cinco objetos ordenados. C es un hecho cierto entre cinco predicados.

Las tres dan el mismo plan de 10 acciones para el mismo problema (el plan de B lleva los niveles como argumentos adicionales, `(move r1 dock lobby n4 n3)`). Difieren en todo lo demás.

	A: número	B: cadena	C: indicadores
Fidelidad	Cualquier cantidad: metros por consumo, fracciones, robots distintos	Solo pasos enteros; un movimiento que gasta 15 unidades necesita cruzar 15 niveles, o una acción por cantidad	Como B, y el número de niveles está fijado en el dominio
Tamaño	31 acciones instanciadas sea cual sea la capacidad; 1 fluente numérico	76 acciones instanciadas con 5 niveles, 1516 con 101: cada movimiento, una vez por nivel	31 acciones instanciadas, cada una con un efecto condicional por nivel
Guía	Finito solo si está acotado; la relajación numérica no ve que se agota	Finito; los niveles de un robot forman una variable de estado que las heurísticas clásicas manejan bien	Como B
Claridad	Se lee como la historia	Dos parámetros más por acción y los hechos `next` en cada problema	Largo; una capacidad nueva obliga a editar el dominio
Portabilidad	Necesita `:numeric-fluents`: no está en los planificadores clásicos	STRIPS y tipos (también usamos una igualdad negada en `charge`)	Necesita efectos condicionales y precondiciones negativas

Tabla 8.2.: Una batería contada en movimientos, codificada de tres maneras. Los tamaños son las acciones instanciadas que indica `kairos -v` para los ficheros `moves-*-p01.pddl` y, para 101 niveles, `moves-chain-p02.pddl`.

Veredicto. Usa un número cuando las cantidades son datos: un consumo que depende de la distancia y del robot, pesos que se suman, cualquier cosa con fracciones o con un rango amplio. Ese es el mensajero de este capítulo, y es la única de las tres codificaciones que sobrevive al paso de «una unidad por movimiento» de vuelta a los metros. Usa una cadena cuando la magnitud es un recuento pequeño que cambia de uno en uno (artículos en una bandeja, plantas, unas pocas cargas), sobre todo si el planificador que tienes a mano es clásico: el modelo sigue siendo finito y se aplican las heurísticas más potentes. Reserva un predicado por nivel para dos o tres niveles con nombre propio (vacía, baja, llena), donde ya es el umbral del paso O y no un recuento.

Sobre la portabilidad, la tabla necesita una nota más larga. Los planificadores clásicos, como Fast Downward y LAMA, no leen fluentes numéricos en absoluto Helmert, 2006; Richter y Westphal, 2010. Los planificadores numéricos y temporales sí los leen; entre ellos, Metric-FF Hoffmann, 2003, ENHSP Scala et al., 2016, POPF Coles et al., 2010 y OPTIC Benton et al., 2012. Quedan pequeñas diferencias. POPF y OPTIC, por ejemplo, rechazan una condición o un efecto numérico que no esté envuelto en `(and ...)`: para ellos, el efecto de `refuel` debe escribirse

```
    :effect (and (increase (fuel ?t) 10))
```

8.9. Errores comunes

El consumo sin la precondición

El error. La acción resta, y nada comprueba que haya suficiente:

```
    :precondition (and
      (robot-at ?r ?from) (connected ?from ?to))
```

El síntoma. Planes demasiado buenos. El problema `p03`, que no tenía

plan porque 20 unidades no pagan 25 metros, ahora tiene uno:

```
$ kairos courier-battery-negative-domain.pddl \
    courier-battery-negative-p03.pddl
; cost = 5
(pick r1 meds1 icu)
(move r1 icu ward-b)
(move r1 ward-b corridor)
(move r1 corridor lab)
(drop r1 meds1 lab)
```

La traza muestra la batería en 15, luego en 3, luego en −5. Un fluente es un número, y los números bajan de cero sin quejarse.
La corrección. Todo efecto que consume necesita una precondición de que hay suficiente, con la misma expresión. La prueba «sin plan» del paso E es la que detecta el olvido.

Un valor inicial olvidado

El error. Un fluente que alguna acción lee no tiene línea en `:init`.
El síntoma. Ningún error. Las acciones que lo leen nunca se aplican (Sección 8.6): o no hay plan y Kairos informa de un puñado de estados explorados, o el plan evita algo sin motivo visible. Cuando el valor olvidado hace inalcanzable el objetivo, la nota puede señalar la acción. Para un problema de carga sin el peso de la comida:

```
note: the goal needs atoms that no action can reach
(item-at meal1 ward-a) can never be reached:
  drop adds it, but needs (carrying ?r - robot meal1), which
     never holds
    pick adds it, but is never applicable (types, negative or
     numeric conditions) [orchestrator.unreachable_goal]
```

La corrección. Da un valor a todo fluente que se lea. Para las funciones estáticas sobre pares, cuenta las líneas: siete conexiones en los dos sentidos son catorce distancias.

La sintaxis de los hechos usada para números

El error. Escribir un valor inicial como si fuera un argumento más de un hecho, o escribir la asignación de un efecto con el signo igual.
El síntoma. Los dos son errores de sintaxis, y el comprobador los señala:

```
$ kairos check courier-battery-domain.pddl \
    courier-battery-wrong-init.pddl
courier-battery-wrong-init.pddl:30:17: error: expected term,
     found number '100' [parse.unexpected_token]
 30 |     (battery r1 100)
    |                 ^
invalid
$ kairos check courier-battery-wrong-domain.pddl
courier-battery-wrong-domain.pddl:39:8: error: expected effect,
     found operator '=' [parse.unexpected_token]
 39 |       (= (battery ?r) (battery-capacity ?r))))
    |        ^
invalid
```

La corrección. El signo igual tiene dos funciones, y los efectos no son una de ellas. En `:init`, `(= (battery r1) 100)` da un valor. En una condición, `(= (amount big) 4)` es una comprobación. Un efecto que fija un valor es `(assign (battery ?r) 100)`.

Igualdad con fracciones

El error. Un objetivo o una precondición que pide que una fracción calculada sea exactamente igual a un decimal. Un gotero añade 0,1 ml cada vez y el objetivo es `(= (dose) 0.3)`.
El síntoma. No hay plan, para un problema que resuelve un niño:

```
$ kairos drip-domain.pddl drip-p01.pddl
note: no plan: every state reachable from the initial one was
     searched (8 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Los planificadores calculan con los números en coma flotante de la máquina, en los que 0,1 no tiene representación exacta: tres sumas de 0,1 dan 0,30000000000000004, y eso no es 0,3.
La corrección. Pide un intervalo, como hace `drip-p02.pddl` con `(and (>= (dose) 0.29) (<= (dose) 0.31))`, que se resuelve con tres gotas. Mejor todavía, elige unidades en las que los números sean enteros: décimas de mililitro, céntimos, porcentajes.

8.10. Resumen

PDDL 2.1 añade números al estado. Una función, declarada en `:functions`, da un fluente numérico para cada tupla de objetos; el problema da su valor inicial. Las condiciones comparan expresiones aritméticas y los efectos cambian un fluente cada vez, leyendo siempre el estado anterior a la acción. Las funciones estáticas son datos que el planificador incorpora a las acciones instanciadas; las dinámicas son variables nuevas del estado. Esta potencia tiene un precio: un modelo cuyos números pueden crecer sin cota tiene un espacio de estados infinito, y para esos modelos ningún planificador puede saber siempre que no hay plan.

Construcción	Significado
`:numeric-fluents`	el requisito (`:fluents` en PDDL 2.1)
`(:functions (f ?x - t))`	declara una función; `- number` detrás es opcional
`(= (f a) 5)` en `:init`	valor inicial; sin él, el fluente está indefinido
`(<e1 e2)`, `<=`, `=`, `>=`, `>`	condiciones numéricas, en precondiciones y objetivos
`(+ e1 e2)`, `-`, `*`, `/`	expresiones aritméticas
`(increase f e)`, `(decrease f e)`	sumar a un fluente, restar de él
`(assign f e)`	fijar un fluente
`(scale-up f e)`, `(scale-down f e)`	multiplicar, dividir un fluente

Tabla 8.3.: Las construcciones numéricas de PDDL 2.1 usadas en este capítulo.

Para el método, el capítulo añade tres preguntas al paso O (¿es una magnitud?, ¿solo importa un umbral?, ¿está acotada?), la obligación de actualizar todo fluente resumen en el paso D y dos pruebas por magnitud en el paso E. El mensajero tiene ya batería, distancias y carga transportada. Pero sigue considerando igual de buenos todos los planes que alcanzan el objetivo: un plan que cruza el hospital dos veces es tan válido como uno que no lo hace. El Capítulo 9 usa los números de este capítulo para distinguir un plan bueno de uno malo.

8.11. Lecturas complementarias

La definición de los fluentes numéricos, con la semántica precisa de condiciones, efectos y valores indefinidos, está en el artículo que introdujo PDDL 2.1 Fox y Long, 2003; sus secciones sobre el nivel 2 son breves y fáciles de leer. La sintaxis completa, tal como se amplió hasta PDDL 3.1, está en la gramática de Kovacs Kovacs, 2011. El libro de texto de Haslum y colaboradores Haslum et al., 2019 dedica un capítulo a la planificación numérica con más ejemplos de modelado.

El resultado de indecidibilidad, y el mapa de qué características numéricas mantienen decidible la planificación, está en Helmert, 2002. Cómo puede una heurística «ignorar las reducciones» se explica en el artículo sobre Metric-FF Hoffmann, 2003, el planificador que extendió a los números la relajación de FF Hoffmann y Nebel, 2001; una relajación posterior y más general, basada en intervalos de valores, es la de ENHSP Scala et al., 2016.

8.12. Ejercicios

Ejercicios de la Sección 8.1 — Lo que un hecho no puede decir

Ejercicio 8.1.1. Para cada propiedad, di si un modelo necesita una magnitud o si bastan los hechos, y por qué: (a) la planta en la que está un ascensor, en un edificio de seis plantas; (b) el dinero de una cartera en un dominio de compras; (c) el número de una cama de hospital; (d) la temperatura de un horno que las recetas necesitan «a 180 grados o más»; (e) si una impresora tiene papel; (f) el número de hojas que le quedan a la impresora, cuando los trabajos de impresión tienen longitudes distintas.

Ejercicios de la Sección 8.2 — Funciones: números en el estado

Ejercicio 8.2.1. Un dron de reparto tiene una altitud en metros, un límite de carga en gramos, y vuela entre puntos separados por unos cuantos metros. Los paquetes tienen un peso. Escribe la sección `:functions` de su dominio con parámetros tipados, y di de cada función si es estática o dinámica.

Ejercicio 8.2.2. Encuentra los tres errores de este fragmento de un problema para el dominio de la batería, y di qué haría un planificador con cada uno:

```
(:init
  (robot-at r1 dock) (free r1)
  (battery r1 80)
  (= (battery-capacity r1) 100)
  (= (distance lobby dock) 4)
  (= (consumption) 1))
```

Ejercicios de la Sección 8.3 — Comparar y cambiar números: la batería

Ejercicio 8.3.1. En el segundo problema de la batería, el robot empieza con la batería llena. Antes de ejecutar nada, predice el plan si empieza con 50 unidades: ¿cuántas veces cargará, y cuándo? Después escribe el problema y comprueba tu predicción.

Ejercicio 8.3.2. Los ingenieros del robot piden que la batería no baje nunca de una reserva de 10 unidades, guardada para emergencias. Cambia el dominio para que ningún movimiento deje al robot por debajo de su reserva, con la reserva como función del robot. Ejecuta con él el segundo problema. ¿Por qué deja de ser válido el plan de la Sección 8.3, y qué hace el planificador en su lugar?

Ejercicio 8.3.3. Escribe un problema como el segundo en el que el robot deba terminar, además, con al menos la mitad de su carga. ¿Necesitas cambiar el dominio? ¿Cómo cambia el plan?

Ejercicios de la Sección 8.4 — Números que nunca cambian

Ejercicio 8.4.1. Junta la batería y la carga en un solo dominio, y haz que el consumo dependa de la carga: un robot gasta un 2 % más de batería por metro por cada kilo que lleva. Pruébalo con un robot que lleva una comida de 5 kg y unos medicamentos de 2 kg del vestíbulo a la UCI, y calcula a mano la carga de batería con la que llega.

Ejercicio 8.4.2. El primer problema de la batería da 17 valores numéricos en su `:init` y `kairos -v` indica «1 numeric fluents». Explica la diferencia. ¿Cuántos indicaría para el problema de carga del Listado 8.4, y por qué?

Ejercicio 8.4.3. Escribe un problema para el dominio del camión en el que la carretera directa al destino sea demasiado larga para el combustible que lleva y la única gasolinera esté en una tercera ciudad, fuera de la ruta. Predice el plan y su longitud.

Ejercicios de la Sección 8.5 — Asignar, escalar y preguntar

Ejercicio 8.5.1. Una variante clásica de las jarras no tiene grifo ni fregadero: una jarra de 8 litros está llena de vino, dos jarras de 5 y 3 litros están vacías, y hay que repartir el vino en dos partes iguales. Escribe un dominio sin `fill` ni `empty` y el problema. ¿Cuánto mide el plan más corto?

Ejercicio 8.5.2. Sustituye las dos acciones de verter de las jarras por una sola acción `pour` con efectos condicionales (Capítulo 6). ¿Cuál de las dos versiones prefieres, y por cuáles de los cinco criterios?

Ejercicio 8.5.3. En el dominio de la dilución, `discard-half` exige al menos 2 ml en el tubo. ¿Qué podría ir mal, para el planificador y no para el químico, si se quitara esa precondición?

Ejercicio 8.5.4. Modela un ascensor que lleva personas entre plantas y admite como mucho un número dado de ellas. Decide qué es un hecho y qué es un número. Pruébalo con tres personas que van de la planta baja a la segunda en un ascensor para dos.

Ejercicios de la Sección 8.6 — Valores que nunca se dieron

Ejercicio 8.6.1. Borra el peso de la comida del problema del Listado 8.4. Predice qué dirán `kairos check` y `kairos`, y después pruébalo.

¿Qué habría pasado si los valores indefinidos contaran como cero?

Ejercicio 8.6.2. Un colega propone evitar las distancias olvidadas declarando un solo sentido, `(distance lobby corridor)`, y escribiendo dos acciones `move`, una que lee `(distance ?from ?to)` y otra que lee `(distance ?to ?from)`. Juzga la propuesta con los cinco criterios.

Ejercicios de la Sección 8.7 — Por qué los números dificultan la planificación

Ejercicio 8.7.1. ¿Cuáles de estos modelos tienen un número finito de estados alcanzables, sea cual sea el problema? (a) El dominio de la batería del Listado 8.1. (b) El almacén del Listado 8.8. (c) El almacén sin la precondición de `restock`. (d) El dominio de la dilución. (e) El dominio de la dilución sin la precondición de `discard-half`.

Ejercicio 8.7.2. Ejecuta `truck-endless-p03.pddl` con límites de tiempo de 1, 2 y 4 segundos y apunta los estados explorados. ¿Qué tienen en común las tres respuestas? Explica a un colega por qué «el planificador no encontró nada en una hora» no significa que el modelo tenga un fallo.

Ejercicios de la Sección 8.8 — El método MODEL: magnitudes

Ejercicio 8.8.1. Codifica la capacidad del mensajero sin números, con el patrón de los huecos del Capítulo 7: el robot tiene dos huecos y cada artículo ocupa uno. Ejecuta con él el problema de los tres artículos para la sala A y aplica el paso L frente al dominio del Listado 8.3: rellena una tabla con los cinco criterios y da un veredicto.

Ejercicio 8.8.2. En la codificación B de la Sección 8.8, cada movimiento cuesta un nivel. ¿Cómo harías que las dos conexiones largas (12 metros) costaran dos niveles y el resto uno, todavía sin números? ¿Qué efecto tiene en el tamaño?

Ejercicio 8.8.3. *De una historia a un modelo.* Un robot de riego en un invernadero lleva un depósito de 6 litros, que llena en un grifo. Cada bancal de plantas necesita un número de litros, y el robot da el agua en dosis de 2 litros, nunca más de lo que necesita un bancal. Aplica los pasos M, O y D (escribe los cambios en la ficha del modelo, la tabla de estado y la tabla de acciones), escribe el dominio y un problema con dos bancales que necesitan 6 y 4 litros, y ejecuta las cuatro pruebas del paso E.

Calidad del plan: métricas y costes de acción

9.

En el vestíbulo espera una comida destinada al mostrador del pasillo. Un robot está en la farmacia, a 6 metros de la comida; otro está ya en el pasillo, a 10 metros. Cualquiera de los dos puede hacer el trabajo con cuatro acciones: ir al vestíbulo, coger la comida, ir al pasillo y dejarla. Para todos los modelos que hemos escrito hasta ahora, los dos planes son el mismo plan. Ambos alcanzan el objetivo, y un planificador al que se le pide «un plan» puede devolver cualquiera de ellos. Para el hospital no son lo mismo: un robot recorre 16 metros y el otro 20. El Capítulo 8 dio números al modelo, y el plan mantuvo la batería dentro de sus límites. Pero nada en un problema ha dicho todavía que menos es mejor.

Este capítulo añade esa frase. Un problema puede terminar con una *métrica*: una expresión que asigna un número a cada plan, con la instrucción de hacerlo lo más pequeño, o lo más grande, posible. Las métricas más sencillas suman un coste que declara cada acción, y PDDL tiene para eso exactamente una característica pequeña y muy extendida, los costes de acción. Las métricas más generales leen cualquier fluente numérico del Capítulo 8 y pueden ponderar varios en una misma suma. El mensajero minimizará los metros que recorre, después la energía que gasta y después una mezcla de ambas; una cocina cocinará con los ingredientes más baratos; un coche elegirá entre autopistas de peaje y carreteras gratuitas.

Una métrica cambia lo que le pedimos al planificador, así que el capítulo también mira cómo responde. Veremos a Kairos imprimir un plan, después uno mejor y después otro mejor todavía, y aprenderemos qué significa que diga que el último es óptimo y cómo indicarle que con algo bueno basta. El capítulo termina con la pregunta que el método añade en su primer paso, qué hace que un plan sea mejor que otro, y con dos comparativas: tres maneras de decir «mejor», y un límite frente a una métrica.

9.1 Cuando no basta con un plan cualquiera 197
9.2 Costes de acción 198
El requisito y sus reglas . 200
9.3 Costes a partir de datos: una cocina con precios . 200
Acciones que no cuestan nada 202
9.4 El planificador en acción: búsqueda anytime 203
Qué significa `proven optimal` 204
Lo bastante bueno 205
9.5 Métricas sobre cualquier fluente 206
Maximizar 208
9.6 Varios deseos en un solo número 209
Un viaje por carretera: peajes y combustible . . . 210
9.7 Un límite o una métrica 212
9.8 El método MODEL: qué hace mejor a un plan . . 213
Alternativas comparadas 213
9.9 Errores comunes 215
9.10 Resumen 217
9.11 Lecturas complementarias 217
9.12 Ejercicios 218

9.1. Cuando no basta con un plan cualquiera

Los planificadores del Capítulo 3 se describieron como *satisfactorios*: se detienen en el primer plan que encuentran. Cuál sea ese plan depende del orden en que la búsqueda vaya mirando las cosas. Esta es la situación del párrafo inicial como problema para el mensajero con tipos, sin ningún número en lo que se le pide al planificador:

```
$ kairos courier-cost-domain.pddl courier-cost-p01-nometric.pddl
; cost = 4
(move r2 corridor lobby)
(pick r2 meal1 lobby)
(move r2 lobby corridor)
(drop r2 meal1 corridor)
```

El planificador ha elegido a `r2`, el robot que recorre 20 metros. No es un error. Fíjate en la primera línea: `; cost = 4`. Desde el Capítulo 2 ese comentario ha encabezado cada plan, y siempre ha sido el número de acciones. Cuando un problema no dice nada de la calidad, Kairos

toma como coste la **longitud del plan**, el número de acciones de un plan secuencial. Con esa medida, el plan de arriba es todo lo bueno que puede ser un plan, y también lo es el de `r1` (Figura 9.1).

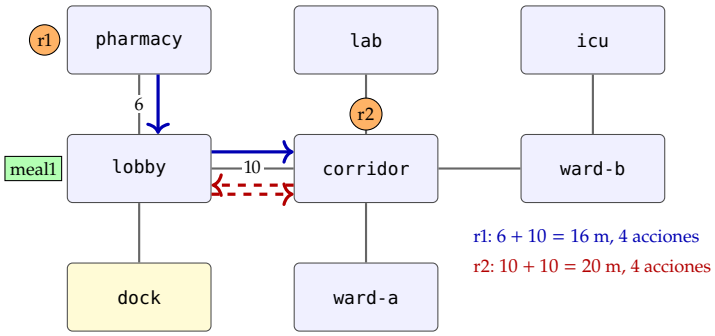


Figura 9.1.: Dos planes para la comida. Tienen la misma longitud, cuatro acciones, y se diferencian en los metros recorridos. Solo un modelo que cuente metros puede preferir uno.

La longitud del plan es un valor por defecto razonable. Los planes cortos son fáciles de leer y de comprobar, y en los rompecabezas de los primeros capítulos, donde cada movimiento vale lo mismo que otro, «mejor» significa precisamente menos movimientos. Deja de ser razonable en cuanto las acciones se diferencian en lo que le cuestan al mundo: un desplazamiento de 4 metros y otro de 12, un ingrediente de un euro y otro de cinco, una carretera de peaje y una gratuita. Entonces el número de acciones mide la longitud del texto del plan, y nada que le importe a nadie.

Sin métrica, el coste de un plan secuencial es su número de acciones. Esa es la medida correcta solo cuando todas las acciones valen lo mismo.

Para hacerlo mejor hacen falta dos cosas. El modelo tiene que decir lo que cuesta un plan, y el planificador tiene que buscar planes baratos y no solo planes. Lo primero es el tema de las secciones siguientes; lo segundo, de la Sección 9.4.

9.2. Costes de acción

La idea de calidad más común es una suma: cada acción tiene un coste, y el coste de un plan es la suma de los costes de sus acciones. Los caminos mínimos en un grafo son el ejemplo más antiguo, y el mensajero que minimiza metros es uno más.

PDDL lo escribe con lo que ya conocemos. El dominio declara una función sin parámetros, `(total-cost)`; cada acción que cuesta algo la incrementa; el problema la inicia a cero y pide minimizarla. El Listado 9.1 es el mensajero con tipos del Capítulo 4 con esos añadidos.

```
;; The courier world with action costs: every move costs the
;; meters it covers, and the plan is judged by their sum.
(define (domain courier-cost)
  (:requirements :strips :typing :action-costs)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot))

  (:functions
    (total-cost) - number                    ; meters so far
    (distance ?from ?to - location) - number) ; static data
```

```
18
19  (:action move
20    :parameters (?r - robot ?from ?to - location)
21    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
22    :effect (and (not (robot-at ?r ?from))
23                 (robot-at ?r ?to)
24                 (increase (total-cost) (distance ?from ?to))))
25
26  ; picking and dropping cover no distance: they cost nothing
27  (:action pick
28    :parameters (?r - robot ?i - item ?l - location)
29    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
30    :effect (and (not (item-at ?i ?l)) (not (free ?r))
31                 (carrying ?r ?i)))
32
33  (:action drop
34    :parameters (?r - robot ?i - item ?l - location)
35    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
36    :effect (and (not (carrying ?r ?i))
37                 (item-at ?i ?l) (free ?r))))
```

Listado 9.1: El mensajero con costes de acción. Un desplazamiento cuesta sus metros; coger y dejar no cuestan nada.

El coste de `move` no es una constante: se lee de la función estática `distance` del Capítulo 8, de modo que cada desplazamiento instanciado tiene su propio coste. El problema añade tres cosas a su `:init` y a su final: las distancias, el valor inicial del coste y una sección nueva después del objetivo.

```
    (= (total-cost) 0)
    (robot-at r1 pharmacy) (free r1)
    (robot-at r2 corridor) (free r2)
    (item-at meal1 lobby))
  (:goal (item-at meal1 corridor))
  (:metric minimize (total-cost)))
```

La **métrica** es la sección `(:metric minimize` *`expression`*`)`. Pertenece al problema, no al dominio: el dominio dice lo que cada acción suma al coste, y cada problema dice si le importa. Un plan se evalúa calculando la expresión en el estado alcanzado al final del plan; un plan con un valor menor es un plan mejor. Con la métrica, la respuesta cambia:

```
$ kairos courier-cost-domain.pddl courier-cost-p01.pddl
; cost = 20
(move r2 corridor lobby)
(pick r2 meal1 lobby)
(move r2 lobby corridor)
(drop r2 meal1 corridor)
; better plan 2
; cost = 16
(move r1 pharmacy lobby)
(pick r1 meal1 lobby)
(move r1 lobby corridor)
(drop r1 meal1 corridor)
```

Dos planes. El primero es el que ya vimos, y su coste es ahora 20, el valor de la métrica, no 4. Kairos no se detuvo ahí: siguió buscando, encontró el plan con `r1` y lo imprimió después del comentario `; better plan 2`. El último plan impreso es el mejor. La Sección 9.4 explica esta forma de trabajar.

Un coste de acción es un incremento de total-cost en el efecto de una acción. El problema lo inicia a cero y termina con (:metric minimize (total-cost)). El coste que se imprime con cada plan es entonces el valor de la métrica.

El validador también calcula la métrica. Para el primer plan, guardado en un fichero:

```
$ kairos validate courier-cost-domain.pddl courier-cost-p01.pddl \
    courier-cost-p01-r2.plan
plan valid, makespan 4, metric 20
```

(En un plan secuencial, lo que el validador llama *makespan* es el número de pasos; la palabra adquiere su significado real en el Capítulo 10.) Un plan válido con una métrica pobre sigue siendo un plan válido. La validación dice que un plan funciona, nunca que sea bueno.

El requisito y sus reglas

El listado declara `:action-costs` y no `:numeric-fluents`. Los costes de acción son un fragmento de la planificación numérica deliberadamente pequeño, definido para la competición de planificación de 2008 como parte de PDDL 3.1 Kovacs, 2011. Un dominio que declara solo `:action-costs` acepta estas reglas:

- el único fluente numérico que cambia es `(total-cost)`, una función sin parámetros;
- solo cambia mediante `(increase (total-cost) e)`, donde `e` es un número no negativo o una función estática aplicada a los parámetros de la acción, como `(distance ?from ?to)`;
- ninguna precondición ni ningún objetivo lee un número;
- la métrica es `(:metric minimize (total-cost))`.

Con estas reglas, el coste nunca influye en qué acciones son aplicables. El espacio de estados es el clásico, finito, con los mismos estados y los mismos planes que sin costes. Los costes solo ponen un peso en cada transición. Por eso esta característica está tan extendida: un planificador clásico no necesita ninguna maquinaria numérica para ella, solo una búsqueda que sume pesos. Es también el motivo para preferirla cuando basta, como dirá la fila de Portabilidad de la Sección 9.8.

Kairos 1.0.6 trata los costes de acción como un caso de fluentes numéricos, y su comprobador imprime un aviso para un dominio que declara solo `:action-costs`: *uses numeric-fluents but :requirements does not include :numeric-fluents*. El dominio es PDDL 3.1 correcto y se planifica con normalidad. Añadir `:numeric-fluents` hace desaparecer el aviso.

Los costes entran en la competición

Las métricas llegaron con PDDL 2.1 en 2002 Fox y Long, 2003, como expresiones sobre fluentes numéricos, y la competición de 2006 situó la calidad del plan en el centro de su evaluación A. E. Gerevini et al., 2009. Pero los planificadores clásicos, que no manejan números, se quedaron fuera de todo aquello. La competición de 2008 introdujo `:action-costs` para ellos, y desde entonces sus categorías clásicas puntuaron a los planificadores por el coste de sus planes y no por su longitud. El ganador de la categoría satisfactoria de aquel año fue LAMA, de Silvia Richter y Matthias Westphal, que encontraba rápidamente un primer plan y después usaba el tiempo restante en buscar otros más baratos Richter y Westphal, 2010: el comportamiento *anytime* que este capítulo muestra con Kairos.

9.3. Costes a partir de datos: una cocina con precios

En el mensajero, la elección que decide la métrica es quién va. En el siguiente dominio es qué se usa. Una cocina prepara platos. Una receta pide un ingrediente de cada una de ciertas clases (un huevo, una grasa, un

edulcorante), y sirve cualquier ingrediente de la clase: aceite o mantequilla para la grasa, azúcar o miel para el edulcorante. Cada ingrediente tiene un precio.

```
;; Cooking with prices: a dish needs one ingredient of each
;; kind of its recipe, and any ingredient of the kind will do.
;; Using an ingredient costs its price; cooking costs the gas.
(define (domain kitchen)
  (:requirements :strips :typing :action-costs)

  (:types
    ingredient dish - object
    egg fat sweetener - ingredient
    omelette cake - dish)

  (:predicates
    (in-stock ?i - ingredient)
    (with-egg ?d - dish)
    (with-fat ?d - dish)
    (with-sweetener ?d - dish)
    (ready ?d - dish))

  (:functions
    (total-cost) - number
    (price ?i - ingredient) - number)   ; static data

  (:action add-egg
    :parameters (?e - egg ?d - dish)
    :precondition (in-stock ?e)
    :effect (and (not (in-stock ?e)) (with-egg ?d)
                 (increase (total-cost) (price ?e))))

  (:action add-fat
    :parameters (?f - fat ?d - dish)
    :precondition (in-stock ?f)
    :effect (and (not (in-stock ?f)) (with-fat ?d)
                 (increase (total-cost) (price ?f))))

  (:action add-sweetener
    :parameters (?s - sweetener ?d - cake)
    :precondition (in-stock ?s)
    :effect (and (not (in-stock ?s)) (with-sweetener ?d)
                 (increase (total-cost) (price ?s))))

  (:action fry
    :parameters (?d - omelette)
    :precondition (and (with-egg ?d) (with-fat ?d))
    :effect (and (ready ?d) (increase (total-cost) 1)))

  (:action bake
    :parameters (?d - cake)
    :precondition (and (with-egg ?d) (with-fat ?d)
                       (with-sweetener ?d))
    :effect (and (ready ?d) (increase (total-cost) 2))))
```

Listado 9.2: Una cocina con precios. Los tipos dicen qué ingrediente puede desempeñar qué función; cada acción `add` cuesta el precio de lo que usa, y cocinar cuesta una cantidad fija.

El dominio muestra las dos formas de coste una junto a otra. Las acciones que usan un ingrediente cuestan su precio, una función estática del ingrediente: `(increase (total-cost) (price ?f))`. Las dos acciones de cocinar cuestan una constante, el gas: 1 por freír y 2 por hornear. Los tipos del Capítulo 4 hacen el trabajo de las recetas: `add-fat` acepta cualquier objeto de tipo `fat`, y el planificador elige cuál.

La despensa del problema tiene versiones baratas y caras de todo (Listado 9.3).

```
;; An omelette and a cake with what the pantry has: which
;; ingredients make the cheapest meal?
(define (problem kitchen-p01)
  (:domain kitchen)
```

```
  (:objects
    om1 - omelette
    cake1 - cake
    farm-egg egg1 egg2 - egg
    butter1 oil1 oil2 - fat
    honey1 sugar1 - sweetener)
  (:init
    (= (total-cost) 0)
    (in-stock farm-egg) (= (price farm-egg) 3)
    (in-stock egg1) (= (price egg1) 1)
    (in-stock egg2) (= (price egg2) 1)
    (in-stock butter1) (= (price butter1) 4)
    (in-stock oil1) (= (price oil1) 2)
    (in-stock oil2) (= (price oil2) 2)
    (in-stock honey1) (= (price honey1) 5)
    (in-stock sugar1) (= (price sugar1) 1))
  (:goal (and (ready om1) (ready cake1)))
  (:metric minimize (total-cost)))
```

```
$ kairos kitchen-domain.pddl kitchen-p01.pddl
; cost = 18
(add-egg farm-egg cake1)
(add-egg egg1 om1)
(add-fat butter1 cake1)
(add-fat oil1 om1)
(add-sweetener honey1 cake1)
(bake cake1)
(fry om1)
; better plan 2
; cost = 10
(add-fat oil1 cake1)
(add-fat oil2 om1)
(add-egg egg1 cake1)
(add-egg egg2 om1)
(add-sweetener sugar1 cake1)
(bake cake1)
(fry om1)
```

Listado 9.3: Una despensa: cada ingrediente es un objeto con su precio. El objetivo es una tortilla y un bizcocho.

Los dos planes tienen siete acciones, y todos los planes de este problema tienen siete: dos platos necesitan cinco ingredientes y dos cocciones. La longitud del plan no puede distinguirlos. El primer plan toma lo primero que encuentra, el huevo de granja, la mantequilla y la miel, y cuesta 18. El segundo usa los dos huevos normales (2), las dos botellas de aceite (4) y el azúcar (1), más 3 de gas: 10. Con esta despensa no hay una comida más barata.

Acciones que no cuestan nada

En el mensajero, `pick` y `drop` no tienen `increase`: cuestan cero. Es lo que dice la historia, porque no recorren ninguna distancia, pero los costes cero merecen pensarse un momento. Para una búsqueda que solo mira los costes, una acción de coste cero es gratis, y un plan con diez acciones gratuitas inútiles es exactamente igual de bueno que el mismo plan sin ellas. Los planificadores toman precauciones: Kairos, como LAMA, deja que sus heurísticas vean cada acción como si costara uno más de lo que cuesta mientras mejora un plan, y aun así compara los planes por su coste real. Como modelador, pregúntate si las acciones gratuitas son de verdad gratuitas. Si cada manipulación de un artículo necesita que una enfermera venga a ayudar, un coste pequeño en `pick` y `drop` lo dice, y el Ejercicio 9.2.1 muestra lo que cambia.

Los costes pueden ser constantes o funciones estáticas de los parámetros. Una acción sin incremento cuesta cero: asegúrate de que eso es lo que dice el mundo.

9.4. El planificador en acción: búsqueda anytime

Encontrar el mejor plan es más difícil que encontrar un plan. Para estar seguro de que un plan es el más barato, un planificador tiene que descartar cualquier plan más barato, y eso puede significar explorar mucho más espacio de estados del que necesitó el primer plan. Un **planificador óptimo** hace ese trabajo antes de responder, y puede que no llegue a responder en un problema que un planificador satisfactorio resuelve en milisegundos. La **búsqueda anytime** es el término medio práctico: devolver un primer plan lo antes posible y seguir buscando después uno estrictamente mejor, y otro, hasta que no quede ninguno o se acabe el tiempo. Se detenga cuando se detenga, el planificador tiene a mano el mejor plan encontrado hasta entonces.

Kairos funciona así por defecto. Después del primer plan, reinicia su búsqueda varias veces, cada vez menos voraz que la anterior y aceptando solo planes más baratos que el mejor conocido. Un problema con más cosas que decidir lo muestra bien. En el Listado 9.4, dos robots se reparten una ronda matinal de cuatro entregas.

```
;; A morning round: four deliveries for two robots. Many plans
;; solve it; the metric asks for the one with fewest meters.
(define (problem courier-cost-p02)
  (:domain courier-cost)
  (:objects
    r1 r2 - robot
    lobby corridor pharmacy lab ward-a ward-b icu dock - location
    meds1 meds2 sample1 meal1 - item)
    (= (total-cost) 0)
    (robot-at r1 dock) (free r1)
    (robot-at r2 icu) (free r2)
    (item-at meds1 pharmacy) (item-at meds2 pharmacy)
    (item-at sample1 ward-a) (item-at meal1 lobby))
  (:goal (and (item-at meds1 ward-a) (item-at meds2 icu)
              (item-at sample1 lab) (item-at meal1 ward-b)))
  (:metric minimize (total-cost)))
```

Listado 9.4: Una ronda de cuatro entregas para dos robots (se omiten el mapa y las distancias).

La salida tiene cinco planes de unas treinta acciones cada uno. Nos quedamos solo con las líneas de los costes:

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    | grep "cost ="
; cost = 188
; cost = 173
; cost = 168
; cost = 164
; cost = 159
```

Con `-v`, los mensajes de progreso que van a la salida de error indican cuándo se encontró cada plan, y la línea de estado da el veredicto:

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl -v 2>&1 \
    >/dev/null | grep -E "plan [0-9]|^kairos: solved"
[   0.003s] [search] plan 0 with 29 actions, cost 188
[   0.007s] [search] plan 1 with 28 actions, cost 173
[   0.014s] [search] plan 2 with 29 actions, cost 168
[   0.385s] [search] plan 3 with 29 actions, cost 164
[   0.944s] [search] plan 4 with 28 actions, cost 159
kairos: solved (proven optimal), cost 159
```

La Figura 9.2 dibuja estos números. El primer plan llega a los 3 milisegundos. Le siguen dos mejores antes de los 14 milisegundos: las mejoras baratas. Los dos siguientes tardan un tercio de segundo y un segundo entero. Después no se imprime nada durante otros tres segundos, hasta que la ejecución termina a los 3,9 segundos con las palabras `proven optimal`.

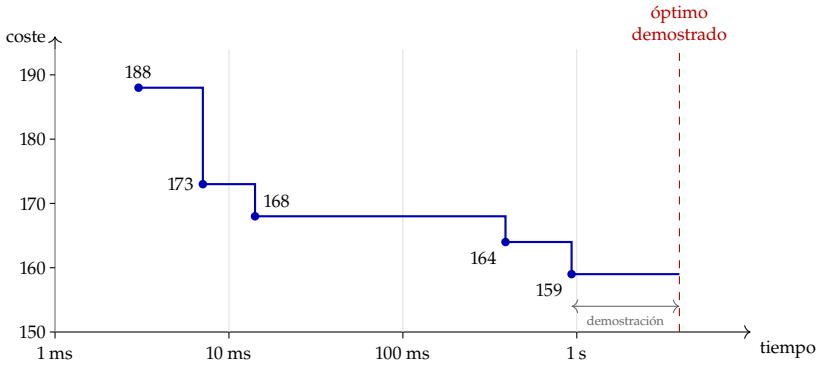


Figura 9.2.: Búsqueda anytime en la ronda de cuatro entregas: el coste del mejor plan conocido frente al tiempo, en un eje logarítmico. La mayor parte de la ejecución se dedica, después de encontrar el último plan, a demostrar que no existe otro mejor.

Qué significa `proven optimal`

La búsqueda de Kairos puede terminar de varias formas, y la línea de estado nombra la que ha ocurrido. `proven optimal` es la más fuerte: en una tarea secuencial, el último reinicio agotó todas las posibilidades de un plan más barato. No existe un plan de 158 metros para esta ronda. La demostración es sobre el modelo, y sobre la métrica tal como está escrita: no dice nada de planes que el dominio no permite, ni de otras ideas de «mejor».

Merece la pena leer el mejor plan con eso en mente. Está guardado en el fichero `courier-cost-p02.plan`:

```
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
(pick r1 sample1 ward-a)
(move r1 ward-a corridor)
(drop r1 sample1 corridor)
(move r1 corridor lobby)
(pick r1 meal1 lobby)
...
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds2 icu)
```

Todas las acciones son de `r1`. El segundo robot no sale nunca de la UCI. Con los metros como única medida, esto es correcto: `r2` está lejos de todo, y cada metro que recorre hacia el trabajo es un metro más en la suma. A un hospital no le haría ninguna gracia un robot pagado que se queda quieto mientras el otro hace todo el trabajo. La métrica pidió pocos metros y los obtuvo. Si lo que se quería era terminar pronto, el modelo necesita el tiempo, que es el tema de la Parte IV; si era repartir el desgaste, necesita otra medida. Un plan óptimo es la mejor respuesta a

la pregunta que se escribió, y por eso el paso M del método gana una pregunta en este capítulo.

La búsqueda anytime imprime cada plan mejor en cuanto lo encuentra. «Óptimo demostrado» significa que no existe en el modelo ningún plan más barato para la métrica tal como está escrita. La mayor parte del tiempo puede irse en esa demostración.

Lo bastante bueno

Casi tres de los cuatro segundos de la ejecución se fueron en demostrar, no en mejorar. En problemas más grandes puede que la demostración no llegue nunca, y la ejecución termina en el límite de tiempo (30 segundos, salvo que `--timeout` diga otra cosa) con el mejor plan encontrado hasta entonces. Tres opciones eligen cuánto esperar.

`--no-anytime` pide el primer plan y nada más. Es el planificador satisfactorio del Capítulo 3:

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    --no-anytime -v 2>&1 | grep -E "cost =|^kairos: solved"
; cost = 188
kairos: solved (no improvement), cost 188
```

`--target-cost C` dice qué es lo bastante bueno: la búsqueda se detiene en cuanto tiene un plan de coste `C` o menor.

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    --target-cost 170 -v 2>&1 | grep -E "cost =|^kairos: solved"
; cost = 188
; cost = 173
; cost = 168
kairos: solved (target reached), cost 168
```

El tercer plan, de 168 metros, fue el primero dentro del objetivo, y la ejecución terminó ahí, a los 14 milisegundos en lugar de a los 4 segundos. Un objetivo inalcanzable no hace ningún daño: con `--target-cost 100`, la misma ejecución sigue hasta `proven optimal` con coste 159.

`--timeout S` da un presupuesto en segundos. Con medio segundo, en nuestra máquina:

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    --timeout 0.5 -v 2>&1 | grep -E "cost =|^kairos: solved"
; cost = 188
; cost = 173
; cost = 168
; cost = 164
kairos: solved (timeout), cost 164
```

En los tres casos, el código de salida es 0 y el estado es `solved`: se encontró un plan. Las palabras entre paréntesis dicen hasta qué punto se puede confiar en que el plan sea el mejor. Solo `proven optimal` es una garantía. Las demás dicen por qué se detuvo la búsqueda.

Cuando los planes van a ficheros, con `--output round.plan`, cada plan mejor recibe su propio fichero, aquí de `round.plan.1` a `round.plan.5`, y el número más alto es el mejor. Ese es el fichero que hay que validar o ejecutar.

Óptimo, acotado, satisfactorio

Los planificadores suelen clasificarse por lo que prometen sobre la calidad. Un planificador *óptimo* solo devuelve planes de coste mínimo, y para ello necesita una heurística que nunca sobrestime y una búsqueda, como A*, que la respete; lo paga en tiempo y memoria. Un planificador *satisfactorio* promete un plan y nada sobre su coste. Entre ambos, los planificadores anytime dan una secuencia de planes cada vez mejores, y algunas búsquedas pueden prometer un plan dentro de un factor del óptimo. Las competiciones tienen categorías separadas para planificadores óptimos y satisfactorios, con reglas distintas: en una, un plan que no es óptimo es una respuesta errónea; en la otra, los planes se puntúan según lo cerca que esté su coste del mejor conocido.
Para un modelador, la diferencia se nota en lo que se puede concluir de una ejecución. Con un planificador óptimo, o con uno anytime que termina con una demostración, el plan dice algo del modelo: *esto* es lo que premia la métrica. Con una ejecución satisfactoria, un plan extraño puede ser cosa del planificador y no del modelo.

9.5. Métricas sobre cualquier fluente

Los costes de acción dan a un plan un número, construido de una sola manera. Las métricas de PDDL 2.1 son más generales: la expresión que sigue a `minimize` puede ser cualquier expresión aritmética sobre los fluentes numéricos del problema Fox y Long, 2003. Un dominio puede llevar varias medidas a la vez, cada una en su propio fluente, y cada problema puede elegir.

El mensajero tiene una segunda medida esperando desde el Capítulo 8: la energía. Los robots tienen consumos distintos, así que el plan con menos metros no tiene por qué ser el que menos energía gasta. El Listado 9.5 lleva dos contadores.

```
;; The courier world with two measures of a plan: the meters
;; covered and the energy used. Each problem says in its
;; metric which one counts, or how to mix them.
(define (domain courier-quality)
  (:requirements :strips :typing :numeric-fluents)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot))

  (:functions
    (distance ?from ?to - location)  ; meters, static
    (consumption ?r - robot)         ; energy per meter, static
    (meters)                         ; covered by all the robots
    (energy-used))                   ; used by all the robots

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
    :effect (and
      (not (robot-at ?r ?from)) (robot-at ?r ?to)
      (increase (meters) (distance ?from ?to))
      (increase (energy-used)
```

```
                (* (distance ?from ?to) (consumption ?r)))))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
    :effect (and (not (item-at ?i ?l)) (not (free ?r))
                 (carrying ?r ?i)))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l) (free ?r))))
```

Listado 9.5: El mensajero con dos medidas. `move` suma los metros a un fluente y la energía, que depende del robot, a otro.

Este dominio necesita `:numeric-fluents`. Incumple dos veces las reglas de los costes de acción: cambian dos fluentes, y la cantidad que se suma a `(energy-used)` es un producto. Ni `(meters)` ni `(energy-used)` aparecen en una precondición; son contadores puros, que están ahí para que los lea una métrica.

El problema de prueba tiene dos entregas, los medicamentos de la farmacia a la sala A y una muestra de la UCI al laboratorio, y dos robots: `r1`, ligero, en la base de carga, con un consumo de 1, y `r2`, pesado, en la sala B, con un consumo de 3. El mismo problema está escrito con varias métricas. Con `(:metric minimize (meters))`:

```
$ kairos courier-quality-domain.pddl courier-quality-meters.pddl
; cost = 68
(move r2 ward-b icu)
(pick r2 sample1 icu)
(move r2 icu ward-b)
(move r2 ward-b corridor)
(move r2 corridor lab)
(drop r2 sample1 lab)
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Cada robot se encarga de la entrega que tiene cerca: 30 metros para `r2` y 38 para `r1`. Con `(:metric minimize (energy-used))`, la búsqueda pasa por tres planes:

```
$ kairos courier-quality-domain.pddl courier-quality-energy.pddl \
    -v 2>&1 >/dev/null | grep -E "plan [0-9]|^kairos: solved"
[   0.002s] [search] plan 0 with 13 actions, cost 128
[   0.002s] [search] plan 1 with 15 actions, cost 92
[   0.003s] [search] plan 2 with 17 actions, cost 88
kairos: solved (proven optimal), cost 88
```

El primero es el plan de arriba: sus 30 metros de `r2` cuestan 90 unidades de energía, y con los 38 de `r1` el total es 128. En el segundo, `r1` hace las dos entregas, una detrás de otra: 92 metros, 92 unidades. Los planes se alargan a medida que se abaratan, de 13 acciones a 15 y a 17. Aquí la longitud y el coste tiran en direcciones opuestas. El tercer plan es este:

```
(move r1 dock lobby)
(move r1 lobby pharmacy)
```

```
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(drop r1 meds1 corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(pick r1 sample1 icu)
(move r1 icu ward-b)
(move r1 ward-b corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
(move r1 lab corridor)
(pick r1 meds1 corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Cuando diseñamos el problema, esperábamos que el óptimo fuera 92. El planificador ahorró 4 unidades con un truco que no se nos había ocurrido: deja los medicamentos en el suelo del pasillo de camino a la UCI, y vuelve a recogerlos cuando la muestra ya está en el laboratorio, de modo que entra en la sala A una sola vez. El plan es válido en el modelo. Si los medicamentos pueden esperar sin vigilancia en un pasillo es una pregunta para la ficha del modelo, y si la respuesta es que no, el modelo debe decirlo: con una precondición, o con las restricciones del Capítulo 15. Un planificador que optimiza explora los rincones de un modelo mucho más a fondo que uno satisfactorio, y encuentra lo que el modelo olvidó prohibir.

Una métrica puede ser cualquier expresión sobre fluentes numéricos. Lleva cada medida en su propio fluente contador y deja que cada problema elija. La optimización explora los resquicios de un modelo: lee el mejor plan con desconfianza.

Maximizar

La sección de la métrica acepta `maximize` en lugar de `minimize`. Se lee con naturalidad para deseos como «terminar con tanta batería como sea posible». Con el dominio de la batería del Capítulo 8 y una entrega:

```
$ kairos courier-battery-domain.pddl courier-battery-max.pddl
; cost = 62
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
note: the search minimizes the number of actions; the metric of the
     problem is reported for each plan but does not guide the search
      yet [orchestrator.metric]
```

El plan termina con 62 unidades, y la nota es sincera sobre lo que ha pasado: Kairos no pudo convertir esta métrica en costes de acciones, buscó un plan corto e informa del valor de la métrica para él. Existe un plan mejor. Si el robot vuelve a la base de carga después de la entrega y se carga, termina con la batería llena, y el validador está de acuerdo:

```
$ kairos validate courier-battery-domain.pddl \
    courier-battery-max.pddl courier-battery-max.plan
plan valid, makespan 11, metric 100
```

La razón está en cómo buscan los planificadores. Kairos minimiza costes que deriva de la métrica: lo que cada acción suma a los fluentes que

pondera la métrica. Un fluente que solo crece, en cantidades conocidas, bajo `minimize`, da un coste a cada acción, y se aplica la búsqueda de la Sección 9.4. La batería baja con los desplazamientos y sube de golpe con un `assign`; «lo que suma una acción» no tiene una respuesta fija. Otros planificadores trazan la línea en otro sitio, pero todos trazan una. El consejo es general: *escribe las métricas como sumas de cosas que solo se acumulan*. «Tanta batería como sea posible» se convierte en «tan poca energía gastada como sea posible», con un contador como en el Listado 9.5.

Una métrica `maximize` tiene un segundo peligro, que está en el modelo y no en el planificador. Si alguna acción puede incrementar el fluente sin límite, no hay mejor plan: todo plan se puede mejorar repitiendo esa acción una vez más. La Sección 9.9 vuelve sobre ello.

9.6. Varios deseos en un solo número

Los metros y la energía dieron dos planes distintos para las mismas dos entregas. Un hospital al que le importen ambos tiene que decir cuánto le importa cada uno, y la forma habitual es una **suma ponderada**: cada medida se multiplica por un peso que la convierte a una unidad común, y la métrica es la suma.

```
(:metric minimize (+ (* 3 (meters)) (energy-used))))
```

Esta es la métrica de `courier-quality-mix1.pddl`: un metro pesa tanto como tres unidades de energía. El mejor plan es el de la métrica de metros, con los dos robots, con un coste de $3 \times 68 + 128 = 332$. El fichero `courier-quality-mix2.pddl` hace pesar un metro tanto como una unidad, `(+ (meters) (energy-used))`:

```
$ kairos courier-quality-domain.pddl courier-quality-mix2.pddl \
    | grep cost
; cost = 196
; cost = 176
```

El plan con los dos robots cuesta ahora $68 + 128 = 196$, y el plan en el que `r1` lo hace todo, con su parada en el pasillo, cuesta $88 + 88 = 176$ y gana. La Figura 9.3 muestra por qué deciden los pesos. Cada plan es un punto en el plano de las dos medidas. Una suma ponderada traza sobre ese plano rectas paralelas de igual coste, y minimizar es deslizar la recta hacia el origen hasta que toque el último punto. La pendiente de la recta, que es la razón entre los pesos, elige el punto.

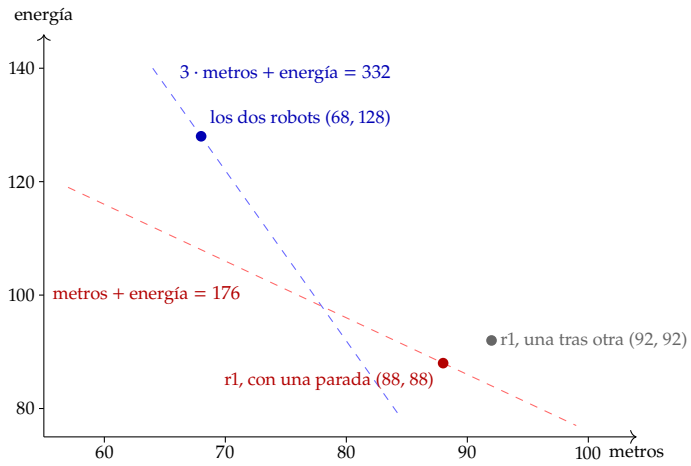


Figura 9.3.: Tres planes para las dos entregas como puntos (metros, energía). Las líneas discontinuas son rectas de igual coste para las dos métricas ponderadas; cada una toca primero un plan distinto. El plan gris es peor que el rojo en las dos medidas, y ninguna elección de pesos puede preferirlo.

Los pesos forman parte del modelo, y son la parte más difícil de obtener de quienes conocen el problema. «Minimizar la distancia y la energía» no es una especificación. «Un metro de recorrido desgasta al robot tanto como cuesta comprar tres unidades de energía» sí lo es. Cuando las medidas tienen una unidad común, el dinero sobre todo, los pesos son precios y la suma significa algo.

Un viaje por carretera: peajes y combustible

Un dominio en el que la unidad común es evidente: un coche va de una ciudad a otra. Entre algunas ciudades hay dos caminos, una carretera gratuita y una autopista más corta que cobra peaje. El viaje cuesta los peajes más el combustible.

```
;; A road trip: between two towns there is a free road and,
;; sometimes, a shorter highway with a toll. The trip costs
;; the tolls paid plus the fuel burned.
(define (domain road-trip)
  (:requirements :strips :typing :numeric-fluents)

  (:types car town)

  (:predicates
    (car-at ?c - car ?t - town)
    (road ?from ?to - town)
    (highway ?from ?to - town))

  (:functions
    (road-km ?from ?to - town)     ; length of the free road
    (highway-km ?from ?to - town)  ; length of the highway
    (toll ?from ?to - town)        ; euros to use the highway
    (km-per-liter ?c - car)
    (tolls-paid)                   ; euros
    (fuel-used))                   ; liters

  (:action drive-road
    :parameters (?c - car ?from ?to - town)
    :precondition (and (car-at ?c ?from) (road ?from ?to))
    :effect (and
      (not (car-at ?c ?from)) (car-at ?c ?to)
      (increase (fuel-used)
                (/ (road-km ?from ?to) (km-per-liter ?c)))))

  (:action drive-highway
    :parameters (?c - car ?from ?to - town)
    :precondition (and (car-at ?c ?from) (highway ?from ?to))
    :effect (and
```

```
34      (not (car-at ?c ?from)) (car-at ?c ?to)
35      (increase (tolls-paid) (toll ?from ?to))
36      (increase (fuel-used)
37                (/ (highway-km ?from ?to) (km-per-liter ?c))))))
```

El dominio lleva las dos cantidades en sus propias unidades, euros y litros. El precio del combustible no está en el dominio. Aparece en la métrica del problema como el peso que convierte los litros en euros:

Listado 9.6: Un viaje por carretera. Cada forma de conducir suma al combustible gastado; la autopista suma además su peaje.

```
; money: the tolls plus the fuel at 2 euros per liter
(:metric minimize (+ (tolls-paid) (* 2 (fuel-used)))))
```

La Figura 9.4 muestra la red de los problemas: cuatro ciudades, una carretera gratuita en cada uno de cuatro tramos y una autopista en tres de ellos.

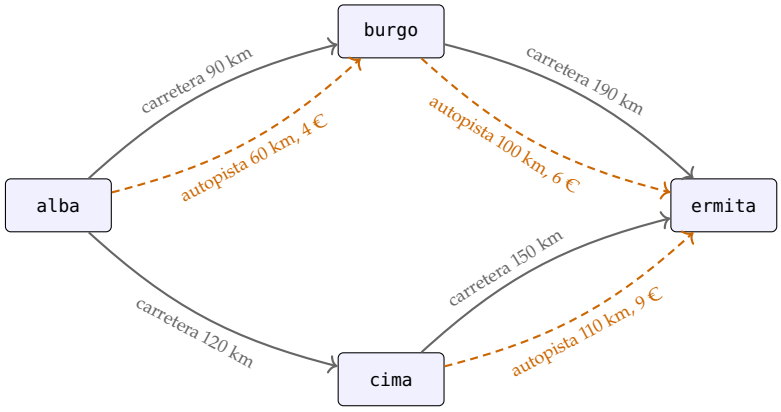


Figura 9.4.: La red de carreteras del viaje de Alba a Ermita. Las flechas continuas son carreteras gratuitas, y las discontinuas, autopistas con su peaje. La carretera de Burgo a Ermita sube un puerto; la autopista pasa por un túnel.

Un coche pequeño hace 20 km por litro, así que cada kilómetro le cuesta 10 céntimos:

```
$ kairos road-trip-domain.pddl road-trip-p01.pddl
; cost = 27
(drive-road car1 alba cima)
(drive-road car1 cima ermita)
; better plan 2
; cost = 25
(drive-road car1 alba burgo)
(drive-highway car1 burgo ermita)
```

El mejor viaje usa una autopista y no la otra. El túnel ahorra 90 km, 9 euros de combustible, a cambio de un peaje de 6. La primera autopista ahorra 30 km, 3 euros, a cambio de un peaje de 4: no compensa. El total son 190 km y 6 euros de peaje, 25 euros. En el segundo problema, el mismo viaje lo hace una furgoneta cargada que hace 8 km por litro, 25 céntimos por kilómetro:

```
$ kairos road-trip-domain.pddl road-trip-p02.pddl
; cost = 67.5
(drive-road van1 alba cima)
(drive-road van1 cima ermita)
; better plan 2
; cost = 50
(drive-highway van1 alba burgo)
(drive-highway van1 burgo ermita)
```

Para la furgoneta, todas las autopistas compensan: 160 km son 40 euros de combustible, más 10 de peajes. El mapa, las acciones y la métrica son los mismos que antes; ha cambiado un número estático, y con él el mejor plan. Y con `(:metric minimize (tolls-paid))`, en `road-trip-p03.pddl`, el coche toma las dos carreteras gratuitas por Cima con un coste de 0, y quema 13,5 litros que esta métrica no ve.

Una suma ponderada convierte varias medidas en un solo número. Los pesos son decisiones de modelado: con una unidad común, como el dinero, son precios, y cambiarlos cambia el mejor plan.

9.7. Un límite o una métrica

«Gastar lo menos posible en peajes» y «no gastar más de 5 euros en peajes» suenan parecido y son afirmaciones de naturaleza distinta. La primera es una métrica: ordena los planes y no excluye ninguno. La segunda es un **límite estricto**: divide los planes en permitidos y prohibidos y no dice nada de qué plan permitido es mejor. El Capítulo 8 nos dio la herramienta para un límite, una precondición sobre un número. En la acción de la autopista:

```
    :precondition (and
      (car-at ?c ?from) (highway ?from ?to)
      (<= (+ (tolls-paid) (toll ?from ?to)) (toll-budget)))
```

El contador `(tolls-paid)` tiene ahora dos lectores: la métrica, como antes, y esta precondición, que lo compara con una nueva función estática, el presupuesto. El fichero es `road-trip-budget-domain.pddl`; el resto del dominio no cambia. Con un presupuesto de 5 euros y la métrica de dinero de antes, el coche pequeño ya no puede usar el túnel:

```
$ kairos road-trip-budget-domain.pddl road-trip-budget-p01.pddl
; cost = 27
(drive-road car1 alba cima)
(drive-road car1 cima ermita)
```

El plan de 25 euros necesita un peaje de 6, por encima del presupuesto. Dentro del límite, lo mejor es la ruta gratuita, 27 euros. El límite ha encarecido el viaje: un límite solo puede quitar planes, así que el mejor plan bajo un límite nunca es mejor, según la métrica, que el mejor sin él. El validador muestra los dos lados de la línea. Un plan solo con la primera autopista está permitido y es pobre, y uno con las dos autopistas no es un plan de este modelo:

```
$ kairos validate road-trip-budget-domain.pddl \
    road-trip-budget-p01.pddl road-trip-budget-hr.plan
plan valid, makespan 2, metric 29
$ kairos validate road-trip-budget-domain.pddl \
    road-trip-budget-p01.pddl road-trip-budget-hh.plan
road-trip-budget-hh.plan:3: error: step 2 (drive-highway car1 burgo
     ermita) at 1.000: precondition false: (<= (+ (tolls-paid) (toll
      ?from ?to)) (toll-budget)): 10 <= 5 is false [validate.
     precondition_failed]
 3 | (drive-highway car1 burgo ermita)
plan invalid
```

Los límites y las métricas se combinan bien, como aquí: el límite dice qué es aceptable y la métrica elige entre lo aceptable. Que un deseo deba ser una cosa u otra depende de lo que pasa cuando no se cumple. Si la consecuencia es un fallo (la batería se agota, la tarjeta es rechazada, la bandeja se rompe), es un límite. Si la consecuencia es solo un resultado

peor, es una métrica. Un deseo expresado como límite sin más razón que la costumbre («como mucho 30 metros por entrega») puede dejar un problema sin ningún plan, donde una métrica habría devuelto el mejor plan que hay.

Un límite estricto es una precondición: prohíbe planes. Una métrica ordena los planes y no prohíbe ninguno. Usa un límite cuando cruzarlo es un fallo, y una métrica cuando solo es peor.

9.8. El método MODEL: qué hace mejor a un plan

Hasta este capítulo, la entrada Éxito de la ficha del modelo describía un conjunto de estados: los artículos están donde deben estar. Cualquier plan que los alcanzara era un éxito. Una métrica añade una segunda mitad a esa entrada, y pertenece al paso M porque hay que preguntársela a los dueños del problema antes de escribir una sola línea de PDDL.

Paso M: dos preguntas sobre la calidad

¿Qué hace que un plan sea mejor que otro? Nombra la medida y su unidad: metros, euros, unidades de energía, número de manipulaciones. Si la respuesta es «nada, cualquier plan sirve», dilo, y el modelo no necesita métrica.
¿Es un solo número? Si hay varias medidas, o bien tienen una unidad común y los pesos son precios, o bien alguien tiene que decidir cuánto de una vale una unidad de otra. Escribe los pesos en la ficha del modelo con su motivo. Y para cada deseo, pregunta si es una preferencia (una métrica) o una condición de éxito (un límite).

Para el mensajero de este capítulo, la ficha gana dos líneas. *Éxito:* cada artículo en su destino; entre los planes que lo consiguen, cuantos menos metros, mejor (o menos energía, o la suma ponderada, en los problemas posteriores). *Supuestos:* un metro cuesta lo mismo para todos los robots y a todas horas; coger y dejar son gratis; un robot que está quieto no cuesta nada. Cada supuesto es una forma en que el plan óptimo del modelo puede diferir de lo que el hospital llamaría el mejor, y el tercero ya nos lo encontramos en la Sección 9.4.

Los demás pasos heredan pequeñas obligaciones. En el paso O, cada medida de la métrica es un fluente contador en la tabla de estado, dinámico, que empieza en cero y que no lee ninguna precondición, salvo que sea también un límite. En el paso D, la tabla de acciones gana una columna: lo que cada acción suma a cada contador, con los ceros escritos y justificados. En el paso E, la batería de pruebas gana un problema con dos planes de coste conocido, para comprobar que el planificador prefiere el correcto, y la prueba de la trampa gana un nuevo blanco: una acción cuyo coste se olvidó.

Alternativas comparadas

Han aparecido tres maneras de decirle al planificador qué plan es mejor, y hemos ejecutado las dos entregas de la Sección 9.5 con cada una de ellas: sin ninguna métrica (`courier-quality-length.pddl`), costes de acción con los metros (`courier-cost-p03.pddl`, para el dominio del Listado 9.1) y una métrica sobre fluentes (`courier-quality-energy.pddl` y las dos mezclas).

	Longitud del plan	**Costes de acción**	**Métrica sobre fluentes**
Fidelidad	Correcta solo si todas las acciones valen lo mismo. Aquí: 13 acciones, 68 m, 128 de energía	Una suma de costes fijos por acción. Aquí: 68 m, óptimo	Cualquier expresión: energía que depende del robot, sumas ponderadas. Aquí: 88 de energía, o las mezclas
Tamaño	92 acciones instanciadas, ningún fluente	92 acciones instanciadas, un fluente que ninguna condición lee	92 acciones instanciadas, dos fluentes contadores
Guía	Búsqueda clásica	Espacio de estados clásico con pesos; la clase de optimización mejor soportado	Buena mientras los contadores solo crezcan en cantidades fijas; si no, puede que el planificador no optimice en absoluto
Claridad	Nada que escribir, y nada dicho	Una línea por acción y una en el problema	Contadores que mantener; la métrica documenta los pesos
Portabilidad	Todos los planificadores	`:action-costs`: incluidos los planificadores clásicos	`:numeric-fluents`: solo planificadores numéricos

Tabla 9.1.: Tres maneras de decir qué plan es mejor, en las dos entregas con dos robots. Las tres dan 92 acciones instanciadas con `kairos -v`; se diferencian en cuál es el mejor plan.

Veredicto. No digas nada cuando nada importa: rompecabezas, preguntas de viabilidad, las primeras versiones de un modelo que todavía se está depurando. Usa costes de acción siempre que la calidad sea una suma de costes fijos de las acciones (distancias, precios, riesgos, esfuerzo), que es lo más frecuente; el modelo sigue siendo clásico y casi cualquier planificador puede optimizarlo. Reserva las métricas sobre fluentes para lo que los costes de acción no pueden decir: un coste que depende del estado, como una energía que crece con la carga transportada; varias medidas que hay que ponderar de forma distinta en problemas distintos; o un contador que es también un límite.

En cuanto a la portabilidad, la fuente es la tabla de compatibilidad de la documentación de Kairos: entre los planificadores probados allí, aceptan costes de acción LAMA, POPF y OPTIC, además de Kairos, y métricas sobre fluentes numéricos POPF, OPTIC y ENHSP, pero no LAMA, que no tiene fluentes numéricos.

La segunda comparativa es la de la Sección 9.7, en el viaje del coche pequeño: los peajes como parte de una métrica (`road-trip-p01.pddl`), como límite estricto y nada más (`road-trip-budget-p02.pddl`, el presupuesto sin métrica) y como ambas cosas (`road-trip-budget-p01.pddl`).

	Solo métrica	Solo límite	Límite y métrica
Fidelidad	No se pierde ningún plan; un plan puede gastar en peajes más de lo que tiene el conductor. Mejor: 25 €, peaje 6	Nunca por encima del presupuesto; se acepta cualquier plan dentro de él, bueno o malo, y puede no haber ninguno	Nunca por encima del presupuesto, y el más barato dentro de él. Mejor: 27 €
Tamaño	Solo contadores	Una precondición numérica por cada acción limitada	Ambos
Guía	Los costes guían la búsqueda	El límite crea callejones sin salida: un peaje pagado pronto puede bloquear el único camino posterior	Ambos efectos
Claridad	El compromiso está en los pesos	La regla está donde se aplica, en la acción	Dos sitios que leer
Portabilidad	Métrica sobre fluentes	Precondiciones numéricas	Ambas

Tabla 9.2.: Los peajes del viaje por carretera como métrica, como límite estricto y como ambas cosas.

Veredicto. No son rivales: responden a preguntas distintas del paso M. Usa el límite para lo que no debe ocurrir y la métrica para lo que debe ser pequeño, y cuenta con que el límite cueste algo en la métrica. En caso de duda, empieza solo con la métrica y lee el mejor plan. Si hace algo inaceptable, faltaba un límite.

9.9. Errores comunes

Una acción que olvidó su coste

El error. Una acción de un dominio con costes no tiene `increase`. En esta cocina es `add-fat`:

```
    :effect (and (not (in-stock ?f)) (with-fat ?d)))
```

El síntoma. Ningún mensaje: una acción sin coste es legal, y gratuita. El planificador la usa todo lo que puede. La mejor comida cuesta ahora 6 en lugar de 10, y usa la mantequilla, la grasa más cara de la despensa, porque para el planificador ninguna grasa cuesta nada:

```
$ kairos kitchen-free-domain.pddl kitchen-free-p01.pddl
...
; better plan 2
; cost = 6
(add-egg egg1 cake1)
(add-egg egg2 om1)
(add-sweetener sugar1 cake1)
(add-fat butter1 cake1)
(bake cake1)
(add-fat oil1 om1)
(fry om1)
```

La corrección. Rellena la columna de coste de la tabla de acciones para todas las acciones, ceros incluidos. Calcula a mano el coste de un plan y compáralo con el coste que imprime el planificador.

La métrica en el fichero equivocado

El error. Escribir `(:metric ...)` en el dominio, junto a las acciones que incrementan el coste.

El síntoma. Un error de sintaxis:

```
$ kairos check kitchen-wrong-domain.pddl
kitchen-wrong-domain.pddl:22:4: error: unknown domain section :
     metric [parse.unknown_section]
 22 |   (:metric minimize (total-cost))
    |    ^
invalid
```

La corrección. La métrica va en el problema, después del objetivo. La equivocación contraria no da ningún error: un dominio lleno de costes con un problema que no tiene métrica se planifica por longitud del plan, y la única señal es un coste en la salida que parece un número de acciones.

Costes negativos y mejora sin fin

El error. Un coste que puede ser negativo, como un reembolso o una recompensa, o una métrica `maximize` sobre algo que una acción puede incrementar indefinidamente.
El síntoma. Con costes de acción, los valores negativos incumplen las reglas de la característica, y cada planificador reacciona a su manera. Kairos deja de optimizar y lo dice. Con una despensa en la que el azúcar tiene un precio de -1 (`kitchen-negative.pddl`), devuelve el primer plan, de coste 18, con la nota que vimos para `maximize`:

```
note: the search minimizes the number of actions; the metric of
     the problem is reported for each plan but does not guide
     the search yet [orchestrator.metric]
```

Detrás de la regla hay un problema real. Si un ciclo de acciones tiene un coste total negativo, o un fluente que se maximiza puede crecer sin límite, todo plan se puede mejorar dando una vuelta más, y «el mejor plan» no existe.
La corrección. Mantén los costes no negativos: modela una recompensa como un coste menor, o como un coste por *no* hacerlo (las preferencias del Capítulo 14 hacen exactamente eso). Acota lo que se maximiza, como el Capítulo 8 acotó cada cantidad.

Fiarse del primer plan, o de la palabra equivocada

El error. Tomar el primer plan de una ejecución anytime como la respuesta, o leer `solved` como «óptimo».
El síntoma. Conclusiones sobre el modelo que en realidad son sobre la búsqueda: «el modelo envía al robot lejano», cuando un plan posterior no lo hace. En la ronda de la Sección 9.4, el primer plan cuesta 188 y el mejor, 159.
La corrección. Lee el último plan, y la línea de estado con `-v`. Solo `proven optimal` dice que no existe nada mejor; `timeout`, `target reached` y `no improvement` dicen por qué se detuvo la búsqueda. Y recuerda qué es la salida anytime: varios planes, uno detrás de otro. Guardada en un solo fichero y pasada al validador como un único plan, no es válida. Usa `--output` y valida el último fichero.

El valor inicial del coste

El error. Omitir `(= (total-cost) 0)` en `:init`.
El síntoma. Según la regla del Capítulo 8, un fluente sin valor inicial está indefinido, y no se puede aplicar un `increase` sobre un fluente

indefinido. Kairos es indulgente en esto y planifica la cocina sin esa línea como si el coste empezara en cero. Un planificador que siga la regla no encontraría aplicable ninguna acción que tenga un coste. **La corrección.** Inicializa siempre cada contador que lea la métrica. Es una línea, y así el modelo significa lo mismo en todas partes.

9.10. Resumen

Una métrica asigna un número a cada plan y pide el menor, o el mayor. Sin ella, el coste de un plan secuencial es su longitud. Los costes de acción son el caso sencillo y portable: un único fluente `(total-cost)` que las acciones incrementan en constantes o en datos estáticos, sobre un espacio de estados que sigue siendo clásico. Las métricas sobre fluentes numéricos son el caso general: varios contadores, sumas ponderadas, una elección por problema. Optimizar es más difícil que resolver; un planificador anytime imprime planes cada vez mejores, y su última palabra dice si está demostrado que el último plan es el mejor o si solo es el mejor encontrado. Y una métrica es una decisión de modelado como cualquier otra: el plan óptimo responde a la pregunta tal como se escribió.

Tabla 9.3.: Las construcciones de PDDL y las opciones de Kairos usadas en este capítulo.

Construcción u opción	Significado
`:action-costs`	requisito: solo cambia `(total-cost)`, en constantes no negativas o funciones estáticas
`(increase (total-cost) e)`	el coste de una acción, en su efecto
`(= (total-cost) 0)`	en `:init`: todo contador empieza en cero
`(:metric minimize e)`	en el problema: preferir planes con un valor final menor de `e`
`(:metric maximize e)`	lo mismo para valores mayores; reescríbela como minimización cuando sea posible
`(+ (* w1 f1) (* w2 f2))`	una suma ponderada de contadores
`kairos D P`	anytime: se imprime cada plan mejor; el último es el mejor
`--no-anytime`	solo el primer plan
`--target-cost C`	parar en el primer plan de coste `C` o menor
`--timeout S`	parar al cabo de `S` segundos con el mejor plan encontrado
`-v`	línea de estado: `solved (proven optimal)` y los demás motivos de parada

El mensajero ya sabe cuánto recorre y lo que eso cuesta. Todavía cree que cada acción ocurre en un instante y que los robots actúan uno detrás de otro, y por eso un plan que deja parado a uno de los dos robots podía ser «óptimo». La Parte IV da duración a las acciones, permite que se solapen y añade la medida que este capítulo no podía escribir: el instante en que termina el plan.

9.11. Lecturas complementarias

Las métricas de planes se definen, junto con los fluentes numéricos, en el artículo sobre PDDL 2.1 Fox y Long, 2003. La sintaxis exacta de `:action-costs` y sus restricciones están en la gramática de PDDL 3.1 Kovacs, 2011. El informe sobre la competición de 2006 discute con detalle cómo comparar planificadores por la calidad de sus planes A. E. Gerevini et al., 2009.

El artículo sobre LAMA Richter y Westphal, 2010 es la mejor descripción de la búsqueda anytime con costes de acción: los reinicios con voracidad decreciente, y por qué importa tratar con cuidado las acciones de coste cero. Para los algoritmos de búsqueda en sí (A*, A* ponderado y lo que garantiza una heurística admisible) sirve cualquier libro de texto general Russell y Norvig, 2020, y la visión desde la planificación está en Geffner y Bonet, 2013. El modelado de costes y métricas, con más ejemplos, se trata en Haslum et al., 2019.

9.12. Ejercicios

Ejercicios de la Sección 9.1 — Cuando no basta con un plan cualquiera

Ejercicio 9.1.1. Para cada situación, di qué hace mejor a un plan, en qué unidad, y si la longitud del plan sería una medida justa de ello: (a) las torres de Hanói; (b) un mensajero al que se le paga por entrega y que paga su propio combustible; (c) un brazo robótico que apila bloques, donde coger o colocar un bloque lleva más o menos el mismo tiempo; (d) una cocina que tiene que servir un menú y quiere desperdiciar la menor cantidad posible de comida.

Ejercicio 9.1.2. Ejecuta las dos versiones del primer problema del capítulo, los ficheros `courier-cost-p01-nometric.pddl` y `courier-cost-p01.pddl`. Las dos salidas empiezan con una línea `; cost =`. ¿Qué significa el número en cada una? ¿Por qué la primera ejecución imprime un plan y la segunda dos?

Ejercicios de la Sección 9.2 — Costes de acción

Ejercicio 9.2.1. En el hospital, cada vez que el robot coge o deja algo tiene que acudir una enfermera. Da a cada una de esas acciones un coste de 3 en el dominio del Listado 9.1 y vuelve a ejecutar la ronda de cuatro entregas. ¿Cuál es ahora el coste del mejor plan, y cuántas veces coge y deja algo el robot en él? Compáralo con el mejor plan de la Sección 9.4, que tiene diez.

Ejercicio 9.2.2. Un estudiante quiere que los desplazamientos con carga sean más caros. En un dominio cuyo único requisito numérico es `:action-costs`, hace que `move` incremente el coste total en `(load ?r)`, un fluente que cambian las acciones de coger y dejar. ¿Qué regla de los costes de acción incumple? ¿Cómo escribirías tú el modelo?

Ejercicio 9.2.3. Calcula a mano el coste del mejor plan de la ronda de cuatro entregas, guardado en `courier-cost-p02.plan` (las distancias están en el fichero del problema y en el Capítulo 8). Comprueba tu suma con `kairos validate`.

Ejercicios de la Sección 9.3 — Costes a partir de datos: una cocina con precios

Ejercicio 9.3.1. La mantequilla de la despensa caduca hoy, así que usarla no cuesta nada: si no, se tiraría. Cambia el problema, no el dominio, y predice la mejor comida y su coste antes de ejecutarlo.

Ejercicio 9.3.2. El dominio de la cocina permite que `add-egg` ponga un segundo huevo en un plato que ya tiene uno. ¿Lo hace alguna vez

el planificador cuando hay una métrica? ¿Y sin métrica? ¿Debería prohibirlo el dominio?

Ejercicios de la Sección 9.4 — El planificador en acción: búsqueda anytime

Ejercicio 9.4.1. Ejecuta la ronda de cuatro entregas con `--no-anytime`, con `--target-cost 165` y sin ninguna opción, añadiendo `-v` cada vez. Rellena una tabla con el coste del plan devuelto, el motivo de parada y el tiempo total. ¿Qué ejecución usarías en un robot que tiene que empezar a moverse en menos de una décima de segundo?

Ejercicio 9.4.2. El mejor plan de la ronda deja parado a `r2`. Propón dos cambios distintos en el modelo, ya sea en el dominio o en la métrica, que harían que un plan óptimo usara los dos robots, y di qué supone cada uno sobre el hospital.

Ejercicio 9.4.3. ¿Qué responde Kairos a `--target-cost 100` en la ronda de cuatro entregas, y por qué es la única respuesta sensata?

Ejercicios de la Sección 9.5 — Métricas sobre cualquier fluente

Ejercicio 9.5.1. Añade un tercer contador al dominio del Listado 9.5: el número de manipulaciones: las veces que el robot coge algo más las que lo deja. Con la métrica «energía, más 5 unidades por cada manipulación», ¿cuál es el mejor plan para las dos entregas? Explica por qué desaparece la parada en el pasillo.

Ejercicio 9.5.2. Un problema para el dominio de la carga del Capítulo 8 tiene la métrica `(:metric maximize (load r1))`. Sin ejecutar nada: ¿qué clase de plan premia esta métrica? ¿Hay un mejor plan? ¿Qué quería probablemente el autor, y cómo debería escribirse?

Ejercicios de la Sección 9.6 — Varios deseos en un solo número

Ejercicio 9.6.1. Para las dos entregas, con la métrica `(+ (meters) (* w (energy-used)))`, ¿con qué peso `w` cambia el mejor plan de «los dos robots» a «`r1` solo»? Calcúlalo a partir de los puntos de la Figura 9.3 y compruébalo después con un peso justo por debajo y otro justo por encima.

Ejercicio 9.6.2. A 2 euros el litro, el coche pequeño solo usa el túnel. ¿A qué precio del combustible empieza a tomar las dos autopistas? Calcúlalo y compruébalo después con un problema a 3 euros.

Ejercicio 9.6.3. Los pesos de una suma ponderada pueden ocultar por completo una medida. En `(+ (* 1000 (meters)) (energy-used))`, ¿qué papel tiene la energía? ¿Cuándo es útil una métrica así?

Ejercicios de la Sección 9.7 — Un límite o una métrica

Ejercicio 9.7.1. Convierte el combustible del viaje por carretera en un límite estricto: el coche sale con un número de litros y no hay gasolineras por el camino. Mantén los peajes como métrica. Con 9 litros, ¿por dónde va el coche pequeño? ¿Y con 7?

Ejercicio 9.7.2. Para cada deseo del mundo del mensajero, di si lo modelarías como un límite o como una métrica, y por qué: (a) la batería no debe agotarse; (b) el robot no debería llevar más de 5 kg; (c) las

muestras deberían pasar poco tiempo fuera del laboratorio; (d) el robot debería evitar la UCI cuando pueda.

Ejercicios de la Sección 9.8 — El método MODEL: qué hace mejor a un plan

Ejercicio 9.8.1. Quita la métrica del problema de la cocina y ejecútalo. Aplica el paso L al par «longitud del plan» y «costes de acción» para este dominio: ¿en cuáles de los cinco criterios se diferencian realmente?

Ejercicio 9.8.2. *De una historia a un modelo.* Un dron lleva un paquete desde un almacén hasta una clínica. Puede sobrevolar la plaza mayor, que es un camino corto y concurrido, o dar un rodeo por el río, que es largo y vacío. Cada vuelo gasta energía y tiene un nivel de riesgo de 0 a 5. El operador dice: «Quiero ahorrar batería, pero no a cualquier riesgo». Aplica el paso M con sus dos preguntas nuevas, escribe el dominio y un problema, y encuentra un peso para el riesgo con el que el dron dé el rodeo y otro con el que cruce la plaza.

Parte IV. Tiempo: PDDL 2.1 y 2.2

10. Acciones durativas

Los planes de los capítulos anteriores son listas: primero esta acción, después aquella. Los números les dieron un coste en el Capítulo 9 y, con él, una manera de preferir una lista a otra, pero una lista sigue siendo una lista. Demos al hospital un segundo robot y pidamos dos entregas: medicamentos a la UCI y una muestra al laboratorio. Un planificador responde con trece acciones, una por línea, unas de `r1` y otras de `r2`. Nada en esa respuesta dice que los dos robots pueden trabajar a la vez, ni que recorrer los doce metros hasta la sala A lleva más tiempo que coger una caja, ni cuándo estarán los medicamentos en la UCI, que es lo único que quiere saber la enfermera que los espera.

Este capítulo da a las acciones una **duración** y les permite solaparse. PDDL 2.1 lo hace con un tipo nuevo de acción, la *acción durativa*, que tiene un inicio, un final y un intervalo entre ambos, y que dice en cuál de los tres debe cumplirse cada condición y ocurre cada efecto. Un plan ya no es una lista, sino un horario: cada acción tiene un instante de inicio y una duración, y la calidad que miramos es el instante en que todo está hecho.

Empezamos por lo que una secuencia no puede decir y escribimos una primera acción durativa para el mensajero. Leer su plan plantea dos preguntas, que responden las dos secciones siguientes: qué ocurre exactamente en cada instante y cómo decir que algo debe seguir siendo cierto mientras una acción se ejecuta. Después calculamos las duraciones a partir de distancias y velocidades, ponemos a dos robots a trabajar en paralelo y obligamos a un robot a hacer una sola cosa cada vez. Una cocina y un ascensor muestran las mismas ideas lejos del hospital. El capítulo termina, como siempre, con el método, los errores, un resumen y ejercicios.

10.1 Lo que una secuencia no puede decir 222
10.2 Una primera acción durativa 224
El mensajero con duraciones fijas 225
Leer un plan temporal . 226
10.3 Qué ocurre en cada instante 227
Por qué 10,001 y no 10 . 227
El validador está de acuerdo 228
10.4 Condiciones en el tiempo 228
El intervalo es abierto . 229
10.5 Duraciones que dependen de la acción 230
10.6 Trabajar en paralelo . . 232
10.7 Una cosa cada vez . . . 234
10.8 Otros dominios: una cocina y un ascensor . . 237
Una cocina: cosas que funcionan solas 237
Un ascensor: esperar a las personas 238
10.9 El método MODEL: tiempo 240
Alternativas comparadas 241
10.10 Errores comunes 243
10.11 Resumen 244
10.12 Lecturas complementarias 245
10.13 Ejercicios 245

10.1. Lo que una secuencia no puede decir

La Figura 10.1 muestra el problema del primer párrafo. El robot `r1` está en la base de carga y `r2` en el pasillo; `meds1` espera en la farmacia y debe ir a la UCI, y `sample1` espera en la sala A y debe ir al laboratorio. Con el modelo del Capítulo 4, un planificador devuelve una secuencia de trece acciones: ocho de `r1`, que lleva los medicamentos, y cinco de `r2`, que lleva la muestra.

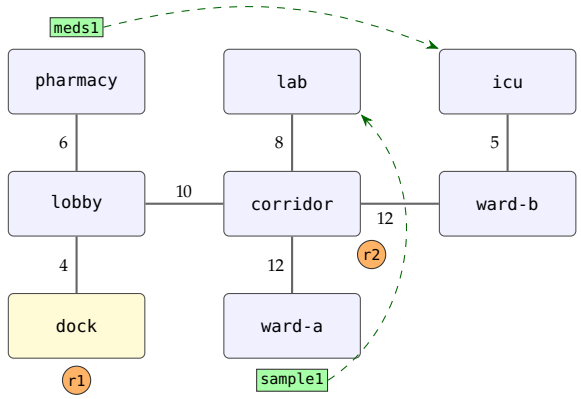


Figura 10.1.: El problema de este capítulo: dos robots (círculos), dos artículos (cajas) y el destino de cada artículo (flechas discontinuas). Los números son las distancias en metros entre habitaciones.

Hay tres cosas que faltan en esa secuencia.

- **Cuánto dura cada acción.** Ir del pasillo a la sala B son doce metros; levantar una caja lleva un par de segundos. En la secuencia, ambas cosas ocupan una línea.
- **Qué puede ocurrir a la vez.** Los dos robots no se necesitan el uno al otro. La secuencia intercala sus acciones en algún orden, y cualquier otro intercalado valdría lo mismo: el orden entre una acción de `r1` y otra de `r2` no significa nada.
- **Cuándo se alcanza el objetivo.** Si las acciones se ejecutan una tras otra, el trabajo termina cuando se han sumado las duraciones de las trece. Si cada robot ejecuta sus propias acciones sin esperar al otro, termina mucho antes.

La Figura 10.2 dibuja las dos lecturas de las mismas trece acciones, con las duraciones que les dará este capítulo. Ejecutadas en secuencia, tardan 67 segundos; en paralelo, 47. Este tipo de dibujo, con el tiempo avanzando hacia la derecha y una fila por cada cosa que trabaja, es un **cronograma** (o diagrama de Gantt), y es como miraremos todos los planes a partir de ahora.

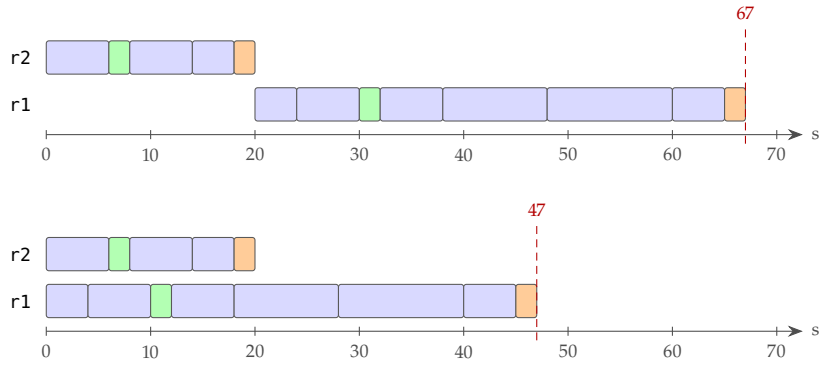


Figura 10.2.: Las mismas trece acciones ejecutadas una tras otra (arriba) y con cada robot trabajando por su cuenta (abajo). Las barras azules son desplazamientos; las verdes, recogidas; las naranjas, descargas. La línea discontinua marca el instante en que los dos artículos están entregados.

Podríamos intentar quedarnos con las herramientas que ya tenemos. Hagamos que el coste de una acción sea su duración, como en el Capítulo 9, y minimicemos el coste total: el planificador preferirá entonces las rutas cortas. Pero el coste total es la *suma* de las duraciones, los 67 del dibujo de arriba, hagan lo que hagan los robots en paralelo. Un coste no distingue los dos dibujos de la Figura 10.2. En la Sección 10.9 mediremos exactamente hasta dónde llega ese enfoque; para distinguirlos, el tiempo tiene que entrar en el modelo.

Una secuencia no dice cuánto duran las acciones, cuáles pueden solaparse ni cuándo se alcanza el objetivo. Un coste suma duraciones; no ve el trabajo en paralelo.

El tiempo entra en PDDL

Las acciones durativas llegaron con PDDL 2.1, el lenguaje que Maria Fox y Derek Long definieron para la tercera International Planning Competition, celebrada en 2002 (Fox & Long, 2003). PDDL 2.1 se organizaba en niveles: fluentes numéricos (el tema del Capítulo 8), acciones durativas con efectos discretos (este capítulo y el siguiente) y acciones durativas con efectos continuos (Capítulo 12). El artículo es también donde se define con precisión la semántica que este capítulo explica con ejemplos (qué ocurre en un instante, qué acciones pueden compartirlo).

10.2. Una primera acción durativa

Tomemos `move` del mensajero con tipos del Capítulo 4. Como acción instantánea dice: si el robot está en `?from` y las habitaciones están conectadas, el robot deja de estar en `?from` y pasa a estar en `?to`. Un desplazamiento que dura diez segundos tiene más que decir, porque ahora hay dos instantes, aquel en que el robot sale y aquel en que llega, y no todos los efectos ocurren en el mismo. Cuando el robot sale, deja de estar en `?from`. Diez segundos después está en `?to`. Entre medias no está en ninguna de las dos.

```
(:durative-action move
  :parameters (?r - robot ?from ?to - location)
  :duration (= ?duration 10)
  :condition (and (at start (robot-at ?r ?from))
                  (at start (connected ?from ?to)))
  :effect (and (at start (not (robot-at ?r ?from)))
               (at end (robot-at ?r ?to))))
```

Una **acción durativa** se escribe con `:durative-action` y tiene cuatro partes.

- `:parameters`, como en cualquier acción.
- `:duration`, una restricción sobre la variable especial `?duration`. Aquí la fija: la acción dura 10. PDDL no tiene unidad de tiempo; en el mundo del mensajero leemos todas las duraciones como segundos.
- `:condition`, que ocupa el lugar de `:precondition`. El nombre nuevo está justificado: estas condiciones no se comprueban todas *antes* de la acción. Cada una lleva un especificador temporal: `(at start ...)` para el instante en que la acción empieza, `(at end ...)` para el instante en que termina y `(over all ...)` para el intervalo entre ambos.
- `:effect`, donde cada efecto es `(at start ...)` o `(at end ...)`. No hay efectos `over all`: un cambio discreto ocurre en un instante.[1]

Las acciones durativas necesitan `:durative-actions` en los `:requirements`. Un dominio puede mezclarlas con acciones instantáneas ordinarias; el nuestro no lo necesitará.

1: El cambio que se produce *durante* la acción, como una batería que se descarga segundo a segundo, es un efecto continuo y pertenece al Capítulo 12.

Una acción durativa tiene una duración, condiciones al inicio, durante todo el intervalo y al final, y efectos al inicio y al final.

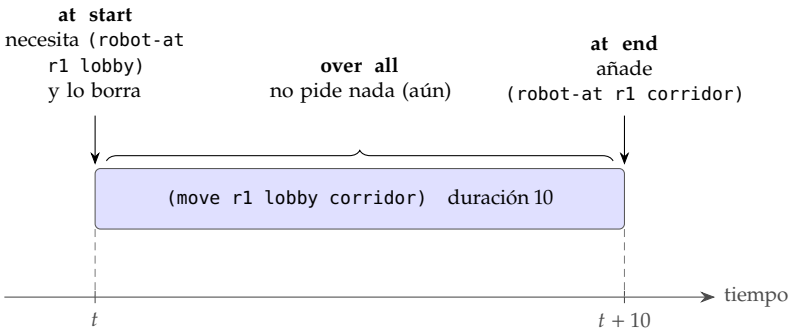


Figura 10.3.: Los tres momentos de una acción durativa. Las condiciones pueden asociarse a cualquiera de ellos; los efectos, solo a los dos extremos. Durante el intervalo, el robot de este `move` no está en ninguna habitación.

El mensajero con duraciones fijas

El Listado 10.1 es el mensajero con tipos completo, reescrito con acciones durativas. Cada desplazamiento dura 10 segundos y cada recogida y cada descarga, 2. Los predicados son los del Capítulo 4.

```
;; The courier world with durative actions of fixed duration:
;; every move lasts 10 seconds, every pick and drop 2 seconds.
(define (domain courier-fixed)
  (:requirements :strips :typing :durative-actions)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot))

  ;; The robot leaves ?from when the move starts
  ;; and is in ?to when it ends.
  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration 10)
    :condition (and (at start (robot-at ?r ?from))
                    (at start (connected ?from ?to)))
    :effect (and (at start (not (robot-at ?r ?from)))
                 (at end (robot-at ?r ?to))))

  (:durative-action pick
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration 2)
    :condition (and (at start (robot-at ?r ?l))
                    (at start (item-at ?i ?l))
                    (at start (free ?r)))
    :effect (and (at start (not (item-at ?i ?l)))
                 (at start (not (free ?r)))
                 (at end (carrying ?r ?i))))

  (:durative-action drop
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration 2)
    :condition (and (at start (robot-at ?r ?l))
                    (at start (carrying ?r ?i)))
    :effect (and (at start (not (carrying ?r ?i)))
                 (at end (item-at ?i ?l))
                 (at end (free ?r)))))
```

Listado 10.1: El mensajero con acciones durativas de duración fija (`courier-fixed-domain.pddl`).

Para `pick` y `drop` hemos seguido el mismo razonamiento que para `move`: lo que la acción quita va al inicio y lo que da va al final. Cuando empieza la recogida, el artículo deja de estar en el estante y la pinza deja de estar libre; cuando termina, dos segundos después, el robot lleva el artículo.

Cuando empieza la descarga, el robot deja de llevarlo; cuando termina, el artículo está en la habitación y la pinza está libre. La Sección 10.9 muestra qué falla con cualquier otra colocación.

El problema es la entrega del Capítulo 4, sin más cambio que el nombre de su dominio: en un problema no hay que decir nada sobre el tiempo. Kairos comprueba el par e indica qué características ha encontrado:

```
$ kairos check courier-fixed-domain.pddl courier-fixed-p01.pddl
valid, plannable
  typing: full
  durative-actions: full (required concurrency included, in engine
     M3)
```

Y planifica:

```
$ kairos courier-fixed-domain.pddl courier-fixed-p01.pddl
; cost = 52.006
; makespan = 52.006
0.000: (move r1 dock lobby) [10.000]
10.001: (move r1 lobby pharmacy) [10.000]
20.002: (pick r1 meds1 pharmacy) [2.000]
20.003: (move r1 pharmacy lobby) [10.000]
30.004: (move r1 lobby corridor) [10.000]
40.005: (move r1 corridor ward-a) [10.000]
50.006: (drop r1 meds1 ward-a) [2.000]
```

Leer un plan temporal

Un **plan temporal** es un conjunto de líneas de la forma

$$t\texttt{: (}action\texttt{) [}d\texttt{]}$$

donde t es el instante en que empieza la acción y d su duración; la acción termina en $t + d$. Las líneas están ordenadas por instante de inicio, pero el orden de las líneas no significa nada más: dos acciones se solapan siempre que se solapen sus intervalos. El reloj empieza en 0 con el estado inicial.

El ***makespan*** de un plan es el instante en que termina su última acción: aquí, 50,006 + 2 = 52,006 segundos. Responde a la pregunta de la enfermera. Si el problema no tiene `:metric`, Kairos toma el makespan como coste de un plan temporal, y por eso las dos líneas de comentario muestran el mismo número.

Leamos ahora el plan como un estudiante debería leer todos los planes: línea a línea, preguntándonos si el mundo lo permitiría. Llaman la atención dos cosas.

1. El segundo desplazamiento empieza en 10,001, no en 10. A lo largo de todo el plan se cuela una milésima de segundo entre una acción y la siguiente, y al final hay seis.
2. La recogida empieza en 20,002 y dura dos segundos, pero el robot sale de la farmacia en 20,003. Agarra la caja mientras se va.

La primera es correcta y tiene una razón precisa, que es el tema de la Sección 10.3. La segunda es un defecto de nuestro modelo: el planificador ha encontrado un plan que el mundo no permite, y la culpa es nuestra. La Sección 10.4 lo repara.

Un plan temporal da a cada acción un instante de inicio y una duración. Su makespan es el final de su última acción. Léelo línea a línea: el planificador aprovecha toda libertad que le deje el modelo.

10.3. Qué ocurre en cada instante

Una acción durativa toca el mundo dos veces: una cuando empieza y otra cuando termina. Entre medias no cambia nada. Así que, por largas que sean las acciones, un plan temporal cambia el mundo solo en un número finito de instantes: los inicios y los finales de sus acciones. Cada uno de esos instantes, con todo lo que empieza o termina en él, se llama **acontecimiento**.

Para seguir un plan, se ordenan sus acontecimientos por tiempo y se aplican uno a uno, como si cada inicio y cada final fueran una pequeña acción instantánea. En un inicio se comprueban las condiciones `at start` y se aplican los efectos `at start`. En un final se comprueban las condiciones `at end` y se aplican los efectos `at end`. Entre dos acontecimientos consecutivos no cambia nada, así que ahí basta con comprobar que se cumple cada condición `over all` de las acciones en curso. La Tabla 10.1 lo hace con los primeros segundos del plan anterior.

Tiempo	Acontecimiento	Estado posterior (hechos dinámicos de `r1`)
0	estado inicial	`(robot-at r1 dock)`, `(free r1)`
0,000	inicio de `move dock lobby`	`(free r1)`
10,000	final de `move dock lobby`	`(robot-at r1 lobby)`, `(free r1)`
10,001	inicio de `move lobby pharmacy`	`(free r1)`
20,001	final de `move lobby pharmacy`	`(robot-at r1 pharmacy)`, `(free r1)`
20,002	inicio de `pick`	`(robot-at r1 pharmacy)`
20,003	inicio de `move pharmacy lobby`	(ninguno)
22,002	final de `pick`	`(carrying r1 meds1)`

Tabla 10.1.: Los primeros acontecimientos del plan de `courier-fixed-p01`. Entre 0 y 10, el robot no está en ninguna habitación; en 20,003 sale de la farmacia con la recogida todavía en curso.

Por qué 10,001 y no 10

El primer desplazamiento termina en 10,000 y añade el hecho `(robot-at r1 lobby)`. El segundo necesita justo ese hecho al empezar. Supongamos que ambos ocurren exactamente en 10. Entonces, en ese instante, una cosa está haciendo cierto el hecho mientras otra depende de él. ¿Cuál va primero? Dentro de un mismo instante no hay «primero». Un ejecutor que envíe las dos órdenes a un robot real «a la vez» puede entregarlas en cualquier orden.

PDDL 2.1 lo resuelve con una regla que Fox y Long llamaron *no moving targets*: nada puede depender de un hecho en el instante mismo en que otra cosa lo cambia. Dos acontecimientos **interfieren** cuando uno añade o borra un hecho (o cambia un número) que el otro necesita, añade o borra. Los acontecimientos que interfieren deben estar separados por una cantidad positiva de tiempo, que se escribe ϵ (épsilon): lo bastante larga para que el cambio haya terminado y se haya observado antes de que nadie dependa de él.

Cuánto vale ϵ depende de quién ejecute el plan; el lenguaje solo pide que no sea cero. Kairos usa 0,001 salvo que se le indique otra cosa, y esa es la milésima que separa cada acción de nuestro plan de la siguiente. Si pedimos medio segundo, el plan se estira:

```
$ kairos courier-walk-domain.pddl courier-walk-p01.pddl --epsilon
    0.5
; cost = 31
; makespan = 31.000
0.000: (move r1 dock lobby) [10.000]
10.500: (move r1 lobby corridor) [10.000]
```

```
21.000: (move r1 corridor ward-a) [10.000]
```

(`courier-walk` es un mensajero que solo se desplaza; volveremos a encontrarlo.) Los acontecimientos que no interfieren pueden compartir instante. Los dos robots de la Figura 10.2 empiezan ambos en 0,000: ninguno toca nada que el otro necesite.

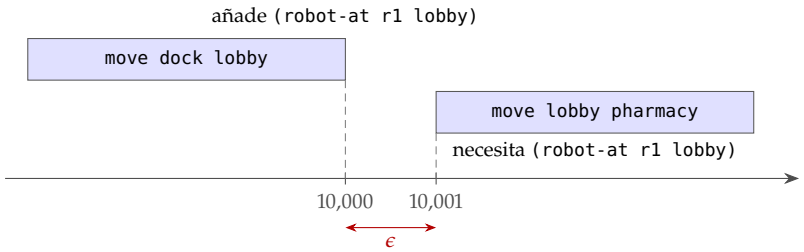


Figura 10.4.: El final de un desplazamiento y el inicio del siguiente, ampliados. El final cambia el hecho que necesita el inicio, así que los dos acontecimientos interfieren y deben estar separados por ϵ.

El validador está de acuerdo

El plan siguiente es la misma entrega para el dominio de la Sección 10.5, escrito a mano, con cada acción empezando en el instante exacto en que termina la anterior:

```
$ kairos validate courier-time-domain.pddl courier-time-p01.pddl \
    courier-time-p01-tight.plan
courier-time-p01-tight.plan:3: error: step 2 (move r1 lobby pharmacy)
      at 4.000: at start condition false: (robot-at r1 lobby) is
     false [validate.precondition_failed]
 3 | 4.000: (move r1 lobby pharmacy) [6.000]
plan invalid
```

En 4,000 el robot todavía no está en el vestíbulo para nadie que lo necesite allí. En la otra dirección hay todo el margen que se quiera: un plan puede dejar pausas, y `courier-time-p01-slow.plan`, en el que el robot descansa un segundo o más entre acciones, da `plan valid, makespan 62`. Nada en un plan temporal obliga a una acción a empezar en cuanto puede.

Un plan cambia el mundo solo en sus acontecimientos: los inicios y los finales de sus acciones. Los acontecimientos que interfieren deben estar separados por épsilon (0,001 en Kairos); los demás pueden compartir instante.

10.4. Condiciones en el tiempo

Volvamos al robot que coge una caja mientras sale. La Tabla 10.1 muestra que no se ha roto ninguna regla. La recogida del Listado 10.1 pide que el robot esté en la habitación `at start`, y en 20,002 el robot está allí. Un milisegundo después el robot se va, y la recogida no tiene nada que decir al respecto: solo preguntaba por un instante.

Lo que queríamos decir es que el robot se queda en la habitación *mientras* coge el artículo. Una condición que debe seguir siendo cierta durante una acción es un **invariante** de la acción, y se escribe con `over all`:

```
    :condition (and (at start (robot-at ?r ?l))
                    (over all (robot-at ?r ?l))
                    (at start (item-at ?i ?l))
                    (at start (free ?r)))
```

Con esa línea (y la misma en `drop`), el plan tramposo deja de ser un plan. Lo copiamos en un fichero, con las duraciones de la sección siguiente, y se lo pasamos al validador:

```
$ kairos validate courier-time-domain.pddl courier-time-p01.pddl \
    courier-time-p01-cheat.plan
courier-time-p01-cheat.plan:5: error: step 3 (pick r1 meds1 pharmacy)
      at 10.002: over-all condition broken after the happening at
     10.003000: (robot-at r1 pharmacy) is false [validate.
     invariant_failed]
 5 | 10.002: (pick r1 meds1 pharmacy) [2.000]
plan invalid
```

El intervalo es abierto

¿Por qué mantener la línea `at start` junto a la de `over all`? Porque `over all` significa el intervalo *estrictamente entre* el inicio y el final, sin los dos extremos. Una condición que debe cumplirse desde el primer instante hasta el último se escribe tres veces: `at start`, `over all` y `at end`. Los tres especificadores son independientes, y cada uno tiene su uso (Tabla 10.2).

Especificador	Cuándo se comprueba	Uso típico
`at start`	en el instante t	lo que la acción necesita para empezar: dónde está el robot, una pinza libre, un artículo en el estante
`over all`	en el intervalo abierto $(t, t+d)$	lo que no debe cambiar mientras la acción se ejecuta: el robot se queda, la puerta sigue abierta, el ascensor espera
`at end`	en el instante $t+d$	lo que debe ser cierto para que la acción se complete: hay alguien para recibir, el ascensor sigue en la planta

Tabla 10.2.: Los tres especificadores temporales de una condición, para una acción que empieza en t y dura d.

El intervalo abierto tiene una consecuencia visible. En los planes del modelo reparado, el robot sale en el instante mismo en que termina la recogida:

```
10.002: (pick r1 meds1 pharmacy) [2.000]
12.002: (move r1 pharmacy lobby) [6.000]
```

La recogida termina en 12,002 y el desplazamiento empieza en 12,002, sin ϵ entre ambos. El final de la recogida solo añade `(carrying r1 meds1)`; el inicio del desplazamiento solo toca `(robot-at r1 pharmacy)`, que la recogida necesitaba hasta justo antes de 12,002 y ya no en ese instante. Los dos acontecimientos no interfieren. Si eso es demasiado ajustado para una pinza real, dilo en el modelo: añade `(at end (robot-at ?r ?l))` a la recogida, y el desplazamiento tendrá que esperar ϵ. El ascensor de la Sección 10.8 hace exactamente eso.

Paso D para una acción durativa: tres preguntas para cada condición

Para cada condición de una acción que lleva tiempo, pregúntate:

1. ¿Hace falta para *empezar*? Entonces, `at start`.
2. ¿Fallaría la acción si dejara de ser cierta *a medias*? Entonces, también `over all`.
3. ¿Debe ser cierta en el instante en que la acción se *completa*?

Entonces, también `at end`.

Para `pick`: el artículo en el estante solo hace falta para empezar (la propia acción lo quita de ahí); el robot en la habitación hace falta para empezar y durante todo el intervalo; una pinza libre, solo para empezar. Los hechos que nunca cambian, como `connected`, solo necesitan `at start`: pedirlos `over all` es correcto, pero no aporta nada.

At start comprueba un instante. Lo que debe seguir siendo cierto mientras la acción se ejecuta es un invariante: over all. El intervalo de over all es abierto, así que los extremos se piden por separado.

10.5. Duraciones que dependen de la acción

Diez segundos para cada desplazamiento era un valor provisional. La planta tiene distancias (el Capítulo 8 se las dio al mensajero como la función estática `distance`) y un robot tiene una velocidad, así que un desplazamiento dura la distancia dividida por la velocidad. El lado derecho de una restricción de duración puede ser cualquier expresión numérica sobre los parámetros de la acción:

```
  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
```

La función nueva `(speed ?r)` está en metros por segundo, y `(handling-time ?r)` es el número de segundos que el robot necesita para coger o dejar algo. Ambas son estáticas: el problema las fija en `:init` y ninguna acción las cambia. Como ahora el dominio tiene `:functions`, declara también `:numeric-fluents`. El Listado 10.2 es el dominio completo, con los invariantes de la sección anterior.

```
;; The courier world in time: a move lasts distance over speed,
;; and the robot stays in the room while it picks or drops.
(define (domain courier-time)
  (:requirements :strips :typing :durative-actions
                 :numeric-fluents)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot))

  (:functions
    (distance ?from ?to - location)   ; meters
    (speed ?r - robot)                ; meters per second
    (handling-time ?r - robot))       ; seconds to pick or to drop

  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
    :condition (and (at start (robot-at ?r ?from))
                    (at start (connected ?from ?to)))
    :effect (and (at start (not (robot-at ?r ?from)))
                 (at end (robot-at ?r ?to))))

  (:durative-action pick
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration (handling-time ?r))
    :condition (and (at start (robot-at ?r ?l))
                    (over all (robot-at ?r ?l))
                    (at start (item-at ?i ?l))
```

```
                    (at start (free ?r)))
    :effect (and (at start (not (item-at ?i ?l)))
                 (at start (not (free ?r)))
                 (at end (carrying ?r ?i))))

  (:durative-action drop
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration (handling-time ?r))
    :condition (and (at start (robot-at ?r ?l))
                    (over all (robot-at ?r ?l))
                    (at start (carrying ?r ?i)))
    :effect (and (at start (not (carrying ?r ?i)))
                 (at end (item-at ?i ?l))
                 (at end (free ?r)))))
```

Listado 10.2: El mensajero en el tiempo: duraciones calculadas e invariantes (`courier-time-domain.pddl`).

El problema da los números. Estas son las líneas que el primer problema del dominio nuevo añade a la entrega del Capítulo 4; cada distancia se da en los dos sentidos, como cada conexión:

```
    (= (distance lobby corridor) 10)
    (= (distance corridor lobby) 10)
    ...
    (robot-at r1 dock) (free r1)
    (= (speed r1) 1) (= (handling-time r1) 2)
    (item-at meds1 pharmacy))
  (:goal (item-at meds1 ward-a))
  (:metric minimize (total-time)))
```

La última línea es nueva. `(total-time)` es un nombre que PDDL 2.1 reserva para el makespan del plan, para usarlo en la `:metric` de un problema temporal. `(:metric minimize (total-time))` pide el plan que termina antes. Es lo que Kairos minimiza de todos modos cuando un problema temporal no tiene métrica, pero otros planificadores no lo hacen, y quien lea el problema no debería tener que adivinarlo: a partir de ahora lo escribiremos en todos los problemas.

```
$ kairos courier-time-domain.pddl courier-time-p01.pddl
; cost = 42.005
; makespan = 42.005
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
10.002: (pick r1 meds1 pharmacy) [2.000]
12.002: (move r1 pharmacy lobby) [6.000]
18.003: (move r1 lobby corridor) [10.000]
28.004: (move r1 corridor ward-a) [12.000]
40.005: (drop r1 meds1 ward-a) [2.000]
```

Ahora cada desplazamiento dura lo que su distancia: 4 segundos de la base de carga al vestíbulo, 12 del pasillo a la sala A. El plan es honrado (el robot espera en la farmacia hasta que termina la recogida) y su makespan es la suma de las siete duraciones, 42, más cinco separaciones de un milisegundo.

Un plan debe usar la duración que da la restricción. El validador la vuelve a calcular a partir del estado en que empieza la acción:

```
$ kairos validate courier-time-domain.pddl courier-time-p01.pddl \
    courier-time-p01-short.plan
courier-time-p01-short.plan:2: error: step 1 (move r1 dock lobby) at
      0.000: duration 3.000000 does not satisfy (= ?duration (/ (
     distance ?from ?to) (speed ?r))): (= ?duration (/ (distance ?
     from ?to) (speed ?r))): 3 = 4 is false [validate.
     duration_mismatch]
```

```
 2 | 0.000: (move r1 dock lobby) [3.000]
plan invalid
```

«A partir del estado en que empieza la acción» importa cuando la expresión contiene un fluente que cambia. Aquí todo es estático, así que cada desplazamiento instanciado tiene una única duración para siempre. Una duración también puede depender de números dinámicos (un repostaje que dura más cuanto más vacío está el depósito); el planificador evalúa entonces la expresión al inicio de cada acción. Y la duración puede dejarse para que el planificador la elija entre unos límites, que es uno de los temas del Capítulo 11.

Una duración puede ser cualquier expresión numérica de los parámetros: distancia entre velocidad. Se evalúa cuando la acción empieza. La métrica total-time es el makespan.

10.6. Trabajar en paralelo

Para que dos robots trabajen a la vez no hace falta nada nuevo. En un plan temporal, las acciones se solapan salvo que algo del modelo las mantenga separadas: una interferencia entre sus acontecimientos o una condición que una de ellas destruye. Basta con dar al planificador el problema de la Figura 10.1 y ver qué hace. Primero, solo con `r1` (`courier-time-p02.pddl`):

```
$ kairos courier-time-domain.pddl courier-time-p02.pddl
; cost = 100.01100000000002
; makespan = 100.011
...
; better plan 2
; cost = 95.01000000000002
; makespan = 95.010
...
18.003: (move r1 lobby corridor) [10.000]
28.004: (drop r1 meds1 corridor) [2.000]
30.004: (move r1 corridor ward-a) [12.000]
...
66.007: (move r1 lab corridor) [8.000]
74.008: (pick r1 meds1 corridor) [2.000]
76.008: (move r1 corridor ward-b) [12.000]
88.009: (move r1 ward-b icu) [5.000]
93.010: (drop r1 meds1 icu) [2.000]
```

Como en el Capítulo 9, Kairos sigue buscando después del primer plan e imprime cada plan mejor. El primero lleva los medicamentos hasta la UCI, vuelve a por la muestra y necesita 100 segundos. El segundo es más listo de lo que parece: el robot deja los medicamentos en el suelo del pasillo al pasar, se ocupa de la muestra y vuelve a coger los medicamentos cuando pasa de nuevo por allí. Dos acciones más, cinco segundos menos. Si un hospital quiere medicamentos abandonados en un pasillo es asunto del paso M y de las restricciones de capítulos posteriores; el modelo, tal como está escrito, lo permite.

Ahora con el segundo robot (`courier-time-p03.pddl`), que empieza en el pasillo y se mueve a 2 metros por segundo. El primer plan que se encuentra es el dibujado en la parte de abajo de la Figura 10.2:

```
$ kairos courier-time-domain.pddl courier-time-p03.pddl --no-anytime
; cost = 47.006
; makespan = 47.006
0.000: (move r2 corridor ward-a) [6.000]
```

```
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
6.001: (pick r2 sample1 ward-a) [2.000]
8.001: (move r2 ward-a corridor) [6.000]
10.002: (pick r1 meds1 pharmacy) [2.000]
12.002: (move r1 pharmacy lobby) [6.000]
14.002: (move r2 corridor lab) [4.000]
18.003: (drop r2 sample1 lab) [2.000]
18.003: (move r1 lobby corridor) [10.000]
28.004: (move r1 corridor ward-b) [12.000]
40.005: (move r1 ward-b icu) [5.000]
45.006: (drop r1 meds1 icu) [2.000]
```

Las líneas de los dos robots están intercaladas por instante de inicio, y para seguir a un robot hay que saltarse las líneas del otro: aquí es donde un cronograma empieza a compensar. Cada robot hace una entrega. `r2` termina en 20,003 y después espera; el makespan, 47,006, es el tiempo de `r1`, que tiene el viaje largo. Los doce metros hasta la sala A le llevan a `r2` seis segundos: el mismo esquema de acción, otra duración.

Sin `-no-anytime`, la búsqueda continúa y, al cabo de un segundo más o menos, termina con este plan:

```
$ kairos courier-time-domain.pddl courier-time-p03.pddl
...
; better plan 4
; cost = 41.007
; makespan = 41.007
0.000: (move r1 dock lobby) [4.000]
0.000: (move r2 corridor lobby) [5.000]
4.001: (move r1 lobby corridor) [10.000]
5.001: (move r2 lobby pharmacy) [3.000]
8.002: (pick r2 meds1 pharmacy) [2.000]
10.002: (move r2 pharmacy lobby) [3.000]
13.003: (move r2 lobby corridor) [5.000]
18.004: (drop r2 meds1 corridor) [2.000]
20.004: (move r2 corridor ward-a) [6.000]
20.005: (pick r1 meds1 corridor) [2.000]
22.005: (move r1 corridor ward-b) [12.000]
26.005: (pick r2 sample1 ward-a) [2.000]
28.005: (move r2 ward-a corridor) [6.000]
34.006: (move r1 ward-b icu) [5.000]
34.006: (move r2 corridor lab) [4.000]
38.007: (drop r2 sample1 lab) [2.000]
39.007: (drop r1 meds1 icu) [2.000]
```

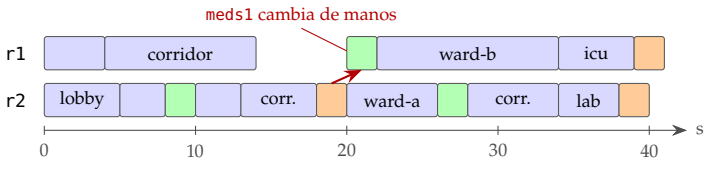


Figura 10.5.: El mejor plan encontrado para dos robots (makespan 41,007). Los desplazamientos (azul) llevan como etiqueta su destino; el verde es una recogida y el naranja una descarga. El robot rápido, `r2`, va a por los medicamentos y los deja en el pasillo en el instante 18; `r1`, que espera allí desde el 14, los coge en el 20 y los lleva a la UCI mientras `r2` se ocupa de la muestra.

La Figura 10.5 lo dibuja. El planificador ha encontrado un relevo en el que no habíamos pensado: el robot rápido hace el largo rodeo hasta la farmacia, entrega los medicamentos en el pasillo (una descarga y una recogida, porque este mensajero no tiene otra forma de hacerlo) y los dos terminan casi a la vez. Se ganan seis segundos respecto a una entrega por robot.

Tres observaciones antes de seguir.

El makespan no es el trabajo. El plan de 47 segundos contiene 67 robot-segundos de acciones; el de 41, 75. El plan mejor hace que los robots trabajen *más* para terminar *antes*. Cuando la energía importa más que el reloj, la métrica debe decirlo; el Capítulo 9 mostró cómo ponderar varias magnitudes, y `(total-time)` puede ser uno de los términos.

Sin demostración de optimalidad. Tras el último plan, Kairos informa de `solved (exhausted)`: sus búsquedas han terminado. Con planes secuenciales, a menudo puede informar de `proven optimal`; con el makespan de un plan temporal no lo afirma. En general, los planificadores temporales devuelven buenos horarios, no horarios demostradamente más cortos.

Las pausas forman parte de los planes. En la Figura 10.5, `r1` está parado en el pasillo de 14 a 20. Ninguna acción dice «espera»: un robot espera cuando no hay en curso ninguna acción suya.

Las acciones se solapan salvo que el modelo las mantenga separadas. El planificador lo aprovecha: con dos robots, el makespan baja de 95 a 47, y a 41 con un relevo que nadie había pedido.

10.7. Una cosa cada vez

«Las acciones se solapan salvo que el modelo las mantenga separadas» corta en los dos sentidos. Dos robots en dos habitaciones deben solaparse. Un robot no debe hacer dos cosas a la vez, y hasta ahora nada lo dice con carácter general. Con tres acciones hemos tenido suerte: una recogida y una descarga no pueden solaparse por la pinza (`free`), y un desplazamiento no puede solaparse con una recogida por el invariante sobre `robot-at`. A cada pareja la separa un accidente distinto.

Añadamos una cuarta acción y la suerte se acaba. El hospital equipa a los robots con una lámpara ultravioleta para desinfectar habitaciones: el robot debe quedarse en la habitación, con la lámpara encendida, durante 20 segundos.

```
;; The lamp must stay in the room for 20 seconds.
(:durative-action disinfect
  :parameters (?r - robot ?l - location)
  :duration (= ?duration 20)
  :condition (and (at start (robot-at ?r ?l))
                  (over all (robot-at ?r ?l)))
  :effect (at end (disinfected ?l))))
```

El dominio `courier-uv` es `courier-time` más esta acción. Pedimos al robot que desinfecte la farmacia y que lleve los medicamentos al vestíbulo:

```
$ kairos courier-uv-domain.pddl courier-uv-p01.pddl
; cost = 28.001
; makespan = 28.001
0.000: (pick r1 meds1 pharmacy) [2.000]
0.000: (disinfect r1 pharmacy) [20.000]
20.000: (move r1 pharmacy lobby) [6.000]
26.001: (drop r1 meds1 lobby) [2.000]
```

El robot coge la caja mientras desinfecta. Las dos acciones quieren al robot en la farmacia, y allí está; ninguna destruye lo que la otra necesita. El invariante que escribimos se cumple al pie de la letra. Lo que falta es un hecho sobre el propio robot: es una sola máquina, y cada una de sus acciones la ocupa entera.

El remedio habitual trata al robot como un **recurso** que una acción toma y devuelve. Un predicado `(idle ?r)` dice que el robot no está haciendo nada. Cada acción del robot lo necesita al inicio, lo borra al inicio y lo vuelve a añadir al final:

```
  (:durative-action disinfect
    :parameters (?r - robot ?l - location)
    :duration (= ?duration 20)
    :condition (and (at start (idle ?r))
                    (at start (robot-at ?r ?l))
                    (over all (robot-at ?r ?l)))
    :effect (and (at start (not (idle ?r)))
                 (at end (disinfected ?l))
                 (at end (idle ?r)))))
```

Mientras se ejecuta una acción de `r1`, `(idle r1)` es falso y no puede empezar ninguna otra acción de `r1`. El Listado 10.3 lo aplica a las cuatro acciones; el problema añade `(idle r1)` a su estado inicial.

```
;; The timed courier that does one thing at a time: every
;; action takes the token (idle ?r) at its start and returns it
;; at its end.
(define (domain courier-idle)
  (:requirements :strips :typing :durative-actions
                 :numeric-fluents)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (idle ?r - robot)
    (disinfected ?l - location))

  (:functions
    (distance ?from ?to - location)   ; meters
    (speed ?r - robot)                ; meters per second
    (handling-time ?r - robot))       ; seconds to pick or to drop

  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
    :condition (and (at start (idle ?r))
                    (at start (robot-at ?r ?from))
                    (at start (connected ?from ?to)))
    :effect (and (at start (not (idle ?r)))
                 (at start (not (robot-at ?r ?from)))
                 (at end (robot-at ?r ?to))
                 (at end (idle ?r))))

  (:durative-action pick
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration (handling-time ?r))
    :condition (and (at start (idle ?r))
                    (at start (robot-at ?r ?l))
                    (over all (robot-at ?r ?l))
                    (at start (item-at ?i ?l))
                    (at start (free ?r)))
    :effect (and (at start (not (idle ?r)))
                 (at start (not (item-at ?i ?l)))
                 (at start (not (free ?r)))
                 (at end (carrying ?r ?i))
                 (at end (idle ?r))))

  (:durative-action drop
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration (handling-time ?r))
```

```
   :condition (and (at start (idle ?r))
                   (at start (robot-at ?r ?l))
                   (over all (robot-at ?r ?l))
                   (at start (carrying ?r ?i)))
   :effect (and (at start (not (idle ?r)))
                (at start (not (carrying ?r ?i)))
                (at end (item-at ?i ?l))
                (at end (free ?r))
                (at end (idle ?r))))

 ;; The lamp must stay in the room for 20 seconds.
 (:durative-action disinfect
   :parameters (?r - robot ?l - location)
   :duration (= ?duration 20)
   :condition (and (at start (idle ?r))
                   (at start (robot-at ?r ?l))
                   (over all (robot-at ?r ?l)))
   :effect (and (at start (not (idle ?r)))
                (at end (disinfected ?l))
                (at end (idle ?r)))))
```

```
$ kairos courier-idle-domain.pddl courier-idle-p01.pddl
; cost = 30.003000000000004
; makespan = 30.003
0.000: (pick r1 meds1 pharmacy) [2.000]
2.001: (disinfect r1 pharmacy) [20.000]
22.002: (move r1 pharmacy lobby) [6.000]
28.003: (drop r1 meds1 lobby) [2.000]
```

Listado 10.3: El mensajero que hace una sola cosa cada vez (`courier-idle-domain.pddl`).

Una acción tras otra, y treinta segundos, como exige el mundo. Fíjate en los tiempos: ahora cada acción empieza ϵ después de que termine la anterior, incluso el desplazamiento que sigue a la desinfección, que antes empezaba en el mismo instante. El final de cada acción añade `(idle r1)` y el inicio de la siguiente lo necesita, así que los dos acontecimientos interfieren. Un testigo que pasa de acción en acción cuesta un milisegundo por cada pase.

Este es el patrón *cerrojo de recurso* del Capítulo 7, con una diferencia. Con acciones instantáneas, un plan secuencial da la exclusión mutua gratis, y un testigo solo hace falta para reservar algo *a lo largo de* varias acciones. Con acciones durativas, el testigo hace falta durante una sola acción. El mismo patrón protege cualquier cosa que sirva a un usuario cada vez: una puerta estrecha, un cargador, un mostrador de farmacia, un fogón. El testigo pertenece entonces a la puerta o al fogón, no al robot (la cocina de la sección siguiente tiene uno).

¿Siguen mereciendo su sitio los invariantes de la Sección 10.4 una vez que cada acción retiene `idle`? En este dominio están implícitos: un robot que no está ocioso no puede empezar un desplazamiento. Los mantenemos porque dicen algo distinto. El testigo dice que el *robot* hace una sola cosa cada vez. El invariante dice en qué se apoya la *acción* en el mundo, y seguirá protegiéndola cuando algo distinto del propio robot pueda estropearla: otro robot que mueve el estante, una puerta que se cierra, una habitación que cierra a las ocho.

Un robot es un recurso: cada acción toma el testigo idle al empezar y lo devuelve al terminar. Los invariantes dicen en qué se apoya una acción; el testigo dice que el agente es uno solo.

10.8. Otros dominios: una cocina y un ascensor

Una cocina: cosas que funcionan solas

En el mundo del mensajero, cada acción pertenece a un robot. En una cocina, parte del trabajo no lo hace nadie: una vez que la olla está en el fogón, la sopa se cuece sola y la cocinera queda libre para otra cosa. Un buen horario prepara el plato siguiente mientras se cuece el anterior. El Listado 10.4 lo modela con tres acciones. `prepare` y `serve` ocupan a la cocinera, mediante el testigo `available`; `cook` solo ocupa un fogón, mediante `burner-free`, y no tiene ninguna cocinera entre sus parámetros. Las duraciones dependen del plato y se leen como minutos.

```
;; A kitchen: a cook prepares dishes, burners cook them alone,
;; and the cook serves them. Times are in minutes.
(define (domain kitchen)
  (:requirements :strips :typing :durative-actions
                 :numeric-fluents)

  (:types cook dish burner)

  (:predicates
    (raw ?d - dish)
    (prepared ?d - dish)
    (cooked ?d - dish)
    (served ?d - dish)
    (available ?c - cook)        ; the cook has nothing in hand
    (burner-free ?b - burner))

  (:functions
    (prep-time ?d - dish)
    (cook-time ?d - dish))

  ;; Washing and chopping keeps the cook busy.
  (:durative-action prepare
    :parameters (?c - cook ?d - dish)
    :duration (= ?duration (prep-time ?d))
    :condition (and (at start (raw ?d))
                    (at start (available ?c)))
    :effect (and (at start (not (raw ?d)))
                 (at start (not (available ?c)))
                 (at end (prepared ?d))
                 (at end (available ?c))))

  ;; The pot is on the burner; nobody has to watch it.
  (:durative-action cook
    :parameters (?d - dish ?b - burner)
    :duration (= ?duration (cook-time ?d))
    :condition (and (at start (prepared ?d))
                    (at start (burner-free ?b)))
    :effect (and (at start (not (prepared ?d)))
                 (at start (not (burner-free ?b)))
                 (at end (cooked ?d))
                 (at end (burner-free ?b))))

  (:durative-action serve
    :parameters (?c - cook ?d - dish)
    :duration (= ?duration 1)
    :condition (and (at start (cooked ?d))
                    (at start (available ?c)))
    :effect (and (at start (not (cooked ?d)))
                 (at start (not (available ?c)))
                 (at end (served ?d))
                 (at end (available ?c)))))
```

Listado 10.4: Una cocina (`kitchen-domain.pddl`).

Una sopa (6 minutos de preparación y 20 de cocción), pasta (1 y 10) y una salsa (4 y 8), una cocinera y un fogón:

```
$ kairos kitchen-domain.pddl kitchen-p01.pddl
...
; better plan 4
; cost = 40.004
; makespan = 40.004
0.000: (prepare ana pasta) [1.000]
1.001: (cook pasta b1) [10.000]
1.001: (prepare ana soup) [6.000]
11.002: (cook soup b1) [20.000]
11.002: (serve ana pasta) [1.000]
12.003: (prepare ana sauce) [4.000]
31.003: (cook sauce b1) [8.000]
31.003: (serve ana soup) [1.000]
39.004: (serve ana sauce) [1.000]
```

El fogón nunca descansa: los tres tiempos de cocción suman 38 minutos, y antes del primero hay que preparar algo (al menos 1 minuto, la pasta) y después del último servir algo (1 minuto). Ningún horario con un solo fogón puede bajar de 40, y el planificador lo alcanzó en su cuarto intento (el primero tardaba 46). La cocinera trabaja en los huecos. Con un segundo fogón (`kitchen-p02.pddl`), el makespan baja a 28,003; la Figura 10.6 dibuja ambos casos.

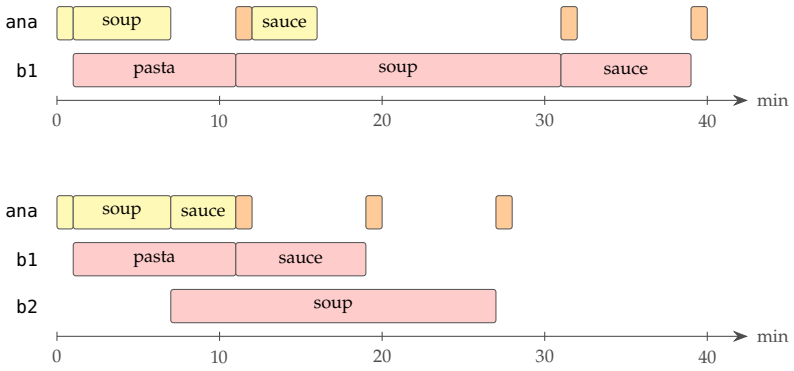


Figura 10.6.: La cocina con un fogón (arriba, 40 minutos) y con dos (abajo, 28). El amarillo es preparar; el rojo, cocinar; el naranja, servir. Las filas de los fogones muestran acciones que no realiza ningún agente; con un fogón, esa fila no tiene huecos, y es el fogón el que fija el makespan.

Un cronograma muestra de un vistazo qué recurso es el cuello de botella: la fila sin huecos. Con un fogón es el fogón, y una segunda cocinera no ayudaría (el Ejercicio 10.8.1 lo comprueba). Con dos fogones no hay nada saturado, y el makespan lo fija la cadena más larga: preparar la pasta, preparar la sopa, cocinar la sopa y servirla, $1 + 6 + 20 + 1$.

Un ascensor: esperar a las personas

El segundo dominio tiene el sabor contrario: un invariante que *permite* compartir. Un ascensor viaja entre plantas contiguas; los pasajeros suben y bajan, cada uno a su ritmo. Mientras alguien sube, el ascensor debe quedarse en la planta. Pero varios pasajeros pueden subir a la vez, así que el ascensor no es un testigo que se toma: es un hecho en el que se apoya cada subida.

```
;; Elevators: a lift travels between adjacent floors and must
;; wait at the floor until passengers have boarded or left.
(define (domain elevators)
  (:requirements :strips :typing :durative-actions
                 :numeric-fluents)

  (:types lift passenger floor)

  (:predicates
```

```
    (adjacent ?a ?b - floor)
    (lift-at ?e - lift ?f - floor)
    (passenger-at ?p - passenger ?f - floor)
    (boarded ?p - passenger ?e - lift))

  (:functions
    (floor-time ?e - lift)        ; seconds per floor
    (step-time ?p - passenger))   ; seconds to board or to leave

  (:durative-action travel
    :parameters (?e - lift ?from ?to - floor)
    :duration (= ?duration (floor-time ?e))
    :condition (and (at start (lift-at ?e ?from))
                    (at start (adjacent ?from ?to)))
    :effect (and (at start (not (lift-at ?e ?from)))
                 (at end (lift-at ?e ?to))))

  (:durative-action board
    :parameters (?p - passenger ?e - lift ?f - floor)
    :duration (= ?duration (step-time ?p))
    :condition (and (at start (passenger-at ?p ?f))
                    (at start (lift-at ?e ?f))
                    (over all (lift-at ?e ?f))
                    (at end (lift-at ?e ?f)))
    :effect (and (at start (not (passenger-at ?p ?f)))
                 (at end (boarded ?p ?e))))

  (:durative-action leave
    :parameters (?p - passenger ?e - lift ?f - floor)
    :duration (= ?duration (step-time ?p))
    :condition (and (at start (boarded ?p ?e))
                    (at start (lift-at ?e ?f))
                    (over all (lift-at ?e ?f))
                    (at end (lift-at ?e ?f)))
    :effect (and (at start (not (boarded ?p ?e)))
                 (at end (passenger-at ?p ?f)))))
```

Listado 10.5: Ascensores (`elevators-domain.pddl`).

`board` y `leave` piden `(lift-at ?e ?f)` tres veces: al inicio, durante todo el intervalo y al final. El ascensor debe estar allí desde el primer paso del pasajero hasta el último, ambos incluidos, para que no pueda salir en el instante mismo en que alguien termina de entrar. Tres pasajeros bajan a la planta baja; Ada y Bob esperan en la segunda y Eve en la tercera, y Bob necesita 4 segundos para subir al ascensor, mientras que los demás necesitan 2:

```
$ kairos elevators-domain.pddl elevators-p01.pddl
; cost = 28.008
; makespan = 28.008
0.000: (travel e1 f0 f1) [3.000]
3.001: (travel e1 f1 f2) [3.000]
6.002: (board bob e1 f2) [4.000]
6.002: (board ada e1 f2) [2.000]
10.003: (travel e1 f2 f3) [3.000]
13.004: (board eve e1 f3) [2.000]
15.005: (travel e1 f3 f2) [3.000]
18.006: (travel e1 f2 f1) [3.000]
21.007: (travel e1 f1 f0) [3.000]
24.008: (leave eve e1 f0) [2.000]
24.008: (leave ada e1 f0) [2.000]
24.008: (leave bob e1 f0) [4.000]
```

Ada y Bob suben juntos en 6,002. Ada está dentro en 8,002, pero el ascensor no puede arrancar antes de que Bob esté dentro, en 10,002; por la condición `at end`, la salida llega ϵ después, en 10,003. En la planta baja salen los tres a la vez, y el plan termina cuando ha salido el más lento. Un segundo ascensor que espera en la tercera planta (`elevators-p02.pddl`)

reduce el makespan a 19,005, y lo hace él solo: el plan nunca mueve el primer ascensor, que llegaría más tarde que el segundo cuando este pasa de bajada. Más recursos no siempre significan más trabajo en paralelo.

Un testigo (available, burner-free) mantiene separados a los usuarios de un recurso. Un invariante (lift-at over all) les permite compartirlo. La fila sin huecos de un cronograma es el cuello de botella.

10.9. El método MODEL: tiempo

Las acciones durativas dejan los cinco pasos como estaban y añaden preguntas a dos de ellos, más una línea nueva en la ficha del modelo.

Lo que este capítulo añade al método

Paso M (ficha del modelo). En *Éxito*: ¿importa *cuándo* se alcanza el objetivo? Si la respuesta es «lo antes posible», el modelo necesita tiempo y la métrica (`total-time`). En *Agentes*: ¿quién puede actuar a la vez que quién?
Paso D (tabla de acciones). Para cada acción, cuatro columnas nuevas:

- *¿Importa la duración, y de qué depende?* De una constante, de los parámetros (distancia entre velocidad), del estado.
- *¿Qué debe cumplirse al inicio, durante todo el intervalo y al final?* Las tres preguntas de la Sección 10.4, para cada condición.
- *¿Cuándo ocurre cada efecto?* Lo que la acción quita, al inicio; lo que produce, al final.
- *¿Qué puede ejecutarse en paralelo con ella?* Lo que no deba necesita un testigo (el agente, una herramienta, un lugar) que se toma al inicio y se devuelve al final.

Paso E (batería de pruebas). La prueba de la trampa se convierte en una familia: pide un agente en dos sitios a la vez; pide un objetivo cuyo horario honrado más corto conozcas y compara el makespan; lee cada par de líneas solapadas de un plan y pregúntate si el mundo permite ese solapamiento.

La Tabla 10.3 es la tabla de acciones del mensajero del Listado 10.3 con las columnas nuevas. Escribirla antes del PDDL es lo que nos habría ahorrado los dos planes tramposos de este capítulo.

Acción	Duración	Necesita (I inicio, D durante)	Cambia (I inicio, F final)
`move`	distancia / velocidad	I: robot ocioso, en `?from`, habitaciones conectadas	I: no ocioso, no en `?from`. F: en `?to`, ocioso
`pick`	tiempo de manipulación	I: ocioso, artículo aquí, pinza libre. I, D: robot aquí	I: no ocioso, artículo fuera, pinza ocupada. F: lo lleva, ocioso
`drop`	tiempo de manipulación	I: ocioso, lo lleva. I, D: robot aquí	I: no ocioso, no lo lleva. F: artículo aquí, pinza libre, ocioso
`disinfect`	20	I: ocioso. I, D: robot aquí	I: no ocioso. F: habitación desinfectada, ocioso

Tabla 10.3.: La tabla de acciones del mensajero con tiempo: el paso D con las columnas que añade el tiempo.

Alternativas comparadas

Acciones instantáneas con coste o acciones durativas

La alternativa anunciada en la Sección 10.1 merece un juicio justo. El dominio `courier-cost` conserva las acciones instantáneas del Capítulo 4 y hace que cada una sume a `(total-cost)` los segundos que tardaría, como enseñó el Capítulo 9:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (increase (total-cost)
                         (/ (distance ?from ?to) (speed ?r)))))
```

Sobre los dos robots y las dos entregas de `courier-time-p03`, minimizando el coste:

```
$ kairos courier-cost-domain.pddl courier-cost-p01.pddl
; cost = 52.5
(move r2 corridor ward-a)
(pick r2 sample1 ward-a)
(move r2 ward-a corridor)
(move r2 corridor lab)
(drop r2 sample1 lab)
(move r2 lab corridor)
...
(move r2 ward-b icu)
(drop r2 meds1 icu)
```

Las catorce acciones son de `r2`. El plan es óptimo, y Kairos lo demuestra. Es también lo contrario de lo que queríamos: el robot rápido lo hace todo y `r1` no sale nunca de la base de carga, porque cada segundo de `r1` costaría lo mismo que dos de `r2`. Para el menor trabajo total, la respuesta es correcta. Como horario, tarda 52,5 segundos, frente a los 41 de la Figura 10.5. La Tabla 10.4 compara las dos codificaciones según los cinco criterios.

	Acciones instantáneas con coste (`courier-cost`)	Acciones durativas (`courier-time`)
Fidelidad	Los planes se ejecutan acción a acción. El coste es el trabajo total: 52,5 s. No puede decir cuándo se alcanza el objetivo, qué se solapa ni que algo debe cumplirse *mientras* se ejecuta una acción	Los planes son horarios: makespan 41,007 con los dos robots. Se usará cada solapamiento que deje abierto el modelo, así que los invariantes y los testigos tienen que estar bien
Tamaño	92 acciones instanciadas, 38 hechos	92 acciones durativas instanciadas, 38 hechos; el motor que gestiona solapamientos divide cada una en un inicio y un final (184)
Guía	Fácil: 3336 estados, plan óptimo demostrado en 13 ms	Primer plan en 2 ms, el mejor tras 0,9 s y 161 871 estados; sin demostración de que sea el más corto
Claridad	Una lista de precondiciones y otra de efectos por acción	Cada condición y cada efecto exigen una decisión (inicio, durante, final); la tabla de acciones gana cuatro columnas
Portabilidad	Cualquier planificador con fluentes numéricos; con costes constantes, cualquier planificador clásico con `:action-costs`	Solo planificadores temporales: Kairos, OPTIC, POPF, TFD, LPG-td. No ENHSP, Metric-FF ni LAMA

Tabla 10.4.: Dos codificaciones de «las acciones llevan tiempo», ejecutadas sobre el mismo problema (dos robots, dos entregas).

Veredicto. Usa un coste cuando actúa un solo agente, o cuando lo que importa es el esfuerzo total (energía, dinero, desgaste) y no el reloj: el modelo es más sencillo, la búsqueda más fácil y el resultado puede demostrarse óptimo. Usa acciones durativas cuando pueden ocurrir varias cosas a la vez y el instante de finalización importa, o cuando una condición debe cumplirse mientras ocurre algo. Una buena práctica es empezar con el modelo de costes, dejar bien la lógica gracias a la respuesta rápida que da y después añadir el tiempo; el Capítulo 19 vuelve sobre cómo hacer crecer un modelo por versiones.

Dónde va un efecto: al inicio, al final o repartido

La segunda elección se hace dentro de cada acción durativa. `courier-walk` es el mensajero más pequeño (un robot, solo `move`, 10 segundos por desplazamiento) en tres versiones que se diferencian en dos palabras: los dos efectos al final, los dos al inicio, o el borrado al inicio y la adición al final, como en todo este capítulo. Cada versión se ejecuta sobre un recorrido de tres desplazamientos de la base de carga a la sala A y sobre una prueba de la trampa: estar en la farmacia y en la base de carga a la vez, partiendo del vestíbulo.

	Ambos al final	Ambos al inicio	Repartidos
Mientras se desplaza, el robot está	aún en `?from`	ya en `?to`	en ningún sitio
Recorrido de tres desplazamientos	30,002	**10,002**	30,002
En dos sitios a la vez	**un plan, 10,001**	sin plan	sin plan

Tabla 10.5.: Tres colocaciones de los dos efectos de `move`, cada una ejecutada sobre los mismos dos problemas (`courier-walk-p01` y `-p02`). En negrita, los planes que el mundo no permite.

Con los dos efectos al final, el robot sigue en el vestíbulo mientras se desplaza, así que puede empezar desde allí un segundo desplazamiento:

```
$ kairos courier-walk-atend-domain.pddl courier-walk-p02.pddl
; cost = 10.001
; makespan = 10.001
0.000: (move r1 lobby dock) [10.000]
0.001: (move r1 lobby pharmacy) [10.000]
```

Con los dos al inicio, el robot está en la habitación siguiente antes de haber andado, y los tres desplazamientos del recorrido se ejecutan casi uno encima de otro:

```
$ kairos courier-walk-atstart-domain.pddl courier-walk-p01.pddl
; cost = 10.002
; makespan = 10.002
0.000: (move r1 dock lobby) [10.000]
0.001: (move r1 lobby corridor) [10.000]
0.002: (move r1 corridor ward-a) [10.000]
```

Veredicto. Borra al inicio lo que la acción consume y añade al final lo que produce. Durante la acción, la cosa no está entonces en ninguno de los dos estados, y nadie puede usarla dos veces (el fallo de «ambos al final») ni usarla antes de que exista (el fallo de «ambos al inicio»). Esta regla decide la mayoría de los efectos. Tiene excepciones razonadas, que el Capítulo 11 usa a propósito: una acción que añade un hecho al empezar y lo borra al terminar hace que el hecho sea cierto exactamente mientras se ejecuta.

Un modelo de costes responde a «el menor trabajo total» y lo demuestra; un modelo temporal responde a «terminar lo antes posible» y deja al descubierto cada solapamiento. Dentro de una acción durativa: tomar al inicio, dar al final.

10.10. Errores comunes

Traducir una acción cambiando el nombre de sus partes

La forma más rápida de hacer durativa una acción es añadir un `:duration` y cambiar la palabra clave. No funciona: las condiciones y los efectos sin especificador temporal no son PDDL.

```
$ kairos check wrong-untimed-domain.pddl
wrong-untimed-domain.pddl:15:21: error: in durative action 'move
     ': conditions of durative actions must be under at start,
     at end or over all [semantic.untimed_condition]
 15 |     :condition (and (robot-at ?r ?from)
    |                     ^
...
invalid
```

Lo mismo ocurre con los efectos (`semantic.untimed_effect`), con `:precondition` en lugar de `:condition` (`unknown key :precondition in durative action`) y con una duración olvidada (`durative action without :duration`). El error es un favor: cada especificador es una decisión de modelado, y el comprobador no va a tomarla por ti.

Comprobar al inicio lo que debe cumplirse todo el tiempo

Síntoma: un plan en el que un agente empieza algo y se va, como la recogida en 20,002 seguida de un desplazamiento en 20,003 de la Sección 10.2. El planificador no se equivoca; una condición `at start` habla de un solo instante. **Solución:** añade la condición `over all`. **Cómo detectarlo:** busca líneas que se solapen en el plan y un makespan más corto que la suma que esperabas; escribe el plan sospechoso en un fichero con el dominio reparado y deja que `kairos validate` confirme que ahora se rechaza.

Un testigo olvidado en una acción

El testigo `idle` solo funciona si *todas* las acciones del agente lo toman y lo devuelven. Si se olvida en una acción, esa acción se solapa con todas las demás; si solo se olvida devolverlo, el agente no vuelve a actuar nunca, lo que suele manifestarse como un problema sin plan. Nada en el lenguaje liga el predicado al agente, así que ningún comprobador avisará. Pruébalo: para cada acción, un problema pequeño cuyo plan honrado más rápido puedas calcular a mano (el Ejercicio 10.7.1 es uno).

Una duración que no se puede calcular

Una duración leída de una función necesita un valor para cada acción instanciada que deba existir. En `courier-time-p05.pddl`, la distancia de la base de carga al vestíbulo se daba solo en un sentido:

```
$ kairos courier-time-domain.pddl courier-time-p05.pddl
note: the goal needs atoms that no action can reach
(item-at meds1 ward-a) can never be reached:
  drop adds it, but needs (robot-at ?r - robot ward-a), which
     never holds
    move adds it, but is never applicable (types, negative or
     numeric conditions) [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

Un desplazamiento cuya duración está indefinida no puede ejecutarse, y el robot nunca sale de la base de carga. Los ficheros son PDDL válido, así que `kairos check` no dice nada; la pista es «numeric conditions» en la nota. Da las distancias en los dos sentidos, como las conexiones.

Leer las líneas de un plan temporal como una secuencia

Las líneas están ordenadas por instante de inicio y nada más. Una acción impresa después de otra puede haber empezado en el mismo instante, y puede terminar antes que ella. Antes de juzgar un plan con más de un agente, dibuja su cronograma, con una fila por agente o recurso. Y no esperes números redondos: las separaciones de ϵ forman parte del plan, y un plan escrito a mano sin ellas se rechaza (`at start condition false`, Sección 10.3).

Declarar menos de lo que se usa

Kairos acepta con un aviso un dominio con acciones durativas que no tiene `:durative-actions` entre sus requisitos:

```
$ kairos check wrong-norequirement-domain.pddl
wrong-norequirement-domain.pddl:4:18: warning: uses durative-
     actions but :requirements does not include :durative-
     actions [semantic.undeclared_requirement]
 4 |   (:requirements :strips :typing)
   |                  ^
valid, plannable
...
```

Otros planificadores rechazan el fichero, o lo leen mal sin decir nada. Lo mismo vale para `:numeric-fluents` cuando las duraciones salen de funciones.

10.11. Resumen

Una acción durativa tiene un inicio, un final y una duración. Sus condiciones dicen cuándo deben cumplirse (`at start`, `over all`, `at end`), y sus efectos, cuándo ocurren (`at start`, `at end`). Un plan temporal da a cada acción un instante de inicio; las acciones se solapan siempre que el modelo no las mantenga separadas, y el planificador aprovecha todo solapamiento que quede abierto. Modelar con tiempo es, en gran medida, decidir qué solapamientos permite el mundo.

Concepto	Qué recordar
`:durative-action`	`:duration`, `:condition`, `:effect`; requisito `:durative-actions`
`(= ?duration e)`	`e` es un número o una expresión (distancia entre velocidad), evaluada cuando la acción empieza
`at start`, `at end`	un instante cada uno; para condiciones y para efectos
`over all`	el intervalo abierto entre ambos; solo condiciones; un invariante
Plan temporal	líneas `t: (action) [d]`; el orden de las líneas no es un orden de ejecución
Makespan	el final de la última acción; `(:metric minimize (total-time))`
Acontecimiento	un instante en que empiezan o terminan acciones; los únicos momentos de cambio
ϵ	la separación entre acontecimientos que interfieren; 0,001 en Kairos, `-epsilon`
Tomar al inicio, dar al final	la colocación por defecto de los efectos; lo que está en uso no está en ninguno de los dos estados
Testigo (`idle`)	se toma al inicio y se devuelve al final: una sola cosa cada vez para un agente o un recurso

Tabla 10.6.: Lo que ha añadido este capítulo.

Cada plan de este capítulo podría haberlo ejecutado un robot paciente, acción a acción: el solapamiento hacía los planes más cortos, nunca posibles. El Capítulo 11 pasa a problemas en los que solo existe un plan *si* las acciones se solapan, y a duraciones que elige el planificador.

10.12. Lecturas complementarias

La referencia de las acciones durativas es el artículo de PDDL 2.1 de Fox y Long (2003): sus secciones sobre acciones durativas definen los tres especificadores temporales, los acontecimientos, la interferencia y la regla de «no moving targets» que este capítulo ha explicado con ejemplos. Haslum et al. (2019) dedican un capítulo a la planificación temporal en PDDL, con la misma semántica contada de forma más formal y un repaso de cómo la tratan los planificadores. Para la planificación temporal más allá de PDDL (líneas de tiempo, crónicas, redes temporales), consulta los capítulos sobre el tiempo de Ghallab et al. (2004) y Ghallab et al. (2016). Los planes se comprueban frente a la semántica con validadores; VAL (Howey et al., 2004) es el clásico, y `kairos validate` sigue las mismas reglas. Cómo elige Kairos entre sus motores temporales se describe en Martín Rico et al. (2026); los planificadores POPF (Coles et al., 2010) y OPTIC (Benton et al., 2012), que aparecen en las filas de portabilidad de esta parte del libro, son las referencias habituales para la planificación con acciones durativas mediante búsqueda hacia delante.

10.13. Ejercicios

Ejercicios de la Sección 10.1 — Lo que una secuencia no puede decir

Ejercicio 10.1.1. Un plan secuencial tiene nueve acciones. Las del robot `a` duran 5, 3, 8 y 2 segundos, en ese orden; las del robot `b` duran 4, 4, 6, 1 y 7. Los robots no se necesitan nunca el uno al otro. ¿Cuánto tarda el plan si las acciones se ejecutan una tras otra? ¿Y si cada robot trabaja por su cuenta? Dibuja el cronograma del segundo caso.

Ejercicio 10.1.2. Para cada situación, di si basta con un coste que sume duraciones o si el modelo necesita tiempo, y por qué: (a) una única furgoneta de reparto que debe recorrer el menor número de kilómetros; (b) dos grúas que deben descargar un barco antes de que baje la marea; (c) una cena en la que todos los platos deben llegar calientes a la mesa; (d) un robot que debe gastar la menor batería posible.

Ejercicios de la Sección 10.2 — Una primera acción durativa

Ejercicio 10.2.1. En `courier-fixed-domain.pddl`, cambia la duración de `pick` de 2 a 5 segundos. Antes de ejecutar el planificador sobre `courier-fixed-p01`, predice el makespan. Después ejecútalo y explica el resultado.

Ejercicio 10.2.2. Desayuno: una tostadora tarda 120 segundos en tostar el pan que tiene dentro, y un hervidor tarda 180 segundos en hervir el agua que contiene. Cada aparato hace una sola cosa cada vez y funciona solo una vez puesto en marcha. Escribe el dominio con dos acciones durativas sin parámetros y un problema que pida tostadas y agua caliente. ¿Qué makespan esperas?

Ejercicio 10.2.3. Un compañero convierte `move` en una acción durativa así, y `kairos check` la rechaza. ¿Qué líneas están mal y qué dice el comprobador?

```
    :condition (and (at start (robot-at ?r ?from))
                    (at start (connected ?from ?to)))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to))))
```

Ejercicios de la Sección 10.3 — Qué ocurre en cada instante

Ejercicio 10.3.1. Estas son cuatro líneas consecutivas del plan del ascensor de la Sección 10.8. Enumera sus acontecimientos por orden temporal. Para cada par de acontecimientos consecutivos, di si interfieren y a través de qué hecho.

```
3.001: (travel e1 f1 f2) [3.000]
6.002: (board bob e1 f2) [4.000]
6.002: (board ada e1 f2) [2.000]
10.003: (travel e1 f2 f3) [3.000]
```

Ejercicio 10.3.2. El plan de `courier-time-p01` tiene un makespan de 42,005. Predice su makespan con `-epsilon 0.1` y compruébalo. ¿Por qué la respuesta no es $42 + 6 \times 0{,}1$, aunque el plan tenga siete acciones?

Ejercicios de la Sección 10.4 — Condiciones en el tiempo

Ejercicio 10.4.1. En `courier-time-domain.pddl`, quita de `pick` la condición que pide el robot en la habitación `at start` y mantén la línea `over all`. ¿Puede hacer trampa ahora el robot? Ejecuta `courier-time-p01` y compara los tiempos del plan con los de la Sección 10.5.

Ejercicio 10.4.2. En `elevators-domain.pddl`, quita las condiciones `over all` y `at end` de `board` y `leave`. Predice qué hará el plan de `elevators-p01`, ejecútalo y describe el viaje de Bob.

Ejercicios de la Sección 10.5 — Duraciones que dependen de la acción

Ejercicio 10.5.1. En `courier-time-p01`, el robot se sustituye por uno que se mueve a 2 metros por segundo. Calcula a mano el nuevo makespan (con $\epsilon = 0{,}001$) y después compruébalo.

Ejercicio 10.5.2. Un robot que lleva algo se mueve más despacio. Amplía `courier-time` con una función `(loaded-speed ?r)` de forma que un desplazamiento dure la distancia entre la velocidad cuando la pinza está libre, y la distancia entre la velocidad con carga cuando el robot lleva un artículo. Las duraciones no pueden depender de una condición, así que necesitarás dos acciones. Pruébalo con la entrega de `courier-time-p01` y una velocidad con carga de 0,5.

Ejercicios de la Sección 10.6 — Trabajar en paralelo

Ejercicio 10.6.1. En `courier-time-p03`, pon la velocidad de `r2` a 1. ¿Sigue encontrando un relevo el planificador? ¿Cuál es el makespan y qué robot lo determina?

Ejercicio 10.6.2. Toma la entrega única de `courier-time-p01` (medicamentos a la sala A) y añade el robot rápido `r2` en el pasillo. Predice qué robot hará la entrega y el makespan, y compruébalo. ¿Qué hace `r1` en el plan?

Ejercicios de la Sección 10.7 — Una cosa cada vez

Ejercicio 10.7.1. En una copia de `courier-idle-domain.pddl`, el testigo se olvidó en `drop`: la acción ni necesita `idle` ni lo cambia. Un robot llega a la farmacia con los medicamentos; debe dejarlos allí y desinfectar la habitación. Calcula el makespan del plan honrado, escribe el problema y averigua qué hace el planificador con el dominio defectuoso.

Ejercicio 10.7.2. El mostrador de una habitación atiende a un robot cada vez: dos robots no pueden coger cosas de la misma habitación a la vez. Añádelo a `courier-idle` con un testigo que pertenezca a la habitación. Pruébalo con dos robots y dos cajas de medicamentos en la farmacia, que hay que llevar ambas al vestíbulo, y compara con el plan del dominio sin el testigo.

Ejercicios de la Sección 10.8 — Otros dominios: una cocina y un ascensor

Ejercicio 10.8.1. En la cocina con un fogón (`kitchen-p01`), ¿acortaría el makespan una segunda cocinera? ¿Y un tercer fogón en `kitchen-p02`? Responde primero a partir de la Figura 10.6 y después comprueba ambos casos.

Ejercicio 10.8.2. En `elevators-p02`, el plan solo usa el segundo ascensor. Calcula el makespan del mejor plan en el que `e1` lleva a Ada y a Bob mientras `e2` lleva a Eve, y explica por qué el planificador hace bien en descartarlo.

Ejercicios de la Sección 10.9 — El método MODEL: tiempo

Ejercicio 10.9.1. Aplica el paso L. Para cada caso, elige entre acciones instantáneas con coste y acciones durativas, y justifícalo con los

criterios de la Tabla 10.4: (a) un único robot de almacén cuya ruta debe ser lo más corta posible; (b) tres pintores y una casa que hay que terminar lo antes posible, donde una pared no se puede pintar mientras se seca su yeso; (c) una flota de camiones cuyo combustible total se quiere minimizar, sin plazo.

Ejercicio 10.9.2. En `courier-time-domain.pddl`, mueve el efecto `(item-at ?i ?l)` de `drop` del final al inicio. ¿Qué plan que el mundo no permite aparece ahora? Ejecuta `courier-time-p03` y encuentra las dos líneas que lo muestran.

11. Coordinación en el tiempo

En el Capítulo 10, dos robots trabajaban a la vez porque así se acababa antes. Nada les obligaba: un único robot paciente podría haber ejecutado cualquiera de aquellos planes acción tras acción, y el objetivo se habría alcanzado igualmente, solo que más tarde. Ahora el laboratorio estrena una puerta con muelle. Solo se queda abierta mientras alguien la sujeta, y un robot que sujeta una puerta no puede cruzarla. Un robot tiene que sujetar la puerta *mientras* el otro pasa. De este plan no hay una versión lenta con una acción cada vez: o las dos acciones se solapan, o la muestra no entra nunca en el laboratorio. Una segunda situación es igual de frecuente. El robot está en la base de carga con la batería casi vacía, y la pregunta no es si cargar, sino durante cuánto tiempo. Con las duraciones del Capítulo 10, fijadas por el modelo, el planificador no tiene nada que decir.

Este capítulo trata de esas dos cosas: acciones que *deben* solaparse y duraciones que *elige* el planificador. La primera se llama concurrencia requerida, y divide los problemas temporales, y los planificadores temporales, en dos familias. Se modela con un patrón que se repite, una acción que mantiene cierto un hecho exactamente mientras se ejecuta. La segunda necesita una pequeña extensión del lenguaje, `:duration-inequalities`, que convierte `?duration` de un número que calcula el modelo en una decisión.

Empezamos con el ejemplo clásico, el sótano de las cerillas, donde la concurrencia requerida cabe en veinte líneas. Después el mensajero se encuentra con la puerta con muelle, y vemos qué supone la distinción para los planificadores y, en particular, para Kairos. Dos robots levantan juntos un cajón, algo más difícil de modelar bien de lo que parece. La segunda mitad del capítulo deja al planificador elegir una duración (una carga, un café), usa el valor elegido en los efectos y echa un primer vistazo a la maquinaria con la que los planificadores mantienen coherentes todos esos tiempos. El método, los errores, un resumen y los ejercicios lo cierran.

11.1 Planes que solo existen si las acciones se solapan 249
11.2 La acción envolvente: una puerta sujeta 251
11.3 Temporalmente simple, temporalmente expresivo 254
Qué hace Kairos 254
11.4 Trabajar juntos: dos robots y un cajón 255
Un primer modelo, y por qué es frágil 256
Un modelo con holgura 257
11.5 Duraciones que elige el planificador 258
`?duration` en los efectos 259
Cómo elige un planificador una duración 260
Una cafetera 261
11.6 Cómo ordenan el tiempo los planificadores 261
11.7 El método MODEL: coordinación 263
Alternativas comparadas 263
11.8 Errores comunes 265
11.9 Resumen 267
11.10 Lecturas complementarias 267
11.11 Ejercicios 268

11.1. Planes que solo existen si las acciones se solapan

Se ha fundido un fusible y el sótano está a oscuras. Tienes una caja de cerillas. Una cerilla arde durante 5 segundos; reparar un fusible lleva 2 segundos y no se puede hacer a oscuras. El Listado 11.1 es el dominio completo.[1]

```
;; The match cellar: a fuse can be mended only by the light of a
;; match, and a match burns for 5 seconds.
(define (domain cellar)
  (:requirements :strips :typing :durative-actions)

  (:types match fuse)

  (:predicates
    (unused ?m - match)
    (lit ?m - match)
    (mended ?f - fuse)
```

1: El sótano de las cerillas es un clásico de la planificación temporal; una versión suya fue *benchmark* de la competición de planificación de 2011. El fichero de aquí es nuestro, con los nombres de este libro.

```
    (hand-free))

  ;; There is light from the start of the action to its end.
  (:durative-action light
    :parameters (?m - match)
    :duration (= ?duration 5)
    :condition (at start (unused ?m))
    :effect (and (at start (not (unused ?m)))
                 (at start (lit ?m))
                 (at end (not (lit ?m)))))

  ;; Mending takes 2 seconds, all of them with light.
  (:durative-action mend
    :parameters (?f - fuse ?m - match)
    :duration (= ?duration 2)
    :condition (and (at start (hand-free))
                    (at start (lit ?m))
                    (over all (lit ?m)))
    :effect (and (at start (not (hand-free)))
                 (at end (mended ?f))
                 (at end (hand-free)))))
```

Listado 11.1: El sótano de las cerillas (`cellar-domain.pddl`).

Fíjate en los efectos de `light`. En el Capítulo 10 dimos una regla para colocar los efectos: borrar al inicio lo que la acción consume y añadir al final lo que produce. `light` la sigue para la cerilla (deja de estar `unused` al inicio) y la incumple, a propósito, para la luz: `(lit ?m)` se añade al inicio y se borra al final. El hecho es cierto exactamente mientras se ejecuta la acción. No es un producto que quede cuando la acción termina; es algo que la acción proporciona mientras dura.

`mend` necesita ese hecho al inicio y durante toda su duración. Así que una reparación solo puede ocurrir dentro de un `light`: después de su inicio y antes de su final.

```
$ kairos cellar-domain.pddl cellar-p01.pddl
; cost = 5
; makespan = 5.000
0.000: (light m1) [5.000]
0.001: (mend f1 m1) [2.000]
```

La reparación empieza ϵ después de encender la cerilla (su inicio necesita un hecho que añade el inicio de `light`, así que los dos acontecimientos interfieren) y termina en 2,001, bien dentro de los 5 segundos. El *makespan* es la vida de la cerilla: un plan termina cuando termina su última acción, aunque el objetivo se haya cumplido antes.

Ahora intenta ejecutar estas dos acciones una después de otra, en cualquier orden. Primero la reparación: está oscuro, la condición falla. Primero la cerilla y la reparación cuando se haya consumido: la cerilla se ha apagado y vuelve a estar oscuro. Ninguna secuencia de acciones completas resuelve el problema. Un problema como este tiene **concurrencia requerida**: todo plan que lo resuelve contiene acciones que se solapan. La Figura 11.1 muestra las dos secuencias fallidas y el plan.

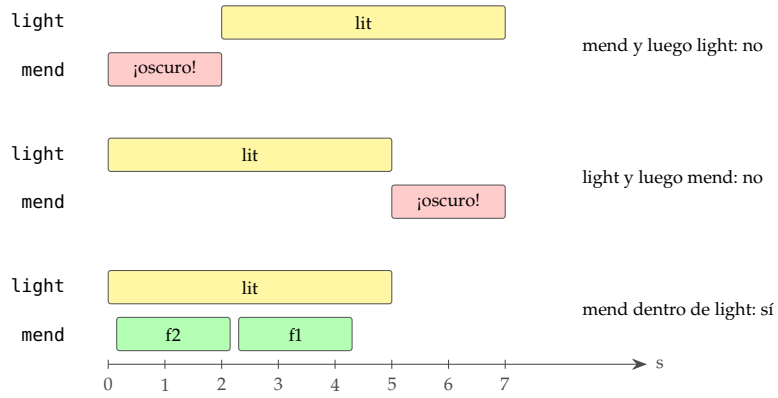


Figura 11.1.: Las dos formas de poner `light` y `mend` en secuencia fallan: la reparación queda a oscuras. Los únicos planes ponen la reparación dentro de la luz. El cronograma de abajo es el plan de `cellar-p02`, con dos fusibles reparados a la luz de una sola cerilla.

¿A cuántos fusibles puede servir una cerilla? Dos reparaciones llevan 4 segundos, que caben en 5:

```
$ kairos cellar-domain.pddl cellar-p02.pddl
; cost = 5
; makespan = 5.000
0.000: (light m1) [5.000]
0.001: (mend f2 m1) [2.000]
2.002: (mend f1 m1) [2.000]
```

Las dos reparaciones no se solapan entre sí: `(hand-free)` es un testigo, como lo era `idle` para el robot. Tres reparaciones necesitan 6 segundos, y una cerilla no basta:

```
$ kairos cellar-domain.pddl cellar-p03.pddl
note: engine M3 found no plan; as POPF, it keeps one partial plan
     per state, so this is strong evidence but not a proof [
     orchestrator.m3_exhausted]
note: no plan: the engines that apply ran out of states (144 states)
      before the time limit, so more time does not help; none of
     them is complete for this task, so a plan may still exist (see
     the note before). [orchestrator.why]
kairos: no plan found (exhausted)
$ echo $?
14
```

La respuesta merece una lectura atenta. Kairos no dice `unsolvable` (código de salida 10), como hacía con los problemas sin plan de capítulos anteriores. Dice `no plan found` (código de salida 14): ha buscado en todo lo que sabe buscar y no ha encontrado nada, pero el motor que gestiona las acciones solapadas no demuestra que no exista nada. Aquí sabemos por aritmética que no hay plan. En general, con concurrencia requerida, «el planificador no ha encontrado ningún plan» es un indicio más débil que antes, y la prueba sin plan del paso E debería ser una cuya respuesta puedas justificar a mano. Con una segunda cerilla (`cellar-p04.pddl`) los tres fusibles quedan reparados en 6,003 segundos, con la segunda cerilla encendida mientras la primera aún arde.

Un problema tiene concurrencia requerida cuando todo plan necesita acciones que se solapen. La cerilla da luz solo mientras arde; la reparación tiene que ocurrir dentro de ella.

11.2. La acción envolvente: una puerta sujeta

La forma de `light` es un patrón que merece un nombre. Una **acción envolvente** es una acción durativa que añade un hecho al inicio y lo borra al final, de modo que otras acciones, su **contenido**, solo pueden

ejecutarse dentro de ella. La acción envolvente modela cualquier cosa que está disponible durante un rato y después deja de estarlo por sí sola: la luz de una cerilla, una puerta que alguien sujeta, una lámpara que se recalienta, una ventana de contacto por radio, un agarre.

El mundo del mensajero recibe su acción envolvente con la puerta con muelle de la introducción. La Figura 11.2 muestra la planta: el pasillo y el laboratorio ya no están `connected`; los une una puerta, con el vocabulario del Capítulo 5: `(joins ?d ?a ?b)` y `(open ?d)`. La puerta está cerrada salvo que un robot la sujete.

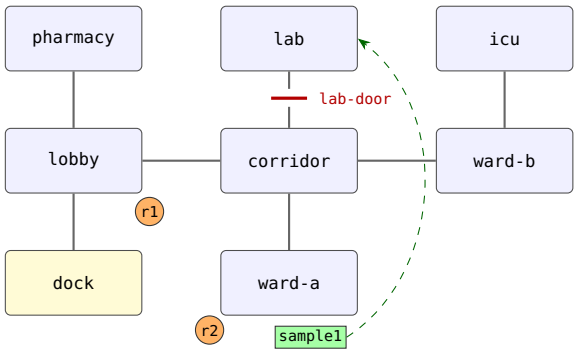


Figura 11.2.: El mundo del mensajero de este capítulo: una puerta con muelle (en rojo) entre el pasillo y el laboratorio. `r2` debe llevar la muestra al laboratorio; `r1`, en el vestíbulo, no lleva nada.

El dominio `courier-door` es el mensajero del Capítulo 10 que hace una sola cosa cada vez (con `idle`), más dos acciones. El Listado 11.2 las muestra.

```
;; The envelope: the door is open from the start of the action
;; to its end, and the robot that holds it does nothing else.
(:durative-action hold-door
  :parameters (?r - robot ?d - door ?l ?other - location)
  :duration (= ?duration 15)
  :condition (and (at start (idle ?r))
                  (at start (robot-at ?r ?l))
                  (over all (robot-at ?r ?l))
                  (at start (joins ?d ?l ?other)))
  :effect (and (at start (not (idle ?r)))
               (at start (open ?d))
               (at end (not (open ?d)))
               (at end (idle ?r))))

;; Passing needs the door open from the first step to the last.
(:durative-action pass
  :parameters (?r - robot ?d - door ?from ?to - location)
  :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
  :condition (and (at start (idle ?r))
                  (at start (robot-at ?r ?from))
                  (at start (joins ?d ?from ?to))
                  (at start (open ?d))
                  (over all (open ?d)))
  :effect (and (at start (not (idle ?r)))
               (at start (not (robot-at ?r ?from)))
               (at end (robot-at ?r ?to))
               (at end (idle ?r))))
```

`hold-door` es la acción envolvente: la puerta está abierta desde su inicio hasta su final, 15 segundos después, y el robot que la sujeta no está ocioso y se queda donde está. `pass` es un desplazamiento a través de la puerta; necesita la puerta abierta al inicio y durante todo el trayecto. Como `pass` toma el testigo `idle` del robot que pasa, y `hold-door` el del robot que sujeta, un mismo robot no puede hacer las dos cosas.

Listado 11.2: La acción envolvente `hold-door` y su contenido `pass` (de `courier-door-domain.pddl`).

Paso D: una acción que proporciona algo mientras se ejecuta

Para cada acción, pregúntate qué hace cierto *solo mientras dura*. Si existe un hecho así, la acción es envolvente: añade el hecho al inicio, bórralo al final y dale el hecho como condición al inicio y durante todo el intervalo a cada acción que dependa de él. Después pregúntate quién ejecuta la acción envolvente y quién su contenido, y si el modelo permite que sean el mismo agente. Aquí responde el testigo `idle`: quien sujeta está ocupado durante toda la acción envolvente.

`r2` está en la sala A con la muestra, a 12 metros del pasillo; `r1` está en el vestíbulo, a 10 metros; cruzar la puerta lleva 8 segundos.

```
$ kairos courier-door-domain.pddl courier-door-p01.pddl
; cost = 25.000999999999998
; makespan = 25.001
0.000: (pick r2 sample1 ward-a) [2.000]
0.000: (move r1 lobby corridor) [10.000]
2.001: (move r2 ward-a corridor) [12.000]
10.001: (hold-door r1 lab-door corridor lab) [15.000]
14.002: (pass r2 lab-door corridor lab) [8.000]
22.003: (drop r2 sample1 lab) [2.000]
```

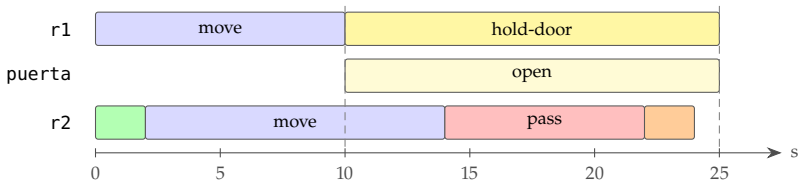


Figura 11.3.: El plan de `courier-door-p01`. La fila central no es una acción, sino el hecho `(open lab-door)`, cierto mientras `r1` sujeta la puerta. El `pass` de `r2` queda dentro de él; la sujeción empieza cuatro segundos antes de que llegue `r2`, porque nada le dice al planificador que sujete lo más tarde posible.

`r1` llega a la puerta en 10 y la sujeta de 10,001 a 25,001. `r2` llega en 14,001 y pasa de 14,002 a 22,002, dentro de la acción envolvente (Figura 11.3). El makespan es el final de la acción envolvente, 25,001, aunque la muestra está en el laboratorio en 24,003: la puerta no puede soltarse antes, porque el modelo fija la sujeción en 15 segundos. En la Sección 11.5 dejaremos que el planificador elija esa duración.

La prueba que no debe tener plan es un robot solo:

```
$ kairos courier-door-domain.pddl courier-door-p02.pddl
note: no plan: not even without durations and times (happenings in
     any order, the over all conditions kept) does the goal hold:
     196 states of that relaxation searched, with no durative action
      overlapping itself (as in POPF). [orchestrator.why]
kairos: unsolvable (exhausted)
```

Esta vez el veredicto es `unsolvable`, con demostración: Kairos se ha olvidado de las duraciones y de los tiempos, ha dejado que los inicios y los finales ocurran en cualquier orden y, aun así, un robot solo no puede estar a los dos lados de su propio testigo. Cuando ese problema más sencillo no tiene solución, el temporal tampoco la tiene.

Una acción envolvente añade un hecho al inicio y lo borra al final; su contenido necesita el hecho al inicio y durante todo el intervalo. La puerta solo está abierta mientras un robot la sujeta, y quien la sujeta está ocupado.

11.3. Temporalmente simple, temporalmente expresivo

Todos los problemas del Capítulo 10 tenían esta propiedad: si tenían algún plan, tenían un plan en el que no se solapaban dos acciones. El sótano y la puerta no la tienen. Cushing et al. (2007) precisaron la distinción y le dieron nombre. Un problema tiene concurrencia requerida si todas sus soluciones contienen acciones solapadas. Un lenguaje (un conjunto de características que un dominio puede usar) es **temporalmente expresivo** si en él pueden escribirse problemas con concurrencia requerida, y **temporalmente simple** si no.

Lo que hace a PDDL temporalmente expresivo es lo que usa la acción envolvente: un efecto al inicio de una acción que se deshace al final, y una condición de otra acción que necesita el hecho entre medias. Si se quita eso (cada acción solo consume al inicio y produce al final, como en el mensajero del Capítulo 10), cualquier plan puede estirarse hasta convertirse en una secuencia sin romperlo.

La distinción importa porque separa dos formas de construir un planificador temporal.

- **Primero la secuencia, después el horario.** Tratar cada acción durativa como una sola acción instantánea con todas sus condiciones y efectos, encontrar un plan secuencial como lo haría un planificador clásico y después calcular los instantes de inicio, dejando que se solapen las acciones que no dependen unas de otras. Es rápido, porque la búsqueda es clásica. No puede resolver el sótano: como un bloque, `light` no deja luz tras de sí, y ninguna secuencia funciona.
- **Buscar sobre inicios y finales.** Dividir cada acción durativa en sus dos acontecimientos y dejar que la búsqueda decida el orden de todos ellos, de modo que una reparación pueda empezar después de que haya empezado un encendido y antes de que termine. Así se resuelve el sótano. Se busca en un espacio mayor, porque ahora cada acción son dos pasos, y hay que llevar la cuenta de qué acciones se están ejecutando y de qué tiempos siguen siendo posibles.

Cushing y sus colegas observaron que la mayoría de los *benchmarks* temporales de la época eran temporalmente simples, y demostraron que los planificadores del primer tipo no pueden resolver problemas con concurrencia requerida, por mucho que busquen. Entre los planificadores del segundo tipo están POPF (Coles et al., 2010) y OPTIC (Benton et al., 2012).

Qué hace Kairos

Kairos tiene los dos. Su motor **M2** es del primer tipo: comprime cada acción durativa en una instantánea, busca y asigna tiempos al plan. Su motor **M3** es del segundo tipo: busca sobre los acontecimientos de inicio y de final y guarda los tiempos en una red temporal, como hace POPF. Mira la tarea para elegir, y con `-v` cuenta lo que ha visto:

```
$ kairos cellar-domain.pddl cellar-p01.pddl -v
...
[   0.001s] [orchestrator] engine M3: (light m1) sets (lit m1) only
     while it runs, and other actions can use that
```

```
[   0.001s] [orchestrator] engine M3: 2 durative actions as snap
     actions, 0 timed initial literals
...
```

«Sets a fact only while it runs, and other actions can use that» es la acción envolvente, detectada. Para el mensajero del Capítulo 10, donde ninguna acción hace eso, las mismas líneas decían `engine M2: 92 durative actions compressed`, y M3 solo se usaba después para acortar el makespan. La opción `-engine` fuerza un motor, y es una buena forma de ver la diferencia:

```
$ kairos cellar-domain.pddl cellar-p01.pddl --engine m2
warning: engine M2 forced on a task that may need actions to overlap:
      plans may be missed [orchestrator.forced_m2]
note: engine M2 found no plan, which does not prove there is none
     for this task (variable durations or possible required
     concurrency) [orchestrator.m2_incomplete]
note: no plan: the engines that apply ran out of states (2 states)
     before the time limit, so more time does not help; none of them
      is complete for this task, so a plan may still exist (see the
     note before). [orchestrator.why]
kairos: no plan found (exhausted)
```

Dos estados explorados: el inicial y el que sigue a un `light` tomado como un bloque, desde el que no se puede hacer nada útil. Esto es todo lo que consigue con el sótano un planificador del primer tipo. Los mensajes explican también el código de salida 14 de la sección anterior: ninguno de los dos motores es completo para estos problemas. M2 pierde todo plan que necesite un solapamiento; M3, como POPF, guarda una sola forma de llegar a cada estado, y puede descartar justo la que tenía futuro.

Qué significa esto para el modelador

- **Portabilidad.** La concurrencia requerida es una característica del lenguaje, aunque no tenga indicador de requisito. De los planificadores temporales de la tabla del Apéndice B, Kairos, OPTIC, POPF y TFD devuelven planes válidos para ella; LPG-td no.
- **Guía.** Un modelo temporalmente simple se explora como uno clásico, y el planificador puede decir a menudo que un problema es irresoluble. Un modelo con acciones envolventes se explora sobre inicios y finales, y «no plan found» llega sin demostración.
- **Elección.** A veces la misma situación puede modelarse con o sin acción envolvente (la Sección 11.7 compara ambas). La acción envolvente debe estar ahí porque el mundo tiene una, no por accidente.

PDDL es temporalmente expresivo: un efecto al inicio deshecho al final, que otra acción necesita, obliga a un solapamiento. Los planificadores que secuencian acciones completas no pueden resolver esos problemas; Kairos cambia a su motor M3 para ellos.

11.4. Trabajar juntos: dos robots y un cajón

Un cajón de bombonas de oxígeno pesa demasiado para un solo robot. Dos robots deben subirlo juntos a un carro. Es una coordinación de otra clase: nadie espera a nadie; dos agentes tienen que estar haciendo algo durante el mismo intervalo.

Un primer modelo, y por qué es frágil

La traducción directa de «juntos» da a cada robot su propia acción, que necesita el agarre de su compañero mientras dura:

```
  (:durative-action lift
    :parameters (?r ?partner - robot ?c - crate ?l - location)
    :duration (= ?duration 4)
    :condition (and (at start (not (= ?r ?partner)))
                    (at start (idle ?r))
                    (at start (robot-at ?r ?l))
                    (at start (crate-at ?c ?l))
                    (over all (gripping ?partner ?c)))
    :effect (and (at start (not (idle ?r)))
                 (at start (gripping ?r ?c))
                 (at end (not (gripping ?r ?c)))
                 (at end (on-cart ?c))
                 (at end (idle ?r)))))
```

Listado 11.3: Una elevación que necesita el agarre del compañero todo el tiempo (de `courier-crate-sync-domain.pddl`).

Cada `lift` es envolvente de su propio agarre y contenido del agarre del otro. Piensa en cuándo pueden empezar las dos acciones. La elevación de `r1` necesita el agarre de `r2` desde justo después de su propio inicio; la de `r2` necesita el agarre de `r1` desde justo después del suyo. Si cualquiera de los dos empieza antes, aunque sea un milisegundo, su invariante falla en el acto. Los únicos planes hacen empezar a los dos en *exactamente* el mismo instante. El validador confirma las dos mitades:

```
$ kairos validate courier-crate-sync-domain.pddl courier-crate-sync-
    p01.pddl \
   courier-crate-sync-p01-same.plan
plan valid, makespan 4
$ kairos validate courier-crate-sync-domain.pddl courier-crate-sync-
    p01.pddl \
   courier-crate-sync-p01-late.plan
courier-crate-sync-p01-late.plan:2: error: step 1 (lift r1 r2 crate1
     dock) at 0.000: over-all condition broken after the happening
    at 0.000000: (gripping r2 crate1) is false [validate.
    invariant_failed]
 2 | 0.000: (lift r1 r2 crate1 dock) [4.000]
plan invalid
```

Así que existe un plan. El planificador no lo encuentra:

```
$ kairos courier-crate-sync-domain.pddl courier-crate-sync-p01.pddl
note: engine M3 found no plan; as POPF, it keeps one partial plan
    per state, so this is strong evidence but not a proof [
    orchestrator.m3_exhausted]
...
kairos: no plan found (exhausted)
```

Una búsqueda sobre inicios y finales pone los acontecimientos en un orden, uno después de otro. Sea cual sea la elevación que empiece primero, su invariante es falso en el estado inmediatamente siguiente, y esa rama es un callejón sin salida: dos inicios nunca se toman como un solo paso. El aviso de la nota («not a proof») era honrado, y este es un caso en el que existe el plan que no podía descartar. Pero el fallo está en el modelo. Un modelo cuyos únicos planes necesitan que dos cosas ocurran exactamente en el mismo instante pide algo que los planificadores no buscan y que ninguna pareja de robots reales puede hacer. La coordinación real tiene holgura.

Un modelo con holgura

Démosle la holgura al modelo. Cada robot *agarra* el cajón durante un rato: una acción envolvente de 8 segundos. El cajón *sube*, lo que lleva 4 segundos, mientras duran los dos agarres.

```
;; Two robots lift a heavy crate onto a cart: each one grips the
;; crate for a while, and the crate rises while both grips last.
(define (domain courier-crate)
  (:requirements :strips :typing :equality
                 :negative-preconditions :durative-actions)

  (:types robot location crate)

  (:predicates
    (robot-at ?r - robot ?l - location)
    (crate-at ?c - crate ?l - location)
    (gripping ?r - robot ?c - crate)
    (on-cart ?c - crate)
    (idle ?r - robot))

  ;; An envelope: the grip lasts from the start to the end.
  (:durative-action grip
    :parameters (?r - robot ?c - crate ?l - location)
    :duration (= ?duration 8)
    :condition (and (at start (idle ?r))
                    (at start (robot-at ?r ?l))
                    (at start (crate-at ?c ?l)))
    :effect (and (at start (not (idle ?r)))
                 (at start (gripping ?r ?c))
                 (at end (not (gripping ?r ?c)))
                 (at end (idle ?r))))

  ;; The crate goes up only inside the two envelopes.
  (:durative-action raise
    :parameters (?c - crate ?r1 ?r2 - robot ?l - location)
    :duration (= ?duration 4)
    :condition (and (at start (not (= ?r1 ?r2)))
                    (at start (crate-at ?c ?l))
                    (at start (gripping ?r1 ?c))
                    (at start (gripping ?r2 ?c))
                    (over all (gripping ?r1 ?c))
                    (over all (gripping ?r2 ?c)))
    :effect (and (at start (not (crate-at ?c ?l)))
                 (at end (on-cart ?c)))))
```

```
$ kairos courier-crate-domain.pddl courier-crate-p01.pddl
; cost = 8
; makespan = 8.000
0.000: (grip r1 crate1 dock) [8.000]
0.000: (grip r2 crate1 dock) [8.000]
0.001: (raise crate1 r2 r1 dock) [4.000]
```

Listado 11.4: Dos robots levantan un cajón, con holgura (`courier-crate-domain.pddl`).

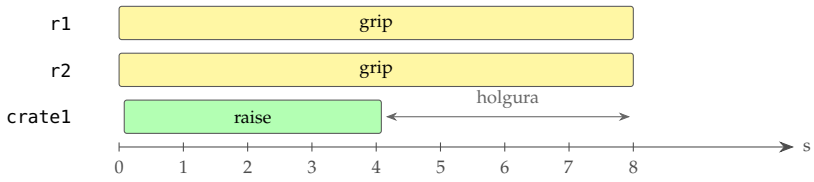


Figura 11.4.: Dos acciones envolventes y un contenido. El `raise` puede estar en cualquier punto dentro de los dos agarres; los agarres no tienen que empezar a la vez, siempre que compartan 4 segundos.

Aquí los dos agarres empiezan a la vez porque nada lo impide, pero no tienen por qué hacerlo: sirven dos agarres cualesquiera que compartan algo más de 4 segundos (el Ejercicio 11.4.1 calcula lo tarde que puede llegar el segundo). `raise` no pertenece a ningún robot; es lo que le ocurre

al cajón, como la cocción era lo que le ocurría a la sopa en la cocina del Capítulo 10.

Dos pruebas completan el paso E. Un robot solo no debe tener plan, y no lo tiene (`courier-crate-p02.pddl` es `unsolvable`). Y la prueba de la trampa: ¿puede un robot hacer los dos papeles? La condición `(at start (not (= ?r1 ?r2)))` está para eso, con el `:equality` del Capítulo 4. En una acción durativa, una igualdad entre parámetros también necesita especificador temporal; `at start` es el natural, porque la igualdad no cambia nunca. Sin esa línea (`courier-crate-alone-domain.pddl`):

```
$ kairos courier-crate-alone-domain.pddl courier-crate-p02.pddl
; cost = 8
; makespan = 8.000
0.000: (grip r1 crate1 dock) [8.000]
0.001: (raise crate1 r1 r1 dock) [4.000]
```

«Juntos» modelado como invariantes mutuos obliga a un inicio simultáneo exacto, que los planificadores no encuentran y los robots no pueden hacer. Modela la holgura: una acción envolvente por agente y el cambio conjunto dentro de todas.

11.5. Duraciones que elige el planificador

Hasta ahora, una duración era un número que calcula el modelo: una constante o una expresión como la distancia entre la velocidad. Para muchas acciones eso está mal, porque cuánto duran es una decisión. Cuánto tiempo sujetar una puerta, cuánto cargar, cuánto mantener la bomba en marcha, cuánto esperar. PDDL 2.1 permite que el `:duration` de una acción sea un conjunto de cotas sobre `?duration` en lugar de una igualdad, y deja el valor al planificador. Necesita el requisito `:duration-inequalities`.

El mensajero recibe una batería, como en el Capítulo 8: `(battery ?r)` va de 0 a `(battery-capacity ?r)`, un desplazamiento gasta `(consumption ?r)` unidades de batería por metro y la base de carga cumple `(has-charger dock)`. Lo nuevo aquí es `(charge-rate ?r)`, las unidades de batería que se ganan por cada segundo de carga. Para que el listado sea corto, este mensajero solo se desplaza y carga.

```
;; The whole consumption of the move is taken at its start.
(:durative-action move
  :parameters (?r - robot ?from ?to - location)
  :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
  :condition (and (at start (idle ?r))
                  (at start (robot-at ?r ?from))
                  (at start (connected ?from ?to))
                  (at start (>= (battery ?r)
                                (* (distance ?from ?to)
                                   (consumption ?r)))))
  :effect (and (at start (not (idle ?r)))
               (at start (not (robot-at ?r ?from)))
               (at start (decrease (battery ?r)
                                   (* (distance ?from ?to)
                                      (consumption ?r))))
               (at end (robot-at ?r ?to))
               (at end (idle ?r))))

;; From 1 second to the time that fills the battery.
(:durative-action charge
  :parameters (?r - robot ?l - location)
  :duration (and (>= ?duration 1)
                 (<= ?duration (/ (- (battery-capacity ?r)
                                     (battery ?r))
                                  (charge-rate ?r))))
  :condition (and (at start (idle ?r))
                  (at start (robot-at ?r ?l))
                  (at start (has-charger ?l))
```

```
                    (over all (robot-at ?r ?l)))
    :effect (and (at start (not (idle ?r)))
                 (at end (increase (battery ?r)
                           (* ?duration (charge-rate ?r))))
                 (at end (idle ?r)))))
```

Listado 11.5: Una carga de duración elegida (de `courier-charge-domain.pddl`).

El `:duration` de `charge` es una conjunción de dos cotas. La inferior, 1 segundo, dice que no merece la pena enchufarse para una carga más corta. La superior se calcula a partir del estado al inicio de la acción: la batería que falta, dividida entre el ritmo de carga, es el tiempo que la llena. Un robot con 20 de 100 que gana 2 por segundo puede cargar como mucho 40 segundos. Cualquiera de las dos cotas puede omitirse: `(<= ?duration 60)` sola significa «cualquier duración de hasta un minuto».[2]

El desplazamiento toma al inicio toda la batería que va a gastar, y necesita tenerla en ese momento. Es la opción prudente: si el consumo se restara al final, otra acción podría gastar, durante el desplazamiento, la carga con la que el desplazamiento contaba.

2: PDDL no fija una duración mínima, pero un plan con una acción de duración cero tiene poco sentido, y la separación ϵ se aplica entre sus dos extremos. Da una cota inferior siempre que el mundo tenga una.

`?duration` en los efectos

El efecto de `charge` es donde la elección cuenta:

```
                (at end (increase (battery ?r)
                          (* ?duration (charge-rate ?r))))
```

`?duration` puede usarse en los efectos numéricos de la acción, y representa el valor que se ha elegido. Una carga más larga da más batería. Sin esto, la elección no serviría de nada: nada dependería de ella, y el planificador tomaría siempre la más corta. Toda la ganancia llega al final de la acción, de una vez; una batería que se llena poco a poco mientras la acción se ejecuta es un efecto continuo, cosa del Capítulo 12.

Kairos dice qué ha encontrado en los ficheros:

```
$ kairos check courier-charge-domain.pddl courier-charge-p01.pddl
valid, plannable
  typing: full
  numeric-fluents: full
  durative-actions: full (required concurrency included, in engine
     M3)
  duration-inequalities: full (the bounds in engines M2 and M3,
     durations in between in steps of time)
  variable-duration: full
  duration-in-effects: full
  metric: full
  metric-total-time: full (alone or added to numeric fluents)
```

En `courier-charge-p01` el robot está en la base de carga con 20 de batería y debe llegar a la UCI: 31 metros, a 1 unidad por metro. Le faltan 11, es decir, al menos 5,5 segundos de carga.

```
$ kairos courier-charge-domain.pddl courier-charge-p01.pddl
; cost = 71.00399999999999
; makespan = 71.004
0.000: (charge r1 dock) [40.000]
40.001: (move r1 dock lobby) [4.000]
44.002: (move r1 lobby corridor) [10.000]
54.003: (move r1 corridor ward-b) [12.000]
66.004: (move r1 ward-b icu) [5.000]
```

```
; better plan 2
; cost = 37.00899999999999
; makespan = 37.009
0.000: (charge r1 dock) [1.000]
1.001: (charge r1 dock) [1.000]
2.002: (charge r1 dock) [1.000]
3.003: (charge r1 dock) [1.000]
4.004: (charge r1 dock) [1.000]
5.005: (charge r1 dock) [1.000]
6.006: (move r1 dock lobby) [4.000]
10.007: (move r1 lobby corridor) [10.000]
20.008: (move r1 corridor ward-b) [12.000]
32.009: (move r1 ward-b icu) [5.000]
```

Dos planes, y los dos dicen algo de cómo elige el planificador. En el primero, la carga dura 40 segundos, su cota superior: una batería llena. En el segundo hay seis cargas de 1 segundo, la cota inferior, una tras otra: 12 unidades, una más de las necesarias, y un makespan de 37,009.

Ninguno es el plan que escribiría una persona, una sola carga de cinco segundos y medio. Ese plan es válido, y mejor:

```
$ kairos validate courier-charge-domain.pddl courier-charge-p01.pddl
      \
    courier-charge-p01-hand.plan
plan valid, makespan 36.504, metric 36.504
```

Cómo elige un planificador una duración

Elegir un número real no es la misma clase de decisión que elegir una acción. Entre 1 y 40 hay infinitos valores, y una búsqueda no puede probarlos todos. Cada planificador sale del paso a su manera. Lo que hace Kairos es esto:

- Sus motores M2 y M3 prueban cada duración en sus dos **cotas**, la más corta y la más larga. Es lo que acabamos de ver: la carga completa y, después, la carga mínima repetida.
- Si así no se encuentra ningún plan, la tarea se planifica también en **pasos de tiempo**: una acción dura entonces hasta que pasa un paso de 0,1 segundos, o hasta el instante en que terminarla hace cierta una condición, y `?duration` en sus efectos finales es el tiempo que se ha ejecutado. Así se encuentran duraciones *intermedias* entre las cotas cuando un plan necesita una.
- Haga lo que haga la búsqueda, el validador acepta cualquier duración dentro de las cotas, como aceptó nuestros 5,5, y rechaza cualquier otra: `courier-charge-p01-long.plan`, con una carga de 41 segundos, falla con `41 <= 40 is false`.

Los planes son válidos; no siempre son los más ajustados. Si la duración exacta importa, dilo de forma que el planificador pueda verlo, o acepta un plan y ajusta tú mismo la duración, con el validador como juez. El ejemplo siguiente necesita una duración intermedia, y la obtiene.

Una cafetera

La bomba de una cafetera funciona durante el tiempo que se le indique, y la taza recibe 25 mililitros por segundo. Un expreso tiene entre 40 y 50 mililitros.

```
;; A coffee machine: the pump runs for as long as the planner
;; decides, 25 ml per second, once for each cup.
(define (domain coffee)
  (:requirements :strips :typing :durative-actions
                 :duration-inequalities :numeric-fluents)

  (:types cup)

  (:predicates
    (under-spout ?c - cup)
    (empty ?c - cup)
    (pump-off))

  (:functions
    (level ?c - cup)        ; ml in the cup
    (size ?c - cup)         ; ml that fit in the cup
    (flow))                 ; ml per second

  (:durative-action pour
    :parameters (?c - cup)
    :duration (and (>= ?duration 1)
                   (<= ?duration (/ (size ?c) (flow))))
    :condition (and (at start (pump-off))
                    (at start (under-spout ?c))
                    (at start (empty ?c))
                    (over all (under-spout ?c)))
    :effect (and (at start (not (pump-off)))
                 (at start (not (empty ?c)))
                 (at end (increase (level ?c)
                                   (* ?duration (flow))))
                 (at end (pump-off)))))
```

Listado 11.6: Una cafetera (`coffee-domain.pddl`).

Una taza se llena de una vez: `(empty ?c)` se consume al inicio del vertido. La cota superior de la duración es el tiempo que llena la taza, aquí $250/25 = 10$ segundos. Ninguna de las dos cotas da un expreso: 1 segundo son 25 ml, demasiado poco, y 10 segundos son 250 ml, muchísimo más de la cuenta. El objetivo es un intervalo, como enseñó el Capítulo 8 para los objetivos numéricos: `(>= (level cup1) 40)` y `(<= (level cup1) 50)`.

```
$ kairos coffee-domain.pddl coffee-p01.pddl
; cost = 1.6
; makespan = 1.600
0.000: (pour cup1) [1.600]
```

El vertido dura 1,6 segundos: el instante en que terminarlo hace cierto el objetivo (40 ml). Con `-v`, la línea `[orchestrator] also in steps of 0.1 s` muestra de dónde ha salido el plan. Cualquier duración de 1,6 a 2 sería válida; el planificador devuelve la primera que funciona.

Con :duration-inequalities la duración está acotada, no fijada, y la elige el planificador. Usada en un efecto, ?duration hace que la elección importe. Kairos prueba primero las cotas, y duraciones intermedias cuando las cotas fallan.

11.6. Cómo ordenan el tiempo los planificadores

Un planificador que busca sobre inicios y finales decide un *orden* de acontecimientos: este inicio, luego aquel inicio, luego este final. Todavía tiene que encontrarles *tiempos* tales que se respete cada duración y se

mantenga cada separación, o descubrir que esos tiempos no existen (tres reparaciones dentro de una cerilla). La herramienta para ello es la misma en la mayoría de los planificadores temporales, y merece la pena conocerla porque explica qué les resulta fácil y qué no.

Redes temporales simples

Una **red temporal simple** (STN) tiene una variable por cada punto temporal (aquí, cada inicio y cada final) y restricciones de una única forma: la diferencia entre dos puntos temporales está entre dos cotas, $l \leq t_j - t_i \leq u$ (Dechter et al., 1991). Todo lo que un plan temporal exige de sus tiempos tiene esa forma:

- una acción de duración 5: $t_{fin} - t_{inicio} = 5$ (las dos cotas valen 5);
- una duración que elegir entre 1 y 40: $1 \leq t_{fin} - t_{inicio} \leq 40$;
- un acontecimiento que debe ir después de otro: $t_j - t_i \geq \epsilon$.

Una STN se dibuja como un grafo: un nodo por cada punto temporal y una flecha de i a j etiquetada con $[l, u]$. Si existen tiempos que satisfacen todas las restricciones, y cuáles son los más tempranos, puede calcularse con un algoritmo de caminos mínimos: la red es consistente exactamente cuando ningún ciclo de restricciones suma una contradicción. No hace falta búsqueda, y por eso los planificadores pueden permitirse comprobar la red cada vez que el plan crece.

La Figura 11.5 es la red del plan de `cellar-p01`. La búsqueda eligió el orden: inicio de `light`, inicio de `mend`, final de `mend`, final de `light`. La red lo convierte en tiempos.

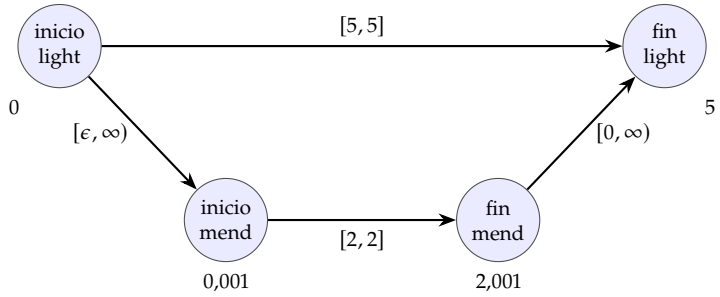


Figura 11.5.: La red temporal simple del plan de `cellar-p01`. Cada flecha acota el tiempo que va de su origen a su destino. Bajo cada nodo, el tiempo más temprano que las satisface todas: los tiempos del plan. Si la reparación durara 5 segundos, el camino que pasa por la reparación necesitaría $\epsilon + 5$ donde la flecha directa permite exactamente 5: una contradicción, y ningún plan.

Las cuatro flechas son las cuatro cosas que el plan debe respetar: las dos duraciones; que la reparación empiece después de encenderse la luz, al menos ϵ más tarde porque su inicio necesita `(lit m1)`; y que la reparación termine no más tarde que la luz (el invariante cubre el intervalo abierto, así que los dos finales pueden coincidir). Los tiempos más tempranos que las satisfacen son 0; 0,001; 2,001 y 5, exactamente el plan impreso en la Sección 11.1.

Esta imagen explica varias cosas que hemos visto. Los planes empiezan las acciones tan pronto como la red lo permite, y por eso `r1` empezó a sujetar la puerta cuatro segundos antes de que hiciera falta. Una duración elegida es una flecha con dos cotas distintas, que la red maneja sin esfuerzo; lo difícil es que los *efectos* dependan del valor elegido (la batería ganada), y los efectos quedan fuera de la red. Y un inicio simultáneo exacto, como el de la elevación frágil, no es un problema de tiempos (la red puede decir $t_j - t_i = 0$), sino de la búsqueda, que tiene que dar con él.

La búsqueda ordena inicios y finales; una red temporal simple convierte ese orden en tiempos, o muestra que no existen. Las duraciones, elegidas o fijas, y las separaciones son todas cotas sobre diferencias.

11.7. El método MODEL: coordinación

Lo que este capítulo añade al método

Paso D (tabla de acciones). Dos preguntas para cada acción, después de las del Capítulo 10:

- *¿Debe solaparse con otra acción?* Porque proporciona algo solo mientras se ejecuta (una acción envolvente), porque necesita algo así (un contenido) o porque dos agentes deben actuar durante el mismo intervalo. Anota qué acción contiene a cuál y cuánta holgura hay.
- *¿Es su duración una decisión?* Si alguien pudiera preguntar con razón «¿durante cuánto tiempo?», da cotas en lugar de un valor, y asegúrate de que algún efecto o alguna condición dependa de `?duration`; si no, la elección no significa nada.

Paso E (batería de pruebas). Para un modelo con acciones envolventes: un contenido que no cabe (tres fusibles, una cerilla) no debe tener plan; el agente de la acción envolvente no debe poder ser su propio contenido (un robot en la puerta); un mismo objeto no debe poder hacer dos papeles (el cajón). Y recuerda que aquí «no plan found» no es una demostración: prefiere pruebas cuya respuesta puedas calcular. Para una duración elegida: escribe a mano el plan con la duración que tú elegirías y valídalo.

Alternativas comparadas

Acción envolvente o predicado de cerrojo

La acción envolvente no es la única forma de decir «la puerta está abierta mientras un robot la sujeta». El patrón del Capítulo 7 para un recurso que se toma y se libera eran dos acciones y un predicado: un **cerrojo**. En el tiempo queda así.

```
;; The lock: the robot opens the door and keeps holding it
;; (it is not idle) until it releases it with another action.
(:durative-action open-door
  :parameters (?r - robot ?d - door ?l ?other - location)
  :duration (= ?duration 1)
  :condition (and (at start (idle ?r))
                  (at start (robot-at ?r ?l))
                  (at start (joins ?d ?l ?other)))
  :effect (and (at start (not (idle ?r)))
               (at end (holding-door ?r ?d))
               (at end (open ?d))))

(:durative-action release-door
  :parameters (?r - robot ?d - door)
  :duration (= ?duration 1)
  :condition (at start (holding-door ?r ?d))
  :effect (and (at start (not (holding-door ?r ?d)))
               (at start (not (open ?d)))
               (at end (idle ?r))))
```

Listado 11.7: La puerta con muelle con un predicado de cerrojo (de `courier-door-lock-domain.pddl`).

`open-door` toma el testigo del robot y no lo devuelve: el robot sigue ocupado, sujetando. `release-door` cierra la puerta y libera al robot. Entre las dos, la puerta está abierta todo el tiempo que el plan quiera. Sobre la entrega de `courier-door-p01`:

```
$ kairos courier-door-lock-domain.pddl courier-door-p01.pddl
; cost = 24.003
; makespan = 24.003
0.000: (move r1 lobby corridor) [10.000]
0.000: (pick r2 sample1 ward-a) [2.000]
2.001: (move r2 ward-a corridor) [12.000]
10.001: (open-door r1 lab-door corridor lab) [1.000]
14.002: (pass r2 lab-door corridor lab) [8.000]
22.003: (drop r2 sample1 lab) [2.000]
```

El plan es un segundo más corto que con la acción envolvente, y tiene un agujero: nadie suelta la puerta. El objetivo no lo pedía, así que el plan termina con `r1` sujetando para siempre la puerta del laboratorio. Una acción envolvente no puede quedarse abierta; un cerrojo sí, salvo que el objetivo lo impida. `courier-door-p04` es la misma entrega con `(idle r1)` y `(idle r2)` añadidos al objetivo, y entonces el plan tiene su `release-door`, en 22,002, el instante en que `r2` ha pasado. La Tabla 11.1 compara las dos codificaciones sobre ese problema.

Tabla 11.1.: Dos codificaciones de una puerta que está abierta mientras un robot la sujeta, ejecutadas sobre `courier-door-p04`.

	Acción envolvente (`hold-door`)	Cerrojo (`open-door`, `release-door`)
Fidelidad	La puerta siempre se cierra, exactamente a los 15 s. Demasiado rígida si la sujeción pudiera ser más corta (makespan 25,001)	La puerta está abierta lo que haga falta (24,003), sin límite superior, y se queda abierta para siempre salvo que el objetivo diga otra cosa
Tamaño	64 acciones instanciadas, 31 hechos	66 acciones instanciadas, 33 hechos
Guía	Concurrencia requerida: se explora sobre inicios y finales (1919 estados); «no plan found» puede llegar sin demostración	Temporalmente simple: se explora como una secuencia y se le asignan tiempos (primer plan del motor M2)
Claridad	Una sola acción lo dice todo: quién sujeta y durante cuánto	Dos acciones y un predicado que deben ir acompasados; el objetivo tiene que mencionar la puerta o los robots
Portabilidad	Solo planificadores con concurrencia requerida	Cualquier planificador temporal

Veredicto. Usa una acción envolvente cuando la cosa termina por sí sola o tiene una duración máxima que el agente no controla: una cerilla se consume, una lámpara se recalienta, el paso de un satélite termina. Ahí el cerrojo está mal, porque deja que el plan mantenga la cerilla encendida (Ejercicio 11.7.2). Usa un cerrojo cuando el agente decide cuándo parar y no hay límite: una puerta que simplemente se abre y se cierra, una herramienta que se coge y se deja. Añade entonces al objetivo, o a cada problema, que las cosas queden como deben. Y cuando el agente decide *y* hay un límite, lo mejor de ambos es una acción envolvente con duración elegida, `(<= ?duration 15)`.

Una duración fija o una elegida

Para la carga hay tres codificaciones, y son tres ficheros con el mismo nombre de dominio, para que el mismo problema se ejecute sobre todos ellos. *Completa*: la carga llena siempre la batería, con una duración calculada a partir del estado (la batería que falta dividida entre el ritmo de carga, escrita con = donde el Listado 11.5 tiene <=) y un efecto que asigna la capacidad. *Bloques*: cada carga dura 5 segundos y suma 5 veces el ritmo de carga, y el planificador elige cuántas hace. *Elegida*: la desigualdad del Listado 11.5. La Figura 11.6 dibuja los planes del viaje a la UCI.

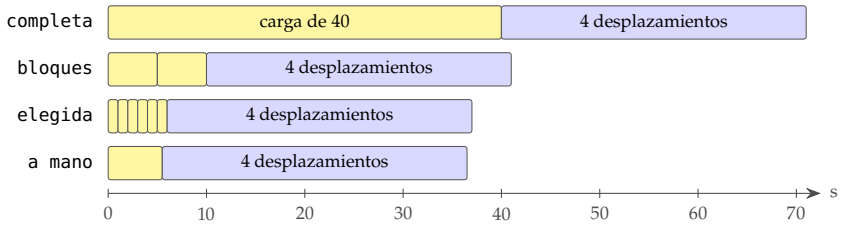


Figura 11.6.: El viaje de `courier-charge-p01` con las tres codificaciones de la carga, tal como las planifica Kairos, y el plan escrito a mano para la tercera. El amarillo es carga. Los cuatro desplazamientos llevan 31 segundos en todos; solo cambia la carga.

Tabla 11.2.: Tres codificaciones de una carga, ejecutadas sobre `courier-charge-p01`.

	Carga completa	Bloques de 5 s	Duración elegida
Fidelidad	Pierde soluciones: el robot no puede salir antes. Makespan 71,004	Aproximada: cargas de 5, 10, 15... s. Makespan 41,005	Cada carga desde 1 s hasta llenar es un plan. Kairos devuelve 37,009; el mejor es 36,504
Tamaño	15 acciones instanciadas	15 acciones instanciadas	15 acciones instanciadas
Guía	Una sola carga lo decide todo	Más pasos; el tamaño del bloque cambia precisión por longitud del plan	El planificador debe elegir un número; cada motor lo hace a su manera, y los planes pueden quedar lejos del mejor
Claridad	Sencilla	Sencilla, con una constante arbitraria (¿por qué 5?)	Dice lo que se quiere decir
Portabilidad	Cualquier planificador temporal que calcule duraciones a partir del estado	Cualquier planificador temporal con números	Kairos, OPTIC, POPF; no TFD ni LPG-td

Veredicto. Si la acción tiene de verdad una duración natural (llenar el depósito, hornear el pan), fíjala. Si la duración es una decisión y el planificador que vas a usar lo admite, acótala: el modelo dice lo que se quiere decir, y todo plan razonable es un plan del modelo. Los bloques son la solución de compromiso portable, y una perfectamente buena cuando la precisión de un bloque basta; elige el bloque con el mismo cuidado que cualquier otro número del modelo.

Una acción envolvente no puede quedarse abierta y tiene una duración; un cerrojo es flexible y portable, pero el objetivo debe cerrarlo. Una duración elegida dice lo que se quiere decir; los bloques son su aproximación portable.

11.8. Errores comunes

Una acción envolvente por accidente

Síntoma: un modelo que se resolvía al instante deja de resolverse, o Kairos con `-v` informa de `engine M3: (...) sets (...) only while it runs, and other actions can use that` para una acción que no veías como envolvente. **Causa:** un hecho añadido al inicio y borrado al final que otra acción lee; a menudo, un indicador de «ocupado» escrito al revés (`(working ?r)` añadido al inicio y borrado al final, y usado como condición en algún sitio). **Solución:** si nada debe ocurrir dentro de la acción por causa de ese hecho, usa un testigo que se borre al inicio y se añada al final, como `idle`. Reserva las acciones envolventes para lo que el mundo proporciona solo durante un rato.

Tomar «no plan found» por «unsolvable»

Con concurrencia requerida son dos respuestas distintas, con códigos de salida distintos (14 y 10). La primera llega con una nota que lo dice:

```
note: engine M3 found no plan; as POPF, it keeps one partial
     plan per state, so this is strong evidence but not a proof
     [orchestrator.m3_exhausted]
```

La elevación frágil de la Sección 11.4 tenía un plan y recibió esta respuesta. Antes de concluir que un modelo no tiene plan, escribe en un fichero el plan que esperabas y valídalo. Si es válido, el modelo está bien y la búsqueda no ha podido encontrarlo: busca qué lo hace difícil (normalmente, la necesidad de una sincronización exacta) y dale holgura al modelo.

El agente de la acción envolvente es también su contenido

Si `hold-door` no tomara `idle`, un robot sujetaría la puerta y la cruzaría; si `raise` no pidiera dos robots distintos, uno levantaría el cajón solo (el plan con `(raise crate1 r1 r1 dock)` de la Sección 11.4). Los parámetros del mismo tipo siempre pueden ligarse al mismo objeto. Siempre que un modelo diga «otro», tiene que haber un testigo o una desigualdad detrás de la palabra.

Una duración elegida de la que nada depende

Una acción con `(<= ?duration 60)` cuyos efectos no mencionan `?duration` da al planificador una elección sin consecuencias, salvo en el reloj. Como acción envolvente tiene sentido (sujetar la puerta el tiempo que haga falta). Para una carga, deja que el planificador llene la batería en el tiempo más corto permitido. Comprueba que la magnitud para la que existe la acción crece con `?duration`, y que la cota superior la detiene donde lo haría el mundo: el validador hace cumplir la cota,

```
courier-charge-p01-long.plan:2: error: step 1 (charge r1 dock)
     at 0.000: duration 41.000000 does not satisfy (and (>= ?
     duration 1) (<= ?duration (/ (- (battery-capacity ?r) (
     battery ?r)) (charge-rate ?r)))): (<= ?duration (/ (- (
     battery-capacity ?r) (battery ?r)) (charge-rate ?r))): 41
     <= 40 is false [validate.duration_mismatch]
```

pero solo si el modelo la expresa. Sin la cota superior, la batería pasaría de su capacidad.

Esperar la mejor duración

El primer plan del viaje a la UCI cargaba 40 segundos cuando bastaban 5,5, y el mejor que devolvió usaba seis cargas de un segundo. Las dos son respuestas correctas al modelo. Un planificador que prueba los extremos de un intervalo dará planes en los extremos. Cuando importa una duración intermedia, haz que el modelo la necesite (un objetivo o una condición que las cotas no satisfagan, como el intervalo del expreso), o usa bloques, o ajusta el plan y valídalo.

11.9. Resumen

Algunos problemas solo pueden resolverse solapando acciones, y algunas duraciones son decisiones. Lo primero viene de un hecho que una acción proporciona solo mientras se ejecuta; lo segundo, de un `:duration` escrito como cotas.

Tabla 11.3.: Lo que ha añadido este capítulo.

Concepto	Qué recordar
Concurrencia requerida	todo plan tiene acciones solapadas; ninguna secuencia de acciones completas funciona
Acción envolvente	añade un hecho al inicio y lo borra al final; su contenido necesita el hecho al inicio y durante todo el intervalo
Temporalmente expresivo	un lenguaje que puede expresar concurrencia requerida; PDDL 2.1 lo es. Los planificadores que secuencian acciones completas son incompletos para él
Motores de Kairos	M2 comprime y asigna tiempos (tareas temporalmente simples); M3 busca sobre inicios y finales (acciones envolventes). `-v` dice cuál, `-engine` fuerza uno
«No plan found» (salida 14)	la búsqueda terminó sin plan y sin demostración; valida el plan que esperabas
Trabajo conjunto	una acción envolvente por agente y el cambio conjunto dentro de todas; nunca un inicio simultáneo exacto
`:duration-inequalities`	`:duration (and (>= ?duration a) (<= ?duration b))`; elige el planificador
`?duration` en los efectos	`(increase (f) (* ?duration rate))` al final: la elección tiene una consecuencia
STN	puntos temporales y cotas sobre sus diferencias; convierte un orden de acontecimientos en tiempos
Predicado de cerrojo	dos acciones y un hecho: flexible y temporalmente simple; el objetivo debe cerrarlo

En todo esto, el mundo solo cambiaba en los dos extremos de una acción: la batería saltaba 80 unidades en el último instante de una carga de 40 segundos. El Capítulo 12 deja que las magnitudes cambien *mientras* la acción se ejecuta, de modo que un robot pueda detenerse a mitad de una carga con la mitad de la ganancia, y una condición pueda vigilar un valor mientras se mueve.

11.10. Lecturas complementarias

La concurrencia requerida, y la clasificación de los lenguajes temporales en simples y expresivos, vienen de Cushing et al. (2007); el artículo es breve y sus ejemplos se parecen a los de este capítulo. Las desigualdades de duración y el uso de `?duration` en los efectos se definen con el resto de PDDL 2.1 en Fox y Long (2003). Las redes temporales simples las introdujeron Dechter et al. (1991), que estudian también las redes más difíciles en las que una restricción puede ser una elección entre varios intervalos. Cómo usa una red así una búsqueda hacia delante sobre inicios y finales es el tema del artículo de POPF (Coles et al., 2010), y OPTIC (Benton et al., 2012) se construye sobre él. Haslum et al. (2019) explican la concurrencia requerida y la compilación de las acciones durativas en acciones de inicio y de final en su capítulo sobre planificación temporal, y Ghallab et al. (2016) tratan las redes temporales y su controlabilidad, la cuestión de qué ocurre cuando algunas duraciones las decide el mundo y no el agente. Los motores de Kairos y cómo elige entre ellos se describen en Martín Rico et al. (2026).

11.11. Ejercicios

Ejercicios de la Sección 11.1 — Planes que solo existen si las acciones se solapan

Ejercicio 11.1.1. En el sótano, una cerilla arde 5 segundos y una reparación lleva 2. Supón que la reparación llevara 5 segundos, exactamente lo mismo que la cerilla. ¿Hay un plan para un fusible? Razónalo a partir de las condiciones de `mend`; después cambia la duración en una copia del dominio y ejecuta `cellar-p01`. ¿Qué código de salida obtienes, y por qué no es 10?

Ejercicio 11.1.2. En `cellar-p01`, la cerilla se enciende en 0. Predice si cada uno de estos planes es válido y compruébalo después con `kairos validate`: (a) la reparación empieza en 0,000; (b) la reparación empieza en 3,000 y, por tanto, termina en el instante en que se apaga la cerilla; (c) la reparación empieza en 3,500.

Ejercicio 11.1.3. Una cámara fotografía las plantas de un invernadero. Una foto lleva 4 segundos y necesita una lámpara de flash encendida todo ese tiempo. Una lámpara se mantiene encendida 10 segundos y después debe enfriarse durante el resto de la sesión; la cámara hace una foto cada vez. Escribe el dominio y un problema con tres plantas y dos lámparas. ¿Cuántas lámparas enciende el plan, y cuándo?

Ejercicios de la Sección 11.2 — La acción envolvente: una puerta sujeta

Ejercicio 11.2.1. En `courier-door-p03` la muestra está dentro del laboratorio y hay que llevarla a la sala A. Entrar, cogerla y salir lleva 18 segundos, más que una sujeción. Ejecútalo y dibuja el cronograma del plan. ¿Dónde está `r1` al final? Escribe a mano un plan más corto en el que `r1` no entre nunca en el laboratorio, y valídalo.

Ejercicio 11.2.2. En `courier-door-p01`, el robot `r2` se vuelve lento: su velocidad es 0,5, así que cruzar la puerta le lleva 16 segundos, uno más que una sujeción. ¿Es irresoluble el problema? Predícelo y después ejecútalo.

Ejercicios de la Sección 11.3 — Temporalmente simple, temporalmente expresivo

Ejercicio 11.3.1. Di si cada dominio puede expresar problemas con concurrencia requerida y, si puede, qué acción es la envolvente: `courier-time` y `kitchen` del Capítulo 10, `cellar`, `courier-door`, `courier-door-lock`, `courier-crate`. Comprueba tus respuestas con las líneas `[orchestrator]` de `kairos -v`.

Ejercicio 11.3.2. Ejecuta `courier-door-p01` y `courier-door-p04` con `-engine m2`, primero con el dominio de la acción envolvente y después con el del cerrojo. Explica cada uno de los resultados.

Ejercicios de la Sección 11.4 — Trabajar juntos: dos robots y un cajón

Ejercicio 11.4.1. En `courier-crate`, un agarre dura 8 segundos y la elevación 4. Si `r1` agarra en 0, ¿cuál es el último instante en que `r2` puede empezar su agarre de modo que siga existiendo un plan

(con $\epsilon = 0{,}001$)? Escribe el plan con ese agarre y valídalo, y otro un milisegundo más tarde.

Ejercicio 11.4.2. Un cajón más pesado necesita tres robots. Cambia `raise` para que tengan que estar agarrando tres robots distintos. ¿Cuántas desigualdades necesita? Pruébalo con tres robots, y di cuántas instancias de `raise` hay para tres robots, un cajón y un lugar.

Ejercicios de la Sección 11.5 — Duraciones que elige el planificador

Ejercicio 11.5.1. En `courier-charge-p01` cambia el objetivo: el robot debe ir a la sala A y estar de vuelta en la base de carga al final. Con lápiz y papel: ¿cuánta batería necesita el viaje de ida y vuelta, cuál es la carga única más corta antes de salir que lo permite y cuál es el makespan de ese plan (sin las separaciones)? ¿Serviría de algo volver a cargar por el camino?

Ejercicio 11.5.2. Amplía la cafetera con una segunda taza y una acción que cambia la taza que está bajo la boquilla, que lleva 3 segundos y no debe ocurrir mientras la bomba funciona. Pide un expreso (de 40 a 50 ml) y un café largo (de 100 a 120 ml). ¿Qué duraciones elige el planificador?

Ejercicio 11.5.3. Cada segundo de carga en la base cuesta 3 céntimos. Añade una función `(price ?l)` y haz que `charge` sume su coste a `(total-cost)`; minimízalo en `courier-charge-p01`. Lee lo que imprime Kairos con el plan. Después valida el plan a mano con una carga de 5,5 segundos y compara la métrica.

Ejercicios de la Sección 11.6 — Cómo ordenan el tiempo los planificadores

Ejercicio 11.6.1. Toma el plan de `courier-crate-p01` y dibuja su red temporal simple: seis puntos temporales y las restricciones entre ellos. ¿Qué restricciones vienen de duraciones, cuáles de condiciones al inicio y cuáles de invariantes? Calcula a mano los tiempos más tempranos y compáralos con el plan.

Ejercicio 11.6.2. En la red de la Figura 11.5, sustituye la duración de `light` por una elegida entre 1 y 5 segundos. ¿Qué flecha cambia? ¿Cuál es la cerilla más corta que todavía permite la reparación, y cuál es entonces el makespan?

Ejercicios de la Sección 11.7 — El método MODEL: coordinación

Ejercicio 11.7.1. Aplica el paso L. Para cada caso, elige entre acción envolvente y cerrojo, y entre duración fija y elegida, con una frase de justificación: (a) un puente levadizo bajo el que debe pasar un barco mientras está levantado, y que un operario sube y baja; (b) un satélite que solo puede enviar datos mientras está sobre la estación de tierra, 8 minutos por órbita; (c) un brazo robótico que sujeta una pieza mientras otro brazo la suelda; (d) un horno que debe cocer una hogaza exactamente 35 minutos.

Ejercicio 11.7.2. Reescribe el sótano de las cerillas con un cerrojo: `strike` enciende una cerilla y `blow-out` la apaga. Ejecuta el problema con tres fusibles y una cerilla (`cellar-p03`). ¿Qué responde ahora el planificador, y qué ha perdido el modelo?

12. Cambio continuo

12.1 Una escalera donde el mundo tiene una rampa 270
12.2 Efectos continuos 272
12.3 Leer un número mientras cambia 274
Agua que viene y va: el invernadero 277
12.4 El generador: cuándo empezar 279
12.5 El horno: cuándo parar 282
12.6 Lo que debe encontrar el planificador 284
12.7 El mundo del mensajero: una batería que se descarga y se carga . . . 286
Ensayar el modelo . . . 287
Dónde cargar, y cuánto tiempo 289
12.8 El método MODEL: cambio continuo 290
Alternativas comparadas 291
12.9 Errores comunes 292
12.10 Resumen 293
12.11 Lecturas complementarias 294
12.12 Ejercicios 294

Las acciones durativas del Capítulo 10 y del Capítulo 11 llevan tiempo, pero sus números no: un número cambia al principio de una acción o al final, de un salto, y entre ambos momentos se queda quieto. Para la batería del mensajero es una simplificación que aceptamos sin mirarla de cerca. Ahora la miramos. El robot `r1` está en la base de carga. Su batería es vieja: admite 30 unidades y le quedan 10. Antes de entrar en la UCI debe encender su lámpara ultravioleta durante 20 segundos, y la lámpara gasta 2 unidades por segundo, 40 en total. Cualquiera que haya usado un móvil mientras se carga sabe qué hacer: enchufar el robot y encender la lámpara. El cargador da 5 unidades por segundo, la lámpara gasta 2 y la batería sube mientras la lámpara funciona. El modelo del Capítulo 11 no puede encontrar este plan. En él, una carga suma su energía cuando termina, así que la lámpara tendría que esperar a una batería de 40 que este robot no puede tener nunca.

Este capítulo da a los números la capacidad de cambiar *mientras* se ejecuta una acción. PDDL 2.1 lo llama efecto continuo: un efecto que indica el ritmo al que un número crece o disminuye, por segundo, mientras dura la acción. Con él, la batería es una línea que sube y baja con una pendiente, varias acciones pueden empujar el mismo número a la vez y una condición puede preguntar por el valor en cualquier instante intermedio. En este capítulo, el cambio continuo ocurre siempre dentro de una acción durativa que el agente decide ejecutar; un mundo que cambia por sí solo, actúe el agente o no, es el tema del Capítulo 17.

Empezamos con la lámpara y vemos exactamente dónde falla un modelo de saltos. Después conocemos la construcción con el ejemplo más sencillo que hay, un depósito y un grifo, y volvemos a la lámpara para resolverla. Siguen tres dominios pequeños, cada uno con una lección: un invernadero en el que el agua entra y sale del depósito a la vez; el generador, donde la pregunta es *cuándo* empezar una acción; y un horno, donde la pregunta es cuándo terminarla. Tras ver qué exige todo esto a un planificador, y cómo responde Kairos, el mundo del mensajero recibe una batería que se descarga por el camino y se carga durante el tiempo que decida el planificador. El capítulo se cierra con las preguntas que el cambio continuo añade al método MODEL y con una comparativa que dice cuándo un salto sigue siendo suficiente.

12.1. Una escalera donde el mundo tiene una rampa

Escribamos la escena de la base de carga con las herramientas del Capítulo 11. Hay dos acciones. `charge` dura lo que quiera el planificador y, cuando termina, suma a la batería el ritmo de carga multiplicado por `?duration`. `sanitize` enciende la lámpara durante 20 segundos. La lámpara no debe encontrarse la batería vacía a medio camino, y en un modelo en el que los números solo saltan hay una única forma segura de garantizarlo: pedir toda la energía antes de empezar y descontarla al principio. El Listado 12.1 muestra las dos acciones.

```
  (:durative-action charge
    :parameters (?r - robot ?l - location)
    :duration (and
      (>= ?duration 1)
      (<= ?duration (/ (- (battery-capacity ?r) (battery ?r))
                       (charge-rate ?r))))
    :condition (and
      (at start (idle ?r))
      (at start (has-charger ?l))
      (over all (robot-at ?r ?l)))
    :effect (and
      (at start (not (idle ?r)))
      (at end (increase (battery ?r)
                (* ?duration (charge-rate ?r))))
      (at end (idle ?r))))

  ; The ultraviolet lamp runs for 20 s on the battery.
  (:durative-action sanitize
    :parameters (?r - robot)
    :duration (= ?duration 20)
    :condition (and
      (at start (lamp-off ?r))
      (at start (>= (battery ?r) (* 20 (lamp-rate ?r)))))
    :effect (and
      (at start (not (lamp-off ?r)))
      (at start (decrease (battery ?r) (* 20 (lamp-rate ?r))))
      (at end (lamp-off ?r))
      (at end (sanitized ?r)))))
```

Listado 12.1: La escena de la base de carga con cambios discretos (las dos acciones de `lamp-discrete-domain.pddl`).

El problema es el de la historia: 10 unidades de batería, una capacidad de 30, un cargador de 5 por segundo y una lámpara de 2 por segundo.

```
$ kairos lamp-discrete-domain.pddl lamp-discrete-p01.pddl
note: engine M3 found no plan; as POPF, it keeps one partial plan
     per state, so this is strong evidence but not a proof [
     orchestrator.m3_exhausted]
note: no plan in steps of time; a plan that needs finer timing than
     the steps would be missed, so this is evidence but not a proof
     [orchestrator.steps_exhausted]
note: no plan: the engines that apply ran out of states (1412 states)
      before the time limit, so more time does not help; none of
     them is complete for this task, so a plan may still exist (see
     the note before). [orchestrator.why]
kairos: no plan found (exhausted)
```

No hay plan, y con razón para este modelo: `sanitize` pide 40 unidades al empezar y la batería admite 30. (Por qué la respuesta es `no plan found` y no `unsolvable` es asunto de la Sección 12.6.) El modelo es prudente, y su prudencia ha perdido una solución que existe en el mundo.

Podríamos ser menos prudentes. Supongamos que la lámpara se paga al final, como la carga, y que la acción solo comprueba al principio que la batería no está vacía. Es el fichero `lamp-careless-domain.pddl`, que se diferencia del listado en dos líneas de `sanitize`:

```
      (at start (>= (battery ?r) 0)))
      ...
      (at end (decrease (battery ?r) (* 20 (lamp-rate ?r))))
```

```
$ kairos lamp-careless-domain.pddl lamp-careless-p01.pddl
; cost = 20
; makespan = 20.000
0.000: (sanitize r1) [20.000]
```

Ahora hay un plan, y es peor que no tenerlo: el robot enciende la lámpara sin enchufarse, y el modelo no ve nada raro hasta el último instante, cuando la batería salta de 10 a −30. En ese momento nada del modelo mira la batería. En el mundo, la batería se vacía a los cinco segundos y la lámpara se apaga (Figura 12.1).

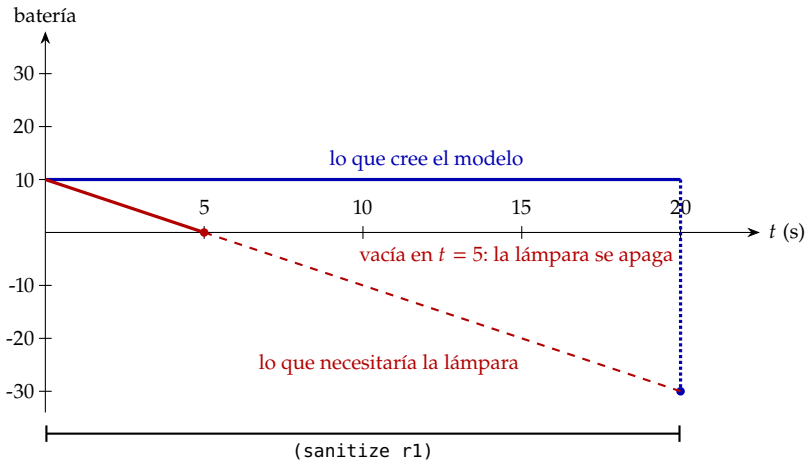


Figura 12.1.: El plan del modelo descuidado. En el modelo, la batería es una escalera con un solo peldaño, al final de la acción. En el mundo es una rampa que llega a cero a los 5 segundos. Fíjate en los 15 segundos en los que modelo y mundo no coinciden: ninguna condición del modelo puede verlos.

Los dos modelos tienen el mismo defecto, visto desde dos lados. En ellos la batería tiene un valor al principio de una acción y otro al final, y nada entre medias: un **cambio discreto**, un salto en un instante. El modelo prudente coloca los saltos donde no pueden hacer daño (la energía sale pronto y llega tarde) y por eso rechaza los planes en los que la energía se usa a medida que llega. El modelo descuidado coloca un salto donde esconde el problema y por eso acepta planes que fallan. Ninguno puede decir lo que es verdad: que la batería baja dos unidades cada segundo mientras la lámpara está encendida, y sube cinco cada segundo mientras el cargador está conectado.

Con cambios discretos, un número no tiene valor en mitad de una acción. Poner el salto pronto pierde planes; ponerlo tarde acepta planes que fallan.

Cambio discreto y cambio continuo

Un **cambio discreto** ocurre en un instante: antes, el número tiene un valor; después, otro. Todos los efectos numéricos del Capítulo 8, y los de las acciones durativas bajo `at start` y `at end`, son discretos. Un **cambio continuo** ocurre durante un intervalo: el número pasa por todos los valores entre el primero y el último. Lo que un modelo dice de un cambio continuo es su **ritmo**: cuánto cambia el número por unidad de tiempo. Si el ritmo no varía mientras se ejecuta la acción, el cambio es **lineal** y su gráfica es un segmento recto. Todos los cambios de este capítulo son lineales.

12.2. Efectos continuos

Un depósito y un grifo bastan para ver la construcción sola. El grifo deja entrar un número fijo de litros por segundo. La acción `fill` lo abre, deja correr el agua durante un tiempo que elige el planificador y lo cierra (Listado 12.2).

```
;; A water tank and its tap. While the tap is open the level
;; rises at a constant rate: a continuous effect.
(define (domain tank)
  (:requirements :typing :durative-actions :numeric-fluents
                 :duration-inequalities :continuous-effects)
```

```
6
7   (:types tank)
8
9   (:predicates
10    (tap-closed ?t - tank))
11
12  (:functions
13    (level ?t - tank)       ; liters in the tank now
14    (capacity ?t - tank)    ; liters it can hold
15    (flow ?t - tank))       ; liters per second through the tap
16
17  ; Open the tap, let the water run, close the tap. The planner
18  ; chooses for how long: at most until the tank is full.
19  (:durative-action fill
20    :parameters (?t - tank)
21    :duration (and
22      (>= ?duration 1)
23      (<= ?duration (/ (- (capacity ?t) (level ?t)) (flow ?t))))
24    :condition (at start (tap-closed ?t))
25    :effect (and
26      (at start (not (tap-closed ?t)))
27      (at end (tap-closed ?t))
28      (increase (level ?t) (* #t (flow ?t))))))
```

Listado 12.2: Un grifo llena un depósito: el nivel sube mientras se ejecuta la acción.

Todo en el fichero es conocido del Capítulo 11 salvo la línea 28:

```
      (increase (level ?t) (* #t (flow ?t)))
```

Esto es un **efecto continuo**. Tres cosas lo distinguen de los efectos que conocemos.

- No lleva alrededor ni `at start` ni `at end`. No ocurre en un instante: actúa durante toda la acción, desde el principio hasta el final.
- Su cantidad es un producto con el símbolo `#t`. El símbolo representa el tiempo transcurrido desde que empezó la acción, así que `(* #t (flow ?t))` se lee «el caudal, por cada segundo transcurrido». Tras s segundos de una acción que empezó con un nivel l_0, y si nada más toca el depósito, el nivel es $l_0 + s \cdot \mathit{flow}$. El factor que multiplica a `#t` es el ritmo.
- Solo puede ser un `increase` o un `decrease`. A un número no se le puede asignar un valor «de forma continua», y los efectos de escalado no tienen forma continua.

Un efecto continuo no lleva etiqueta de tiempo. Indica un ritmo: cuánto crece o disminuye el número por segundo mientras se ejecuta la acción.

El requisito es `:continuous-effects`. Un dominio que lo declara necesita también `:durative-actions` y `:numeric-fluents`, como cualquier dominio con acciones durativas y números. El producto se puede escribir en cualquier orden, `(* #t rate)` o `(* rate #t)`, y el ritmo puede ser cualquier expresión numérica: una constante, una función como `(flow ?t)` o una operación aritmética sobre varias.

La duración de `fill` es una desigualdad, como en el Capítulo 11: al menos un segundo y, como mucho, el tiempo que tarda en llenarse el depósito, calculado a partir del nivel al principio. El problema `tank-p01.pddl` tiene un depósito vacío de 100 litros, un grifo de 2 litros por segundo y el objetivo `(>= (level t1) 100)`. Primero la comprobación, que indica en qué medida planifica Kairos con cada característica, y después el plan:

```
$ kairos check tank-domain.pddl tank-p01.pddl
valid, plannable
  typing: full
  numeric-fluents: full
  durative-actions: full (required concurrency included, in engine
     M3)
```

```
   duration-inequalities: full (the bounds in engines M2 and M3,
      durations in between in steps of time)
   variable-duration: full
   continuous-effects: full (linear ones as discrete changes in
      engines M2 and M3, any in steps of time (integrated with Runge-
      Kutta))
$ kairos tank-domain.pddl tank-p01.pddl
; cost = 50
; makespan = 50.000
0.000: (fill t1) [50.000]
```

Una acción de 50 segundos: 2 litros por segundo durante 50 segundos son los 100 litros. El plan tiene el mismo aspecto que tendría un plan del Capítulo 11, y para este problema un efecto discreto al final, `(at end (increase (level ?t) (* ?duration (flow ?t))))`, también lo habría dado. La diferencia está en lo que el modelo sabe de esos 50 segundos. Con el efecto discreto, el depósito está vacío hasta el último instante. Con el continuo, el nivel en el segundo 10 es de 20 litros y en el segundo 30 es de 60 (Figura 12.2), y la sección siguiente le saca partido a ese conocimiento. Las dos líneas largas de la comprobación, sobre las duraciones y los efectos continuos, resumen cómo los planifica Kairos; la Sección 12.6 las explica.

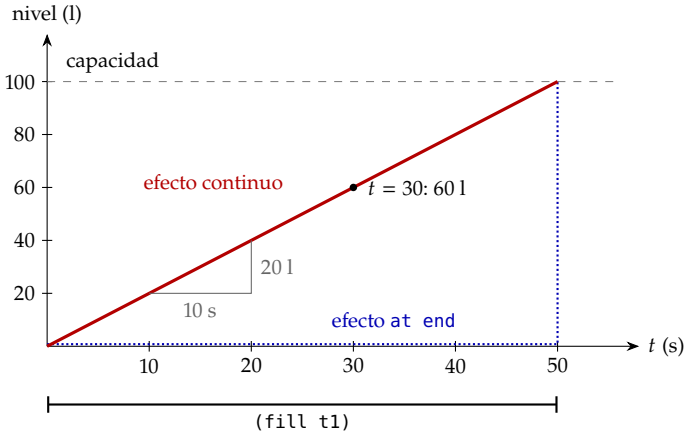


Figura 12.2.: El nivel del depósito durante el plan. La pendiente de la línea continua es el ritmo, 2 litros por segundo. La línea de puntos es lo que describiría un efecto discreto al final: los mismos valores primero y último, y nada cierto entre medias.

Los niveles de PDDL 2.1

PDDL 2.1, de Maria Fox y Derek Long Fox y Long, 2003, se definió por niveles. El nivel 1 es el lenguaje clásico de la Parte II; el nivel 2 añade números; el nivel 3, acciones durativas cuyos efectos ocurren al principio y al final; y el nivel 4, acciones durativas con efectos continuos, el tema de este capítulo. Un quinto nivel, con cambios que el mundo pone en marcha por sí solo, se quedó fuera del lenguaje y acabó siendo PDDL+ Fox y Long, 2006, que estudiamos en el Capítulo 17. Los planificadores siguieron el mismo orden: muchos más planifican con el nivel 3 que con el nivel 4.

12.3. Leer un número mientras cambia

Un efecto continuo se gana su sitio cuando alguien mira el número en mitad de la acción. Es la situación de la lámpara, y ahora podemos escribirla (Listado 12.3).

```
22  (:durative-action charge
23    :parameters (?r - robot ?l - location)
24    :duration (and
25      (>= ?duration 1)
26      (<= ?duration 60))
27    :condition (and
28      (at start (idle ?r))
29      (at start (has-charger ?l))
30      (over all (robot-at ?r ?l))
31      (over all (<= (battery ?r) (battery-capacity ?r))))
32    :effect (and
33      (at start (not (idle ?r)))
34      (increase (battery ?r) (* #t (charge-rate ?r)))
35      (at end (idle ?r))))
36
37  ; The ultraviolet lamp runs for 20 s on the battery.
38  (:durative-action sanitize
39    :parameters (?r - robot)
40    :duration (= ?duration 20)
41    :condition (and
42      (at start (lamp-off ?r))
43      (over all (>= (battery ?r) 0)))
44    :effect (and
45      (at start (not (lamp-off ?r)))
46      (decrease (battery ?r) (* #t (lamp-rate ?r)))
47      (at end (lamp-off ?r))
48      (at end (sanitized ?r)))))
```

Listado 12.3: La escena de la base de carga con efectos continuos (las dos acciones de `lamp-continuous-domain.pddl`).

Compáralo con el Listado 12.1. En `sanitize`, la lámpara ya no pide 40 unidades por adelantado. Enuncia dos hechos del mundo: la batería baja al ritmo de la lámpara mientras está encendida (línea 46) y no debe ser nunca negativa mientras la lámpara está encendida (línea 43). En `charge`, la energía llega a lo largo de toda la acción (línea 34), y el límite de la batería ya no es una cota de la duración, sino una condición `over all` (línea 31). Las dos condiciones son comparaciones sobre un número que se está moviendo, y `over all` significa para ellas lo mismo que para los hechos en el Capítulo 10: en todos los instantes entre el principio y el final de la acción.

```
$ kairos lamp-continuous-domain.pddl lamp-continuous-p01.pddl
; cost = 20
; makespan = 20.000
0.000: (sanitize r1) [20.000]
5.000: (charge r1 dock) [10.000]
note: the search minimizes the number of actions; the metric of the
     problem is reported for each plan but does not guide the search
      yet [orchestrator.metric]
```

Es el plan que el modelo prudente no podía encontrar, en 20 segundos, lo que dura la lámpara sola. Leámoslo con lápiz. De 0 a 5 solo funciona la lámpara: la batería pasa de 10 a 0. En 5 se suma el cargador. Ahora dos efectos continuos actúan sobre el mismo número, y sus ritmos se suman: $+5 - 2 = +3$ por segundo. En los 10 segundos de la carga, la batería sube de 0 a 30. En 15 termina la carga, la lámpara vuelve a estar sola y, en sus últimos 5 segundos, la batería baja hasta 20 (Figura 12.3).

Los efectos continuos de varias acciones sobre el mismo número se suman: en cada instante, el número se mueve a la suma de los ritmos de las acciones en curso.

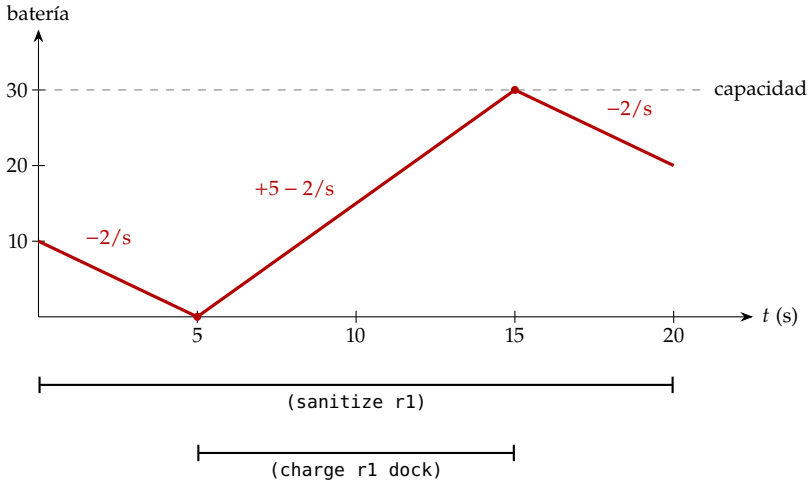


Figura 12.3.: La batería durante el plan del modelo continuo. La pendiente solo cambia donde una acción empieza o termina. Fíjate en los dos puntos: el plan toca la batería vacía en $t = 5$ y la llena en $t = 15$, y las dos condiciones, escritas con >= y <=, lo permiten.

El plan es correcto e incómodo: deja que la batería llegue exactamente a cero antes de enchufarse y la llena exactamente hasta arriba. Las condiciones dicen `(>= (battery ?r) 0)` y `(<= (battery ?r) (battery-capacity ?r))`, y un planificador se toma un modelo al pie de la letra. Un modelador que quiera una reserva escribe la reserva: `(>= (battery ?r) 5)`. El plan del principio del capítulo, con el cargador y la lámpara encendidos a la vez, también es un plan de este modelo. Está en el fichero `lamp-together.plan`, y `kairos validate` lo acepta. El plan del modelo descuidado, la lámpara sola, está en `lamp-alone.plan`:

```
$ kairos validate lamp-continuous-domain.pddl lamp-continuous-p01.
    pddl lamp-together.plan
plan valid, makespan 20
$ kairos validate lamp-continuous-domain.pddl lamp-continuous-p01.
    pddl lamp-alone.plan
lamp-alone.plan:2: error: step 1 (sanitize r1) at 0.000: over-all
    condition broken before the happening at 20.000000: (>= (
    battery ?r) 0): -30 >= 0 is false [validate.invariant_failed]
 2 | 0.000: (sanitize r1) [20.000]
plan invalid
```

El mensaje nombra la condición, la acción a la que pertenece y el valor que la incumplió: al final del intervalo que estaba comprobando el validador, la batería estaría en −30.

Cómo se comprueba un número en movimiento

Entre dos acontecimientos consecutivos de un plan (inicios y finales de acciones), el conjunto de acciones en curso no cambia, así que cada número se mueve a lo largo de un segmento recto. Una comparación como `(>= (battery ?r) 0)` solo puede volverse falsa dentro del segmento si es falsa en uno de sus extremos: una recta que está por encima de cero en los dos extremos está por encima de cero en todo el recorrido. Un validador, o un planificador, solo tiene que mirar los extremos de cada segmento. Esto es lo que hace tratable el cambio lineal, y es la razón por la que los mensajes del validador hablan de «the happening at» seguido de un instante.

Agua que viene y va: el invernadero

El mismo patrón aparece dondequiera que algo se almacena, se usa y se repone. En un invernadero, un depósito alimentado por un grifo abastece las líneas de goteo de los bancales. Regar un bancal lleva 40 segundos, y durante ese tiempo el bancal bebe del depósito a un ritmo constante (Listado 12.4).

```
;; A greenhouse: a tank filled by a tap feeds the drip lines of
;; the beds. Water comes in and goes out at the same time.
(define (domain greenhouse)
  (:requirements :typing :durative-actions :numeric-fluents
                 :duration-inequalities :continuous-effects)

  (:types tank bed)

  (:predicates
    (tap-closed ?t - tank)
    (feeds ?t - tank ?b - bed)   ; the drip line of ?b is on ?t
    (dry ?b - bed)
    (watered ?b - bed))

  (:functions
    (level ?t - tank)       ; liters in the tank now
    (capacity ?t - tank)    ; liters it can hold
    (flow ?t - tank)        ; liters per second through the tap
    (demand ?b - bed))      ; liters per second the bed drinks

  ; The tap may run while the beds drink: how long it can stay
  ; open is not known at its start, so the limit is a condition
  ; on the level all along.
  (:durative-action fill
    :parameters (?t - tank)
    :duration (and (>= ?duration 1) (<= ?duration 120))
    :condition (and
      (at start (tap-closed ?t))
      (over all (<= (level ?t) (capacity ?t))))
    :effect (and
      (at start (not (tap-closed ?t)))
      (increase (level ?t) (* #t (flow ?t)))
      (at end (tap-closed ?t))))

  ; Watering a bed takes 40 s, and the tank must not run dry.
  (:durative-action irrigate
    :parameters (?b - bed ?t - tank)
    :duration (= ?duration 40)
    :condition (and
      (at start (dry ?b))
      (over all (feeds ?t ?b))
      (over all (>= (level ?t) 0)))
    :effect (and
      (at start (not (dry ?b)))
      (decrease (level ?t) (* #t (demand ?b)))
      (at end (watered ?b)))))
```

Listado 12.4: El invernadero: un grifo llena el depósito mientras los bancales beben de él.

La acción `fill` del depósito solitario tenía una cota de duración que calculaba, al principio, cuánto tiempo podía estar abierto el grifo: `(/ (- (capacity ?t) (level ?t)) (flow ?t))`. Aquí ese cálculo sería erróneo. Si un bancal está bebiendo, el depósito tarda más en llenarse de lo que sugiere el nivel al principio, y cuánto más depende de lo que ocurra durante la acción. Así que la cota es una constante holgada, y el límite real se enuncia como lo que es, una condición `over all` sobre el nivel (línea 29). Es una regla general: *una cota calculada al principio de una acción solo es correcta si nada más cambia el número mientras tanto.*

En `greenhouse-p01.pddl` el depósito admite 30 litros y tiene 5, y un bancal bebe 1 litro por segundo: 40 litros en total, más de lo que cabe en

el depósito. Llenar primero y regar después no puede funcionar. El plan tiene que hacer las dos cosas a la vez:

```
$ kairos greenhouse-domain.pddl greenhouse-p01.pddl
; cost = 40
; makespan = 40.000
0.000: (irrigate b1 t1) [40.000]
5.000: (fill t1) [30.000]
note: the search minimizes the number of actions; the metric of the
     problem is reported for each plan but does not guide the search
      yet [orchestrator.metric]
```

La forma es la de la lámpara. El bancal bebe solo durante 5 segundos y lleva el nivel de 5 a 0; el grifo se abre y durante 30 segundos el nivel sube a $2 - 1 = 1$ litro por segundo, hasta 30; el grifo se cierra en 35 y los últimos 5 segundos de riego dejan 25 litros (Figura 12.4). La nota que hay bajo el plan aparece siempre que Kairos planifica en pasos de tiempo, como ha hecho aquí (Sección 12.6): ha encontrado un plan con pocas acciones e informa del *makespan* sin haber buscado el más corto. En los listados siguientes omitimos la nota.

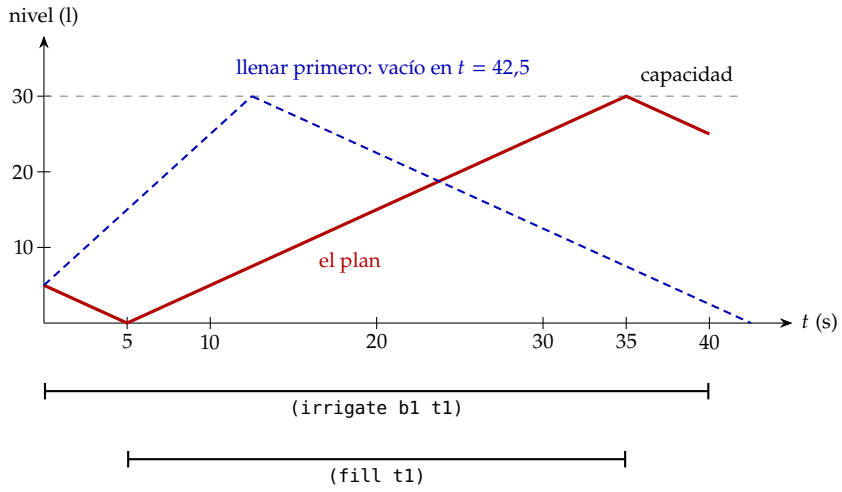


Figura 12.4.: El nivel del depósito en el plan de `greenhouse-p01` (línea continua). La línea discontinua es el plan que llena el depósito hasta arriba y riega después: 30 litros no duran 40 segundos.

El plan «llenar primero» está en el fichero `greenhouse-first.plan`: un llenado de 12,5 segundos y después el riego. El validador encuentra el punto en el que el depósito se queda seco:

```
$ kairos validate greenhouse-domain.pddl greenhouse-p01.pddl
     greenhouse-first.plan
greenhouse-first.plan:3: error: step 2 (irrigate b1 t1) at 12.501:
     over-all condition broken before the happening at 52.501000:
     (>= (level ?t) 0): -10 >= 0 is false [validate.invariant_failed
     ]
 3 | 12.501: (irrigate b1 t1) [40.000]
plan invalid
```

Con dos bancales en el mismo depósito (`greenhouse-p02.pddl`, cada uno bebiendo medio litro por segundo), tres acciones mueven el nivel a la vez:

```
$ kairos greenhouse-domain.pddl greenhouse-p02.pddl
; cost = 50.9
; makespan = 50.900
```

```
0.000: (irrigate b1 t1) [40.000]
10.000: (fill t1) [29.500]
10.900: (irrigate b2 t1) [40.000]
; better plan 2
; cost = 41
; makespan = 41.000
0.000: (fill t1) [24.000]
0.900: (irrigate b1 t1) [40.000]
1.000: (irrigate b2 t1) [40.000]
```

En el segundo plan el nivel empieza en 5 y sube a 2 litros por segundo con el grifo solo, a 1,5 cuando se suma el primer bancal y a 1 con los dos bancales, hasta que el grifo se cierra en 24 con 29,95 litros. Después baja a 1 litro por segundo mientras beben los dos bancales. Los instantes de inicio 0,9 y 1,0 no son redondos por casualidad: salen de los pasos de una décima de segundo con los que Kairos avanzó el tiempo para encontrar este plan.

Pasos O y D en el invernadero

Objetos y estado. El nivel del depósito es una magnitud (Capítulo 8). La pregunta nueva es cómo cambia: no a saltos, sino a lo largo de las acciones, a ritmos que son números estáticos (`flow`, `demand`). La tabla de estado gana una columna para ello.

Propiedad	Cómo cambia	Codificación
`(level ?t)`	mientras se ejecuta `fill`, +flow por segundo; mientras se ejecuta `irrigate`, −demand por segundo	fluente numérico
`(flow ?t)`, `(demand ?b)`	nunca	funciones estáticas

Dinámica. Para cada acción que cambia el nivel nos preguntamos quién puede estar mirando el nivel mientras tanto. `irrigate` lo mira ella misma (el depósito no debe quedarse seco), y también `fill` (no debe desbordarse), y cada una se ejecuta mientras la otra cambia el nivel. Eso lo decide: los cambios son efectos continuos y los límites son condiciones `over all`.

12.4. El generador: cuándo empezar

En los ejemplos vistos hasta ahora el planificador tenía que elegir una duración. En el siguiente, todas las duraciones son fijas y aun así hay algo que encontrar. El generador es un dominio pequeño que acompaña desde hace mucho tiempo al cambio continuo en PDDL; el artículo que presenta el validador de planes VAL lo usa para mostrar el aspecto de los planes con efectos continuos Howey et al., 2004. Un generador debe funcionar durante un tiempo fijo. Quema combustible a un ritmo constante, y su depósito es demasiado pequeño para todo el funcionamiento. Mientras funciona se pueden verter bidones en el depósito. El Listado 12.5 es nuestra versión, con un funcionamiento de 100 segundos.

```
;; The generator: it burns fuel while it runs, and jerrycans can
;; be poured into its tank meanwhile. Fuel changes continuously.
(define (domain generator)
  (:requirements :typing :durative-actions :numeric-fluents
                 :continuous-effects)
```

```
(:types generator jerrycan)

(:predicates
  (stopped ?g - generator)      ; not started yet
  (ran ?g - generator)          ; it completed its run
  (full ?j - jerrycan)
  (funnel-free ?g - generator)) ; one jerrycan at a time

(:functions
  (fuel ?g - generator)         ; liters in the tank now
  (capacity ?g - generator))    ; liters the tank holds

; Run for 100 s, burning 1 liter per second. The tank must
; never be empty while the generator runs.
(:durative-action generate
  :parameters (?g - generator)
  :duration (= ?duration 100)
  :condition (and
    (at start (stopped ?g))
    (over all (>= (fuel ?g) 0)))
  :effect (and
    (at start (not (stopped ?g)))
    (decrease (fuel ?g) (* #t 1))
    (at end (ran ?g))))

; Pour a jerrycan: 2 liters per second for 10 s. The tank must
; never overflow while pouring.
(:durative-action refuel
  :parameters (?g - generator ?j - jerrycan)
  :duration (= ?duration 10)
  :condition (and
    (at start (full ?j))
    (at start (funnel-free ?g))
    (over all (<= (fuel ?g) (capacity ?g))))
  :effect (and
    (at start (not (full ?j)))
    (at start (not (funnel-free ?g)))
    (at end (funnel-free ?g))
    (increase (fuel ?g) (* #t 2)))))
```

Listado 12.5: El generador: el combustible se quema durante el funcionamiento y mientras tanto se vierten bidones.

Las dos acciones tiran del combustible en sentidos opuestos y cada una vigila su propio límite: `generate` no debe encontrarse nunca el depósito vacío y `refuel` no debe desbordarlo nunca. Un bidón se vierte a 2 litros por segundo durante 10 segundos, 20 litros. En `generator-p01.pddl` el depósito admite 100 litros y tiene 95, y hay un bidón. El funcionamiento necesita 100 litros, así que el bidón es necesario, y 115 litros no caben en el depósito, así que no se puede verter antes de empezar. Hay que verterlo durante el funcionamiento. Pero no en cualquier momento:

- Si se vierte demasiado pronto, se desborda. Mientras se ejecutan las dos acciones, el combustible sube a $2 - 1 = 1$ litro por segundo, así que un vertido que empieza en el instante s termina con $95 - s + 10$ litros. Eso cabe en 100 solo si $s \geq 5$.
- Si se vierte demasiado tarde, el generador ya se ha parado. Solo, el generador vacía el depósito en $t = 95$.

Cualquier inicio entre 5 y 95 da un plan. Aquí el planificador no tiene una duración que elegir, solo un instante, y los instantes correctos forman una ventana que ninguna acción enuncia por sí sola: se deduce de dos ritmos, dos límites y el nivel inicial.

```
$ kairos generator-domain.pddl generator-p01.pddl
; cost = 105
; makespan = 105.000
```

```
0.000: (generate gen) [100.000]
95.000: (refuel gen j1) [10.000]
```

Kairos elige el último instante de la ventana. Deja pasar el tiempo hasta que una condición está a punto de incumplirse (el depósito se vacía en 95) y vierte entonces; la mitad del bidón va a parar a un generador que ya ha terminado, cosa que el objetivo no prohíbe. La Figura 12.5 dibuja este plan y otro más sensato, con el vertido en $t = 50$, del fichero `generator-middle.plan`.

Con cambio continuo, el instante en que empieza una acción puede decidir si un plan es válido, aunque todas las duraciones sean fijas.

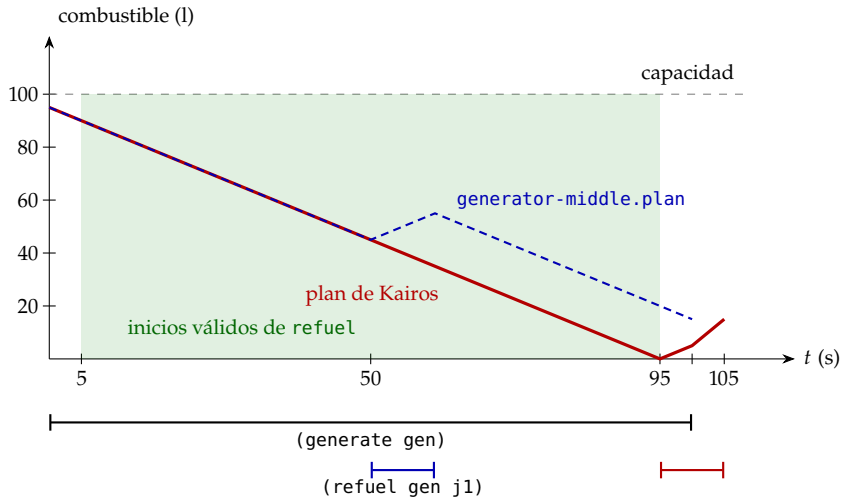


Figura 12.5.: Combustible en el depósito en dos planes válidos de `generator-p01`. La banda sombreada es la ventana en la que puede empezar el vertido. Antes de $t = 5$, la línea cruzaría la capacidad; después de $t = 95$, ya ha cruzado el cero.

Tres planes escritos a mano muestran los bordes de la ventana. Verter al principio del funcionamiento desborda el depósito, verter antes del funcionamiento lo desborda más, y verter a mitad está bien:

```
$ kairos validate generator-domain.pddl generator-p01.pddl generator-
    early.plan
generator-early.plan:3: error: step 2 (refuel gen j1) at 0.000: over-
    all condition broken before the happening at 10.000000: (<= (
    fuel ?g) (capacity ?g)): 105 <= 100 is false [validate.
    invariant_failed]
 3 | 0.000: (refuel gen j1) [10.000]
plan invalid
$ kairos validate generator-domain.pddl generator-p01.pddl generator-
    before.plan
generator-before.plan:2: error: step 1 (refuel gen j1) at 0.000:
    over-all condition broken before the happening at 10.000000:
    (<= (fuel ?g) (capacity ?g)): 115 <= 100 is false [validate.
    invariant_failed]
 2 | 0.000: (refuel gen j1) [10.000]
plan invalid
$ kairos validate generator-domain.pddl generator-p01.pddl generator-
    middle.plan
plan valid, makespan 100
```

Con 65 litros y dos bidones (`generator-p02.pddl`), un bidón cabe antes del funcionamiento y el otro no:

```
$ kairos generator-domain.pddl generator-p02.pddl
; cost = 110.001
; makespan = 110.001
0.000: (refuel gen j1) [10.000]
```

```
10.001: (generate gen) [100.000]
95.001: (refuel gen j2) [10.000]
```

El primer vertido deja el depósito en 85 litros, que duran hasta $t = 95{,}001$, y ahí llega el segundo bidón, otra vez en el último instante.

El tercer problema no tiene plan: 60 litros y un bidón son 80 litros para un funcionamiento que quema 100.

```
$ kairos generator-domain.pddl generator-p03.pddl
note: no plan in steps of time; a plan that needs finer timing than
     the steps would be missed, so this is evidence but not a proof
     [orchestrator.steps_exhausted]
note: no plan: the engines that apply ran out of states (136 states)
      before the time limit, so more time does not help; none of
     them is complete for this task, so a plan may still exist (see
     the note before). [orchestrator.why]
kairos: no plan found (exhausted)
$ echo $?
14
```

La respuesta es `no plan found`, con código de salida 14, y no el `unsolvable` con código 10 que conocemos de los problemas clásicos. Las notas dicen por qué, y son honestas sobre un límite real: la búsqueda ha probado todos los instantes a los que llegan sus pasos de tiempo y no ha encontrado nada, pero el tiempo es continuo y los pasos no, así que esto es un indicio y no una demostración. Aquí podemos aportar la demostración nosotros mismos, con una resta.

12.5. El horno: cuándo parar

El generador pedía el instante en que empezar una acción de duración fija. Un horno pide lo contrario: el instante en que parar una acción de duración libre. La resistencia sube la temperatura 2 grados por segundo mientras está encendida, y cada plato se hornea durante 10 segundos dentro de su propio intervalo de temperaturas (Listado 12.6).

```
;; An oven: the heater raises the temperature at a constant rate,
;; and each dish bakes only inside its own range of temperatures.
(define (domain oven)
  (:requirements :typing :durative-actions :numeric-fluents
                 :duration-inequalities :continuous-effects)

  (:types dish)

  (:predicates
    (heater-off)
    (oven-empty)
    (raw ?d - dish)
    (baked ?d - dish))

  (:functions
    (temperature)          ; degrees inside the oven now
    (heat-rate)            ; degrees per second with the heater on
    (min-temp ?d - dish)
    (max-temp ?d - dish))

  ; Heater on for as long as the planner decides.
  (:durative-action heat
    :parameters ()
    :duration (and (>= ?duration 1) (<= ?duration 300))
    :condition (at start (heater-off))
```

```
   :effect (and
     (at start (not (heater-off)))
     (increase (temperature) (* #t (heat-rate)))
     (at end (heater-off))))

 ; Ten seconds in the oven, never too cold and never too hot.
 (:durative-action bake
   :parameters (?d - dish)
   :duration (= ?duration 10)
   :condition (and
     (at start (raw ?d))
     (at start (oven-empty))
     (over all (>= (temperature) (min-temp ?d)))
     (over all (<= (temperature) (max-temp ?d))))
   :effect (and
     (at start (not (raw ?d)))
     (at start (not (oven-empty)))
     (at end (oven-empty))
     (at end (baked ?d)))))
```

Listado 12.6: El horno: la resistencia funciona durante un tiempo elegido, y un plato necesita la temperatura dentro de su intervalo durante todo el horneado.

La duración de `heat` puede ser cualquier valor de 1 a 300 segundos. En `oven-p01.pddl` el horno está a 20 grados y el pan se hornea entre 200 y 220. Calentar debe durar, por tanto, entre 90 y 100 segundos: ninguna de las dos cotas de la acción, sino un valor intermedio que ninguna línea del dominio menciona. Se deduce de las dos condiciones de `bake`.

```
$ kairos oven-domain.pddl oven-p01.pddl
; cost = 100.001
; makespan = 100.001
0.000: (heat) [90.000]
90.001: (bake bread) [10.000]
```

La resistencia se para a los 90 segundos, el primer instante en que el pan puede entrar. Con un segundo plato, una pizza que se hornea entre 210 y 240 grados (`oven-p02.pddl`), hay dos planes:

```
$ kairos oven-domain.pddl oven-p02.pddl
; cost = 115.00300000000001
; makespan = 115.003
0.000: (heat) [90.000]
90.001: (bake bread) [10.000]
100.002: (heat) [5.000]
105.003: (bake pizza) [10.000]
; better plan 2
; cost = 114.90100000000001
; makespan = 114.901
0.000: (heat) [95.000]
94.900: (bake bread) [10.000]
104.901: (bake pizza) [10.000]
```

El primer plan calienta dos veces. El segundo calienta una vez, hasta 210 grados, la única temperatura a la que se pueden atender los intervalos de los dos platos uno tras otro sin volver a tocar la resistencia (cualquier valor de 210 a 220 serviría). Fíjate en su segunda línea: el pan entra en 94,9, una décima de segundo *antes* de que se pare la resistencia, con el horno a 209,8 grados y todavía subiendo. La condición de `bake` se comprueba sobre una temperatura que se está moviendo, y se cumple (Figura 12.6).

Cuando la duración de una acción es libre, el momento adecuado para terminarla lo pueden fijar las condiciones de otras acciones que leen el número que ella cambia.

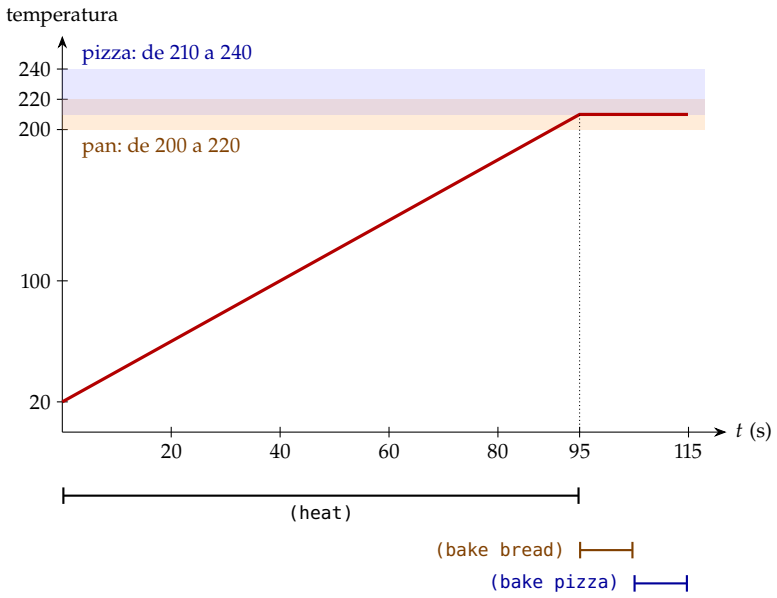


Figura 12.6.: La temperatura en el segundo plan de `oven-p02`. La resistencia se para a 210 grados, dentro de los dos intervalos (la banda más oscura, donde se solapan). Un calentamiento de 90 segundos, el primer valor que le va bien al pan, dejaría fuera la pizza.

En `oven-short.plan`, un cocinero con prisa para la resistencia a los 80 segundos, y el validador muestra que el pan y el horno discrepan en 20 grados:

```
$ kairos validate oven-domain.pddl oven-p01.pddl oven-short.plan
oven-short.plan:3: error: step 2 (bake bread) at 80.001: over-all
     condition broken after the happening at 80.001000: (>= (
     temperature) (min-temp ?d)): 180 >= 200 is false [validate.
     invariant_failed]
 3 | 80.001: (bake bread) [10.000]
plan invalid
```

Nuestro horno no pierde calor cuando la resistencia está apagada, y nada en él enfría. Las dos cosas son decisiones del paso M: el alcance de este modelo es «calentar para hornear», y un plato que necesite un horno más frío que el actual no tiene plan. El Ejercicio 12.5.2 añade una forma de enfriar.

12.6. Lo que debe encontrar el planificador

Un planificador clásico elige qué acciones y en qué orden. Un planificador temporal, desde el Capítulo 10, además las programa en el tiempo, y desde el Capítulo 11 puede elegir duraciones. Pero hasta ahora toda elección de tiempo se podía hacer tarde y con poco coste: una vez fijadas las acciones y su orden, cualquier calendario que respetara el orden y las duraciones era bueno. El cambio continuo rompe esto. En el generador, las mismas dos acciones en el mismo orden son un plan o no según un número, el inicio del vertido, y los números válidos son los de un intervalo. En el horno es una duración. El planificador tiene que encontrar números reales, y las condiciones que los restringen están repartidas entre varias acciones.

Para el cambio lineal hay una salida, que anunciaba la caja de concepto de la Sección 12.3: como los números se mueven a lo largo de segmentos rectos entre acontecimientos, solo importan los acontecimientos, y las condiciones sobre ellos son desigualdades lineales sobre los instantes

del plan. Un planificador puede reunirlas y pedir a un resolutor de programación lineal unos instantes que las satisfagan todas. Es el enfoque de los planificadores de la familia de POPF Coles et al., 2010. Otros planificadores, y Kairos es uno de ellos, combinan mecanismos más sencillos.

Cómo planifica Kairos el cambio continuo

Kairos planifica una tarea con efectos continuos de dos formas, y valida lo que encuentre cualquiera de ellas.
Como cambios discretos. Cuando un número que cambia de forma continua solo se lee como «lo bastante grande» o solo como «lo bastante pequeño», cada efecto continuo se sustituye por un salto de su cantidad total, colocado en el instante más seguro para las condiciones que leen el número: lo que se gasta sale al principio de la acción y lo que se repone llega al final. Cada duración dada por una desigualdad se prueba en sus dos cotas. Es nuestro modelo prudente de la Sección 12.1, construido por el planificador, y lo resuelven los motores temporales de siempre.
En pasos de tiempo. La tarea también se busca dejando pasar el tiempo: un paso de 0,1 segundos, o un salto al instante en que alguna condición numérica está a punto de cambiar. Una acción de duración libre termina entonces en un paso, o en el instante en que terminarla hace cierta una condición de alguna acción. Así se encuentran planes en los que las acciones empujan un número en sentidos opuestos, y duraciones entre las cotas.
Validación. Todo plan se comprueba con la semántica continua real, segmento a segmento, antes de imprimirlo.

Con la opción `-v`, el planificador dice qué camino ha tomado. El mensajero de la sección siguiente se planifica como cambios discretos, con los pasos de tiempo al lado; el generador, en el que `refuel` necesita el combustible lo bastante bajo y `generate` lo necesita lo bastante alto, solo en pasos:

```
$ kairos courier-battery-domain.pddl courier-battery-p01.pddl -v
...
[   0.001s] [orchestrator] durations: 32 durative actions after
     fixing durations and continuous effects
[   0.001s] [orchestrator] engine M2: 32 durative actions compressed
[   0.002s] [orchestrator] also in steps of 0.1 s
...
$ kairos generator-domain.pddl generator-p01.pddl -v
...
[   0.001s] [orchestrator] durations: (generate gen): a continuous
     effect on a variable read in other ways than "large enough" or
     "small enough" alone; planned as PDDL+
[   0.001s] [orchestrator] PDDL+: 0 processes and 0 events, steps of
      0.1 s, 0 pairs of symmetric objects
...
```

(«Planned as PDDL+» nombra la maquinaria, que describe el Capítulo 17; el modelo sigue siendo el que escribimos.) Tres consecuencias de esta forma de planificar se ven en las salidas del capítulo, y es mejor conocerlas que llevarse una sorpresa con ellas.

- **Las duraciones salen en sus cotas, o donde una condición las fija.** El horno se calentó exactamente 90 segundos porque en ese instante terminar `heat` hacía posible `bake`. Una duración que nada fija sale como la más corta o la más larga permitida. Veremos al

mensajero cargar durante 5 segundos, el mínimo, o hasta llenar la batería.

- **Los planes esperan hasta el último momento.** El salto «al instante en que una condición está a punto de cambiar» es la razón de que el bidón se vierta en $t = 95$ y de que el grifo del invernadero se abra cuando el depósito está vacío.
- **Sin plan, la respuesta puede ser `no plan found`.** Los pasos de tiempo no pueden visitar todos los números reales, así que quedarse sin estados no demuestra nada. Kairos dice entonces `no plan found` (código de salida 14) y reserva `unsolvable` (código de salida 10) para lo que ha demostrado.

El tercer punto merece un hábito. Cuando un modelo con efectos continuos no tiene plan, y crees que debería tenerlo, escribe el plan a mano y dáselo a `kairos validate`. El validador trabaja en tiempo continuo y su veredicto es exacto para el cambio lineal: si acepta tu plan, es un plan, diga lo que diga la búsqueda.

Portabilidad. Los efectos continuos están entre las características de PDDL con menos soporte. En la prueba que mantiene la documentación de Kairos, un problema pequeño con diez segundos por planificador, solo POPF devolvió un plan válido además de Kairos; OPTIC, TFD, LPG-td, ENHSP, Metric-FF y LAMA no.

Planificar con cambio continuo significa encontrar instantes y duraciones, no solo acciones. Kairos prueba las cotas y los instantes en que cambian las condiciones, y valida todos los planes en tiempo continuo.

12.7. El mundo del mensajero: una batería que se descarga y se carga

El mensajero del Capítulo 11 pagaba un desplazamiento al empezarlo y cobraba una carga al terminarla. Ahora le damos la batería de este capítulo: baja a lo largo de todo un desplazamiento, sube a lo largo de toda una carga, y el planificador elige cuánto tiempo cargar.

Paso M: lo que cambia en la ficha del modelo

Alcance, dentro: la carga de la batería en todo instante, no solo entre acciones. **Alcance, fuera:** el consumo mientras el robot está parado, recoge o deja algo; una descarga que dependa de la carga transportada o de la carga que queda. **Supuestos:** la batería se descarga a un ritmo constante mientras el robot se mueve y se llena a un ritmo constante mientras está enchufado; un robot solo se enchufa en una habitación con cargador; una carga de menos de 5 segundos no compensa ir a la base. La tabla de estado gana dos funciones estáticas del vocabulario común del mundo del mensajero: `(drain-rate ?r)` y `(charge-rate ?r)`, ambas en unidades de batería por segundo.

El dominio es el mundo del mensajero con tipos y con las acciones durativas del Capítulo 10: `move` dura la distancia dividida por la velocidad, `pick` y `drop` duran 2 segundos, y `(idle ?r)` hace que el robot haga una sola cosa a la vez. El Listado 12.7 muestra las funciones y `move`, y el Listado 12.8 muestra `charge`.

```
  (:functions
    (distance ?from ?to - location)  ; meters
    (speed ?r - robot)               ; meters per second
    (battery ?r - robot)             ; charge, 0 to 100
    (battery-capacity ?r - robot)
    (drain-rate ?r - robot)          ; per second, moving
    (charge-rate ?r - robot))        ; per second, charging

  ; The battery goes down all along the way, and must never
  ; reach a negative value on it.
  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
    :condition (and
```

```
      (at start (robot-at ?r ?from))
      (at start (idle ?r))
      (over all (connected ?from ?to))
      (over all (>= (battery ?r) 0)))
    :effect (and
      (at start (not (robot-at ?r ?from)))
      (at start (not (idle ?r)))
      (decrease (battery ?r) (* #t (drain-rate ?r)))
      (at end (robot-at ?r ?to))
      (at end (idle ?r))))
```

Listado 12.7: Las funciones y la acción `move` de `courier-battery-domain.pddl`.

```
  ; Charge for as long as the planner wants: at least 5 s, at
  ; most until the battery is full.
  (:durative-action charge
    :parameters (?r - robot ?l - location)
    :duration (and
      (>= ?duration 5)
      (<= ?duration (/ (- (battery-capacity ?r) (battery ?r))
                       (charge-rate ?r))))
    :condition (and
      (at start (idle ?r))
      (at start (has-charger ?l))
      (over all (robot-at ?r ?l)))
    :effect (and
      (at start (not (idle ?r)))
      (increase (battery ?r) (* #t (charge-rate ?r)))
      (at end (idle ?r)))))
```

Listado 12.8: La acción `charge` de `courier-battery-domain.pddl`.

`move` enuncia la descarga como un efecto continuo (línea 39) y el límite como una condición sobre todo el trayecto (línea 35). No calcula cuánta batería necesita el desplazamiento: eso se deduce del ritmo y de la duración, y el validador y el planificador lo calculan. `charge` es el grifo solitario del Listado 12.2: su cota superior es el tiempo hasta tener la batería llena, calculado al principio. Según la regla de la Sección 12.3, eso solo es correcto si nada más cambia la batería durante una carga, y aquí nada puede hacerlo: la carga retiene `(idle ?r)`, así que el robot no puede moverse mientras tanto, y solo los desplazamientos descargan.

Ensayar el modelo

El paso E pide cuatro clases de problemas. Todos usan el mapa y las distancias del mundo del mensajero, un robot con una velocidad de 1 metro por segundo, un ritmo de descarga de 1 y un ritmo de carga de 5. A esa velocidad, un metro cuesta una unidad de batería.

Uno trivial. En `courier-battery-p01.pddl` la batería está llena. El plan es el del Capítulo 10, 42,006 segundos sin carga, y los 38 metros dejan 62 unidades.

Uno típico. En `courier-battery-p02.pddl` el robot empieza en la base de carga con 20 unidades y la misma entrega, que necesita 38.

```
$ kairos courier-battery-domain.pddl courier-battery-p02.pddl
; cost = 47.007
; makespan = 47.007
0.000: (charge r1 dock) [5.000]
5.001: (move r1 dock lobby) [4.000]
9.002: (move r1 lobby pharmacy) [6.000]
15.003: (pick r1 meds1 pharmacy) [2.000]
17.004: (move r1 pharmacy lobby) [6.000]
23.005: (move r1 lobby corridor) [10.000]
33.006: (move r1 corridor ward-a) [12.000]
```

```
45.007: (drop r1 meds1 ward-a) [2.000]
```

Una carga de 5 segundos, la más corta permitida, lleva la batería de 20 a 45, y el viaje la baja a 7 (Figura 12.7). La necesidad exacta era de 18 unidades, 3,6 segundos de carga, pero la acción no permite menos de 5.

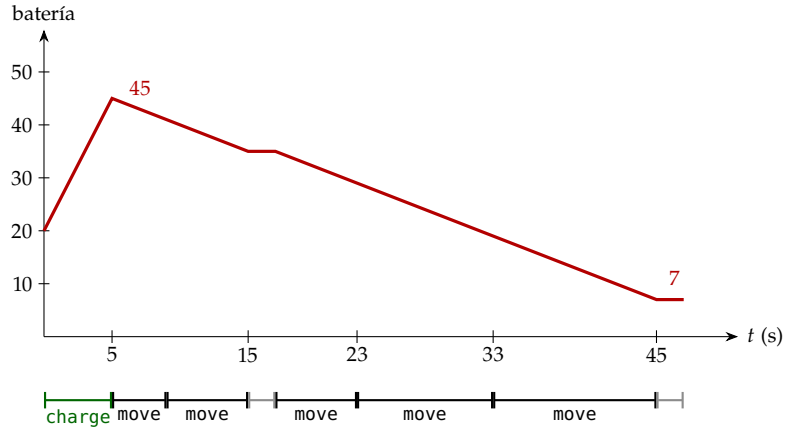


Figura 12.7.: La batería de `r1` durante el plan de `courier-battery-p02`. Sube 5 unidades por segundo durante la carga y baja 1 por segundo durante cada desplazamiento. Fíjate en los dos tramos planos, la recogida y la descarga (en gris): el modelo dice que no cuestan nada.

Uno sin plan. En `courier-battery-p04.pddl` el robot está en el vestíbulo con 3 unidades. La base de carga está a 4 metros.

```
$ kairos courier-battery-domain.pddl courier-battery-p04.pddl
note: engine M3 found no plan; as POPF, it keeps one partial plan
     per state, so this is strong evidence but not a proof [
     orchestrator.m3_exhausted]
note: no plan in steps of time; a plan that needs finer timing than
     the steps would be missed, so this is evidence but not a proof
     [orchestrator.steps_exhausted]
note: no plan: the engines that apply ran out of states (35 states)
     before the time limit, so more time does not help; none of them
      is complete for this task, so a plan may still exist (see the
     note before). [orchestrator.why]
kairos: no plan found (exhausted)
```

Treinta y cinco estados han bastado para quedarse sin nada que probar: con 3 unidades, el robot no puede terminar ningún desplazamiento.

Uno que tienta al planificador a hacer trampa. El robot está en el vestíbulo con 30 unidades (`courier-battery-p05.pddl`) y la entrega desde allí tiene 34 metros. Un modelo que mirase la batería solo al principio de cada desplazamiento dejaría empezar el último con 8 unidades para 12 metros. Ese plan está en `courier-battery-cheat.plan`, y nuestro modelo lo rechaza en el punto exacto:

```
$ kairos validate courier-battery-domain.pddl courier-battery-p05.
     pddl courier-battery-cheat.plan
courier-battery-cheat.plan:6: error: step 5 (move r1 corridor ward-a)
      at 24.004: over-all condition broken before the happening at
     36.004000: (>= (battery ?r) 0): -4 >= 0 is false [validate.
     invariant_failed]
 6 | 24.004: (move r1 corridor ward-a) [12.000]
plan invalid
$ kairos courier-battery-domain.pddl courier-battery-p05.pddl
; cost = 51.007999999999996
; makespan = 51.008
```

```
0.000: (move r1 lobby pharmacy) [6.000]
6.001: (pick r1 meds1 pharmacy) [2.000]
8.002: (move r1 pharmacy lobby) [6.000]
14.003: (move r1 lobby dock) [4.000]
18.004: (charge r1 dock) [5.000]
23.005: (move r1 dock lobby) [4.000]
27.006: (move r1 lobby corridor) [10.000]
37.007: (move r1 corridor ward-a) [12.000]
49.008: (drop r1 meds1 ward-a) [2.000]
```

El plan honrado recoge los medicamentos, vuelve por el vestíbulo, se desvía a la base de carga con 14 unidades, carga 5 segundos y entrega con 13 de sobra (Figura 12.8).

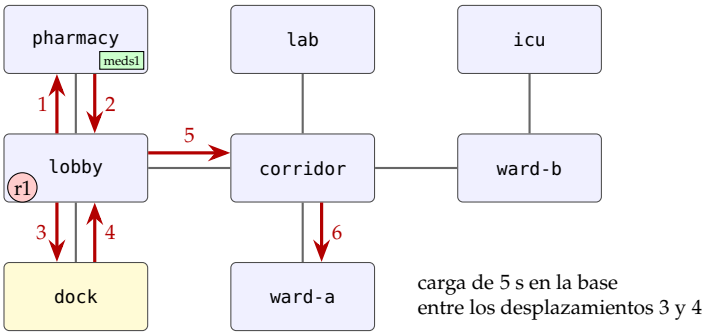


Figura 12.8.: El plan de la prueba de la trampa `courier-battery-p05`. El camino directo (desplazamientos 1, 2, 5 y 6) tiene 34 metros y la batería dura 30. El rodeo por la base de carga añade 8 metros y una carga.

Dónde cargar, y cuánto tiempo

El último problema es aquel para el que se escribió el capítulo. En `courier-battery-p03.pddl` el robot empieza en la base de carga con 60 unidades y tiene una ronda de 90 metros: medicamentos de la farmacia a la UCI, una muestra de la UCI al laboratorio y vuelta a la base. Tiene que cargar, y pasa por la base más de una vez. Kairos imprime cuatro planes en unas centésimas de segundo y luego sigue buscando uno mejor hasta haberlo probado todo, lo que aquí lleva varios segundos; lo paramos a los tres con `-timeout`. El listado conserva las primeras líneas de cada plan y sus cargas.

```
$ kairos courier-battery-domain.pddl courier-battery-p03.pddl --
    timeout 3
; cost = 158.42300000000006
; makespan = 158.423
0.000: (move r1 dock lobby) [4.000]
...
22.005: (charge r1 dock) [5.000]
...
93.016: (charge r1 dock) [19.400]
...
; better plan 2
; cost = 149.02400000000006
; makespan = 149.024
0.000: (move r1 dock lobby) [4.000]
...
22.005: (charge r1 dock) [5.000]
...
93.016: (charge r1 dock) [5.000]
98.017: (charge r1 dock) [5.000]
...
; better plan 3
; cost = 148.82100000000003
; makespan = 148.821
```

```
0.000: (charge r1 dock) [5.000]
...
85.014: (charge r1 dock) [17.800]
...
; better plan 4
; cost = 106.01600000000002
; makespan = 106.016
0.000: (charge r1 dock) [8.000]
8.001: (move r1 dock lobby) [4.000]
12.002: (move r1 lobby pharmacy) [6.000]
18.003: (pick r1 meds1 pharmacy) [2.000]
20.004: (move r1 pharmacy lobby) [6.000]
26.005: (move r1 lobby corridor) [10.000]
36.006: (move r1 corridor ward-b) [12.000]
48.007: (move r1 ward-b icu) [5.000]
53.008: (drop r1 meds1 icu) [2.000]
55.009: (pick r1 sample1 icu) [2.000]
57.010: (move r1 icu ward-b) [5.000]
62.011: (move r1 ward-b corridor) [12.000]
74.012: (move r1 corridor lab) [8.000]
82.013: (drop r1 sample1 lab) [2.000]
84.014: (move r1 lab corridor) [8.000]
92.015: (move r1 corridor lobby) [10.000]
102.016: (move r1 lobby dock) [4.000]
```

Los cuatro planes son cuatro respuestas a «dónde y cuánto tiempo». Los tres primeros recogen los medicamentos, vuelven a cargar, entregan y vuelven a cargar otra vez, con cargas de todas las clases: el mínimo de 5 segundos, dos mínimos seguidos y duraciones que no son cota de nada, 19,4 y 17,8 segundos, encontradas en pasos de tiempo. El último plan ve que basta con una carga al principio: 8 segundos, de 60 a la batería llena, la cota superior de la acción. La ronda gasta 90 unidades y el robot vuelve a la base con 10.

¿Es el mejor plan? Seis segundos de carga habrían bastado, y el plan sería dos segundos más corto. Kairos no encuentra esa duración: nada en el modelo convierte «90 unidades» en una condición que terminar la carga satisfaga; es la suma de lo que necesitan quince desplazamientos posteriores. El plan es válido, y está cerca del mejor. Lo que podemos decir del modelo es más útil que lo que podemos decir del planificador: la cota inferior de 5 segundos, que parecía un detalle del paso M, es lo que da forma a los tres primeros planes. Una cota de una duración libre es una decisión de modelado con consecuencias, y el Ejercicio 12.6.2 las examina.

El mensajero carga durante un tiempo que elige el planificador. Los planes son válidos para la batería continua; las duraciones salen en sus cotas o de los pasos de tiempo, así que el último plan puede cargar más de lo estrictamente necesario.

12.8. El método MODEL: cambio continuo

Los efectos continuos añaden preguntas a dos pasos del método MODEL, y las respuestas deciden si hace falta siquiera esta característica.

Preguntas nuevas para los pasos O y D

Objetos y estado. Para cada magnitud de la tabla de estado:

- ¿Cambia *mientras* se ejecuta una acción, o solo cuando las acciones empiezan y terminan?
- Si cambia mientras tanto: ¿a qué ritmo, y es constante el ritmo?

Dinámica. Para cada acción que cambia una magnitud así:

- ¿Lee alguien la magnitud mientras se ejecuta la acción: una

condición de otra acción que pueda ejecutarse a la vez, o un límite que la propia acción deba respetar todo el tiempo?
- ¿Puede otra acción cambiar la misma magnitud a la vez?
- ¿Es la duración una decisión? Entonces, ¿qué la acota: una constante, un valor calculado al principio (correcto solo si nadie más cambia la magnitud) o una condición `over all`?

Si nadie lee la magnitud mientras tanto y nadie más la cambia, basta con un cambio discreto: al principio si la acción consume, al final si produce.

La última frase de la caja es la importante. Los efectos continuos son caros: pocos planificadores los aceptan, y los que lo hacen buscan con más esfuerzo. Deben ser la respuesta a una necesidad del problema, y la necesidad es siempre la misma: alguien mira el número en mitad de la acción.

Alternativas comparadas

La escena de la base de carga nos ha dado tres codificaciones de la misma batería, todas ejecutadas en este capítulo: el modelo discreto descuidado (la lámpara se paga al final), el modelo discreto prudente (se paga al principio y la carga se abona al final) y el modelo continuo. La Tabla 12.1 los juzga con los cinco criterios.

Tabla 12.1.: Tres codificaciones de una batería que se usa mientras se carga.

	Discreto descuidado	Discreto prudente	Continuo
Fidelidad	Planes imposibles: la lámpara sola, batería en −30	Ningún plan imposible; pierde todos los planes que usan la energía a medida que llega (sin plan para la batería vieja)	Exacto para ritmos constantes
Tamaño	2 acciones instanciadas	2 acciones instanciadas	2 acciones instanciadas
Guía	Fácil, y equivocada	Búsqueda temporal corriente	El planificador también debe encontrar instantes y duraciones; sin plan, a menudo sin demostración
Claridad	Esconde la regla de la batería	El modelador calcula los totales a mano (`(* 20 (lamp-rate ?r))`)	Enuncia ritmos y límites tal como son en el mundo
Portabilidad	Cualquier planificador temporal con números	Necesita `?duration` en los efectos (Kairos, OPTIC, POPF)	Kairos y POPF

Veredicto. El modelo descuidado nunca es la elección correcta: un plan equivocado cuesta más que no tener plan. Entre los otros dos, decide la pregunta de la caja de método.

- *Elige el modelo discreto prudente* cuando cada magnitud la cambia una sola acción a la vez y nadie la lee mientras tanto. El mensajero de

este capítulo es uno de esos casos: un robot que hace una sola cosa a la vez, con una batería que solo tocan sus propios desplazamientos y cargas. Su modelo continuo y uno discreto prudente tienen los mismos planes, y el propio Kairos convierte el primero en el segundo para planificarlo. Aun así, la versión continua es el texto más claro, porque enuncia un ritmo donde la discreta enuncia un producto.

- *Elige el modelo continuo* cuando los productores y los consumidores de la misma magnitud pueden solaparse (la lámpara y el cargador, el grifo y los bancales, el generador y el bidón), o cuando un límite debe cumplirse en mitad de una acción larga y la acción no puede saber al principio qué más va a ocurrir.
- *Ve más allá de los dos*, a los procesos del Capítulo 17, cuando la magnitud cambia aunque el agente no haga nada: una batería que se descarga siempre que el robot está encendido.

12.9. Errores comunes

`#t` bajo una etiqueta de tiempo

Un efecto continuo no es ni `at start` ni `at end`. Escribir la etiqueta por costumbre, como en `tank-wrong-domain.pddl`,

```
      (at end (increase (level ?t) (* #t (flow ?t)))))))
```

es un error que la comprobación señala:

```
$ kairos check tank-wrong-domain.pddl
tank-wrong-domain.pddl:29:39: error: in durative action 'fill':
     #t can only appear in increase/decrease effects of durative
      actions and processes [semantic.misplaced_continuous]
 29 |       (at end (increase (level ?t) (* #t (flow ?t)))))))
    |                                       ^
invalid
```

Si lo que quieres es la cantidad al final, el efecto discreto del Capítulo 11 lo dice con `?duration`: `(at end (increase (level ?t) (* ?duration (flow ?t))))`. El mismo mensaje aparece para `#t` en un `assign` o en una acción instantánea.

Un límite comprobado solo al principio

Un efecto continuo sin una condición que lea el número todo el tiempo no cambia nada: el número se mueve y nadie mira. La condición `(at start (>= (battery ?r) 0))` es cierta para casi cualquier batería y no dice nada del camino. El síntoma es un plan que se acepta y no debería aceptarse, como con el modelo descuidado de la Sección 12.1:

```
$ kairos validate lamp-careless-domain.pddl lamp-careless-p01.
    pddl lamp-alone.plan
plan valid, makespan 20
```

Un límite que debe cumplirse durante una acción es una condición `over all`. La prueba de la trampa del paso E es la herramienta que detecta su ausencia: un problema cuyo único plan corto incumple el límite en mitad de una acción.

Leer `no plan found` como «no hay plan»

Con efectos continuos, Kairos a menudo no puede demostrar que un problema no tiene plan, y lo dice:

```
$ kairos generator-domain.pddl generator-p03.pddl
...
kairos: no plan found (exhausted)
```

`unsolvable` (código de salida 10) es una demostración. `no plan found` (código de salida 14) significa que la búsqueda ha terminado sin plan y sin demostración; las notas que lo preceden dicen qué parte de la búsqueda era incompleta. Los guiones que prueban un modelo deben esperar el código que corresponda, y un modelador que dude debe escribir el plan a mano y validarlo.

Un requisito olvidado

Kairos planifica un dominio que usa `#t` sin declarar `:continuous-effects`, con un aviso que muestra `kairos check`:

```
warning: uses continuous-effects but :requirements does not
     include :continuous-effects [semantic.
     undeclared_requirement]
```

Otros planificadores leen los requisitos para decidir si pueden planificar el dominio, y algunos se detienen en la primera construcción que no se declaró. Declara lo que usa el fichero.

12.10. Resumen

Un número que solo salta al principio o al final de una acción no tiene valor entre medias. Los efectos continuos le dan uno. Indican un ritmo, se suman cuando se ejecutan varias acciones, y las condiciones `over all` pueden vigilar el número mientras se mueve. El precio lo pagan el planificador, que debe encontrar instantes y duraciones además de acciones, y el modelador, que tiene menos planificadores entre los que elegir y respuestas más débiles cuando no hay plan.

Construcción o idea	Qué recordar
`:continuous-effects`	El requisito; con `:durative-actions` y `:numeric-fluents`
`(increase (f) (* #t r))`	Mientras se ejecuta la acción, `f` crece `r` por segundo; sin etiqueta de tiempo; solo `increase` y `decrease`
Varios efectos sobre un número	Los ritmos se suman
`(over all (>= (f) 0))`	Se cumple en todos los instantes de la acción; para el cambio lineal basta con mirar donde empiezan y terminan las acciones
Cota de una duración libre	Una constante, un valor calculado al principio (solo si nadie más cambia el número) o una condición `over all`
Lo que encuentra el planificador	Qué acciones, cuándo empieza cada una, cuánto dura cada una
Kairos	Cotas de las duraciones y saltos más seguros, más pasos de tiempo; todo plan validado en tiempo continuo; `no plan found` no es una demostración
Cuándo usarlo	Alguien lee el número, o también lo cambia, mientras se ejecuta la acción

Tabla 12.2.: Lo que ha añadido este capítulo.

12.11. Lecturas complementarias

Los efectos continuos se definen en el artículo de PDDL 2.1 Fox y Long, 2003: sus secciones sobre el nivel 4 dan la sintaxis de `#t` y la semántica de los planes en los que los números cambian durante las acciones, y explican por qué el lenguaje separa los efectos discretos de los continuos. La validación de esos planes, con el generador como ejemplo, se describe junto con la herramienta VAL Howey et al., 2004. En cuanto a la planificación, la búsqueda hacia delante de POPF Coles et al., 2010 es la referencia para el cambio continuo lineal en un planificador que también maneja la concurrencia requerida del Capítulo 11; OPTIC Benton et al., 2012 extiende esa línea de planificadores hacia la calidad del plan. El libro de Haslum y sus colegas Haslum et al., 2019 sitúa los efectos continuos en toda la familia de versiones de PDDL. La forma en que los planifica Kairos, que resumió la Sección 12.6, está documentada junto con el planificador Martín Rico et al., 2026. El cambio que el mundo pone en marcha por sí solo, con procesos y eventos, es el tema de PDDL+ Fox y Long, 2006 y del Capítulo 17.

12.12. Ejercicios

Ejercicios de la Sección 12.1 — Una escalera donde el mundo tiene una rampa

Ejercicio 12.1.1. Un robot con 50 unidades de batería hace un desplazamiento de 30 segundos que descarga 1 unidad por segundo y, durante ese desplazamiento, del segundo 10 al 20, una cámara del robot toma fotos, para lo que necesita la batería a 35 o más. Di si la cámara tiene batería suficiente (a) en el mundo, (b) en un modelo en el que el desplazamiento toma sus 30 unidades al principio, (c) en un modelo en el que las toma al final. ¿Cuál de los dos modelos se equivoca en la dirección peligrosa?

Ejercicio 12.1.2. En `lamp-discrete-p01.pddl`, cambia la capacidad de la batería a 100 y ejecuta el modelo prudente. Lee el plan y compara su *makespan* con los 20 segundos del plan de la Sección 12.3. ¿Qué ha perdido esta vez el modelo prudente?

Ejercicios de la Sección 12.2 — Efectos continuos

Ejercicio 12.2.1. Añade al depósito del Listado 12.2 una válvula de desagüe: una acción `drain` que baje el nivel a `(outflow ?t)` litros por segundo durante un tiempo que elija el planificador, como mucho hasta que el depósito quede vacío. Escribe un problema en el que haya que vaciar un depósito con 80 litros, con un caudal de salida de 4 litros por segundo, y predice el plan antes de ejecutarlo.

Ejercicio 12.2.2. Para cada uno de estos efectos de una acción durativa, di si es un efecto continuo correcto, un efecto discreto correcto o un error, y comprueba tus respuestas con `kairos check`: (a) `(increase (level ?t) (* (flow ?t) #t))`; (b) `(at start (decrease (level ?t) (* #t 2)))`; (c) `(assign (level ?t) (* #t (flow ?t)))`; (d) `(at end (increase (level ?t) (* ?duration (flow ?t))))`.

Ejercicio 12.2.3. El depósito de `tank-p01.pddl` se llena con una acción de 50 segundos. Cambia el objetivo a `(>= (level t1) 80)`, ejecuta el planificador y explica el último plan que imprime. Después di qué cambiarías en el *dominio* si un grifo no se puede abrir y cerrar cada segundo.

Ejercicios de la Sección 12.3 — Leer un número mientras cambia

Ejercicio 12.3.1. En el plan de `greenhouse-p01.pddl`, el grifo se abre en $t = 5$ durante 30 segundos. Demuestra a mano que ningún otro inicio es válido para un llenado de esa duración. Después encuentra todos los instantes en los que podría empezar un único `fill` de 20 segundos, y comprueba los dos extremos de tu respuesta con `kairos validate`.

Ejercicio 12.3.2. El jardinero quiere que queden siempre 2 litros en el depósito. Cambia el dominio del invernadero, vuelve a ejecutar `greenhouse-p01.pddl` y explica cómo se ha desplazado el plan.

Ejercicio 12.3.3. En el plan de la lámpara de la Sección 12.3, la carga empieza en $t = 5$. ¿Sería un plan una carga de 10 segundos que empezara en $t = 0$? ¿Y una de 6 segundos que empezara en $t = 0$? Responde con la gráfica de la batería y después valida.

Ejercicios de la Sección 12.4 — El generador: cuándo empezar

Ejercicio 12.4.1. El depósito del generador de `generator-p01.pddl` empieza ahora con 90 litros en lugar de 95. Calcula la ventana de inicios válidos para el vertido y di si el bidón se podría verter antes del funcionamiento.

Ejercicio 12.4.2. En el plan de `generator-p02.pddl`, el segundo bidón se vierte en 95,001. Da otro plan válido en el que no haya ningún vertido en curso cuando el generador se para, y valídalo.

Ejercicio 12.4.3. Una norma de seguridad prohíbe verter combustible en un generador en marcha. Escribe la norma en el dominio. ¿Cuál de

los problemas `p01` y `p02` (el segundo con 60 litros en el depósito) sigue teniendo plan, y por qué?

Ejercicios de la Sección 12.5 — El horno: cuándo parar

Ejercicio 12.5.1. En `oven-p01.pddl`, ¿qué duraciones de `heat` llevan a un plan válido si el pan entra cuando se para la resistencia? ¿Y si el pan puede entrar con la resistencia todavía encendida, como en `oven-overlap.plan`? Valida ese plan y explica por qué se acepta.

Ejercicio 12.5.2. Dale al horno una forma de enfriarse: una acción `cool`, con la puerta abierta y el horno vacío, que baje la temperatura a `(cool-rate)` grados por segundo durante un tiempo elegido. Resuelve un problema en el que el horno esté a 150 grados y un merengue se hornee entre 100 y 120.

Ejercicios de la Sección 12.6 — Lo que debe encontrar el planificador

Ejercicio 12.6.1. Ejecuta el primer problema del invernadero y el segundo del mensajero con `-v` y encuentra en la salida cuál de las dos formas de la caja de concepto ha usado Kairos en cada uno. Explica la diferencia a partir de las condiciones que leen el nivel y la batería.

Ejercicio 12.6.2. Un colega elimina la cota inferior de `charge` y deja solo la superior. ¿Qué planes esperas para el problema `p02` del mensajero? Explica qué hacía la cota inferior, como afirmación sobre el mundo y como ayuda para la búsqueda.

Ejercicios de la Sección 12.7 — El mundo del mensajero

Ejercicio 12.7.1. El hospital quiere que los robots mantengan una reserva: la batería no debe bajar nunca de 10 mientras se mueven. Cambia el dominio y vuelve a ejecutar `p02` y `p05`. ¿Qué plan cambia y por qué?

Ejercicio 12.7.2. Dibuja, como en la Figura 12.7, la batería del robot durante el último plan de `courier-battery-p03.pddl`. ¿En qué instante está la batería más baja? ¿Cuántos segundos de la primera carga se podrían ahorrar?

Ejercicio 12.7.3. En nuestro modelo, recoger y dejar no gastan batería. Haz que gasten 0,5 unidades por segundo, como efectos continuos. ¿Necesita `pick` una condición `over all` sobre la batería? Justifica tu respuesta con la pregunta de la caja de método.

Ejercicios de la Sección 12.8 — El método MODEL: cambio continuo

Ejercicio 12.8.1. Para cada situación, responde a las dos preguntas (¿cambia la magnitud mientras se ejecuta una acción?, ¿la lee o la cambia alguien mientras tanto?) y elige entre un efecto discreto y uno continuo: (a) un camión quema combustible en cada viaje y solo reposta cuando está aparcado; (b) una impresora gasta tinta mientras imprime, y un cartucho solo se puede cambiar cuando se detiene; (c) un calentador calienta el agua mientras alguien se ducha con ella; (d) un dron descarga su batería mientras vuela y mientras su cámara graba, y las dos cosas pueden ocurrir a la vez.

Ejercicio 12.8.2. Aplica el paso L al mensajero de este capítulo: escribe la versión discreta prudente de `move` y `charge`, con `?duration` en el efecto de la carga, y compárala con la versión continua según los cinco criterios. ¿Cuál le darías a un planificador que no admite `:continuous-effects`?

Predicados derivados y un mundo con calendario

13.

En todos los modelos vistos hasta ahora, un hecho es cierto por uno de dos motivos: lo dice el estado inicial, o una acción lo hizo cierto y ninguna lo ha hecho falso desde entonces. Hay dos situaciones del hospital que no encajan. La primera: la planta tiene puertas, y un robot que está en la base de carga quiere saber si puede llegar a la UCI. La respuesta no está escrita en ninguna parte y ninguna acción la fija. Se deduce de qué puertas están abiertas, y cambia, para muchos pares de habitaciones a la vez, cada vez que alguien abre o cierra una puerta. La segunda: la farmacia abre a las ocho y cierra a las dos. Ningún robot la abre. Abre porque es la hora, y un mensajero que llega antes de tiempo solo puede esperar. Nuestros modelos no tienen forma de decir «esto es cierto porque esas otras cosas son ciertas», ni de decir «esto será cierto en ese instante, haga lo que haga cualquiera».

PDDL 2.2 añadió una construcción para cada caso. Un *predicado derivado* es un hecho definido por una regla sobre otros hechos: el planificador calcula si es cierto en cada estado, y ninguna acción lo toca. Un *literal inicial temporizado* es un hecho que el problema programa para que se haga cierto o falso en un instante dado: un calendario que el mundo sigue por sí solo. Las dos construcciones no tienen nada más en común, y por eso comparten capítulo: son las dos maneras en que un hecho puede cambiar sin que lo cambie una acción. Una es consecuencia del estado; la otra, del reloj.

La primera mitad del capítulo trata de reglas. Empezamos con un hecho que el mundo del mensajero mantiene a mano desde el Capítulo 2, `(free ?r)`, y lo convertimos en un predicado derivado; después vienen reglas que se llaman a sí mismas, para las habitaciones alcanzables a través de puertas abiertas, para torres de bloques y para circuitos eléctricos; y después, qué hace un planificador con las reglas y cuánto cuesta. La segunda mitad trata del calendario: horarios de apertura, plazos y ventanas de visibilidad, en el mensajero y en un satélite, y números que cambian en instantes conocidos. La sección del método añade una pregunta al paso O y otra al paso M, y compara un hecho derivado con el mismo hecho mantenido mediante efectos.

13.1 Hechos que se siguen de otros hechos 298
13.2 Predicados derivados . 299
Dónde puede aparecer un predicado derivado . 300
13.3 Reglas que se llaman a sí mismas 301
Dos recursiones más: torres y circuitos 304
Reglas y negación 306
13.4 Qué hace un planificador con las reglas 306
13.5 Un mundo con calendario: literales iniciales temporizados 307
13.6 Ventanas y plazos . . . 309
Plazos 310
Un satélite y sus ventanas 311
13.7 Números en el calendario 313
13.8 El método MODEL: consecuencias y calendario 314
Alternativas comparadas 315
Un plazo: literal temporizado o restricción 316
13.9 Errores comunes 316
13.10 Resumen 317
13.11 Lecturas complementarias 318
13.12 Ejercicios 318

13.1. Hechos que se siguen de otros hechos

Miremos el dominio del mensajero con tipos del Capítulo 4 con los ojos de la tabla de estado. Tiene dos predicados sobre lo que lleva un robot: `(carrying ?r ?i)` y `(free ?r)`. No son independientes. Un robot está libre exactamente cuando no lleva nada; si conocemos todos los hechos `carrying`, conocemos todos los hechos `free`. El dominio guarda los dos porque `pick` necesita preguntar «¿este robot no lleva nada?», y en STRIPS una precondición solo puede pedir un hecho. Así que `free` se almacena, y las acciones lo mantienen acompasado con `carrying` a mano: `pick` lo borra, `drop` lo añade, y el estado inicial debe declararlo para cada robot que empieza con las manos vacías.

Funciona, y el Capítulo 7 puso nombre a su riesgo: un predicado redundante que se desacompasa. Un problema cuyo estado inicial olvida `(free`

`r1`) tiene un robot que nunca puede coger nada. Una acción nueva que le quita un artículo a un robot (por ejemplo, una enfermera se lo descarga) y olvida añadir `free` deja un robot que no lleva nada y no está libre. Nada avisa de ninguno de los dos errores: los dos estados son conjuntos de hechos perfectamente válidos.

Con `free`, la contabilidad es ligera: un hecho por acción. Para otros hechos no hay manera. Pensemos en «la UCI es alcanzable desde la base de carga». Abrir una puerta puede hacerlo cierto para decenas de pares de habitaciones, o para ninguno, según cómo estén todas las demás puertas. Un efecto de `open-door` que mantuviera al día un hecho así tendría que contener todo el cálculo de caminos en un grafo.

La distinción que necesitamos es esta. Un **predicado básico** contiene hechos que se declaran: en el estado inicial y en los efectos de las acciones. Un **predicado derivado** contiene hechos que se calculan: una regla dice cuándo son ciertos, en función de otros hechos, y son ciertos en un estado exactamente cuando la regla lo dice.

Algunos hechos no los decide ninguna acción: son consecuencia de otros hechos. Mantenerlos a mano en los efectos es frágil cuando es fácil, e imposible cuando es difícil.

13.2. Predicados derivados

El Listado 13.1 es el dominio del mensajero con tipos, con `free` convertido en un predicado derivado.

```
1  ;; The typed courier world where (free ?r) is not kept by the
2  ;; actions any more: a rule derives it from (carrying ?r ?i).
3  (define (domain courier-free)
4    (:requirements :strips :typing :negative-preconditions
5                   :existential-preconditions :derived-predicates)
6
7    (:types robot location item)
8
9    (:predicates
10     (connected ?from ?to - location)
11     (robot-at ?r - robot ?l - location)
12     (item-at ?i - item ?l - location)
13     (carrying ?r - robot ?i - item)
14     (free ?r - robot))            ; derived: see the rule below
15
16   ; A robot is free when there is no item that it carries.
17   (:derived (free ?r - robot)
18     (not (exists (?i - item) (carrying ?r ?i))))
19
20   (:action move
21     :parameters (?r - robot ?from ?to - location)
22     :precondition (and (robot-at ?r ?from) (connected ?from ?to))
23     :effect (and (not (robot-at ?r ?from))
24                  (robot-at ?r ?to)))
25
26   ; pick and drop no longer mention (free ?r) in their effects
27   (:action pick
28     :parameters (?r - robot ?i - item ?l - location)
29     :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
30     :effect (and (not (item-at ?i ?l))
31                  (carrying ?r ?i)))
32
33   (:action drop
34     :parameters (?r - robot ?i - item ?l - location)
35     :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
36     :effect (and (not (carrying ?r ?i))
37                  (item-at ?i ?l))))
```

Lo nuevo está en las líneas 17 y 18:

Listado 13.1: El dominio del mensajero con `(free ?r)` definido por una regla.

```
  (:derived (free ?r - robot)
    (not (exists (?i - item) (carrying ?r ?i))))
```

Una definición `:derived`, una **regla**, tiene dos partes. La primera es la **cabeza**: el predicado con sus variables, con sus tipos, como en una lista de parámetros. La segunda es el **cuerpo**: una condición, escrita con todo lo que el Capítulo 5 enseñó para las precondiciones (`and`, `or`, `not`, `exists`, `forall`, `imply`), cuyas variables libres son las de la cabeza. La regla se lee de derecha a izquierda: *si* no hay ningún artículo que lleve `?r`, *entonces* se cumple `(free ?r)`. Y como el mundo es cerrado, como siempre, también se lee en el otro sentido: si el cuerpo es falso, `(free ?r)` es falso. El dominio declara `:derived-predicates`, y el predicado sigue figurando en `:predicates`, como cualquier otro.

Alrededor de la regla han desaparecido tres cosas. `pick` ya no borra `free`, `drop` ya no lo añade, y el estado inicial de `courier-free-p01.pddl` no lo menciona. La precondición de `pick` no ha cambiado: pide `(free ?r)` como antes, y el objetivo del problema también pide `(free r1)`.

```
$ kairos courier-free-domain.pddl courier-free-p01.pddl
; cost = 15
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds2 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds2 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Dos artículos, de uno en uno. Después del primer `pick` el robot lleva `meds2`, el cuerpo de la regla es falso y el segundo `pick` no es aplicable hasta el `drop`: la regla hace lo que hacía el hecho mantenido a mano. El plan es el mismo que da el dominio del Capítulo 4 para el mismo problema.

Dónde puede aparecer un predicado derivado

Un predicado derivado se puede *leer* en cualquier sitio donde se lea uno básico: en las precondiciones, en las condiciones de las acciones durativas, en la condición de un `when`, en los objetivos y en los cuerpos de otras reglas. Nunca se puede *escribir*. No puede aparecer en un efecto:

```
$ kairos check courier-free-wrong-domain.pddl
courier-free-wrong-domain.pddl:22:18: error: in action 'drop':
     derived predicate 'free' cannot be changed by an effect [
     semantic.derived_in_effect]
 22 |                  (free ?r))))     ; not allowed: free is
     derived
    |                  ^
invalid
```

Tampoco puede declararlo el estado inicial. Un problema que declare `(free r1)` para este dominio se rechaza con `derived predicate 'free' cannot be initialized`. Las dos prohibiciones responden a la misma idea. Si una acción o un problema pudieran fijar el hecho, habría estados en los que el hecho y su regla no coincidirían, y la regla dejaría de ser una definición.

Un predicado derivado se lee como cualquier otro y nunca se escribe: ni en los efectos ni en el estado inicial. Su valor en cada estado es el que digan sus reglas.

Un predicado puede tener varias reglas. Entonces se cumple cuando se cumple el cuerpo de *cualquiera* de ellas: varias reglas equivalen a un `or`. Un hospital en el que se puede entrar en una habitación con tarjeta de acceso o con escolta podría decirlo en dos líneas, una por motivo, en lugar de con una sola regla con una disyunción.

Axiomas: fuera de PDDL y de vuelta

Reglas como estas son mucho más antiguas que PDDL 2.2. La primera definición de PDDL, en 1998 McDermott et al., 1998, ya las tenía, con el nombre de *axiomas*. Pocos planificadores las admitían y las versiones posteriores del lenguaje las dejaron fuera. Stefan Edelkamp y Jörg Hoffmann las recuperaron, como predicados derivados, en PDDL 2.2, el lenguaje de la competición de planificación de 2004 Edelkamp y Hoffmann, 2004; la otra novedad de esa versión, los literales iniciales temporizados, ocupa la segunda mitad de este capítulo. Poco después, Sylvie Thiébaux, Jörg Hoffmann y Bernhard Nebel defendieron que los axiomas merecen su sitio: demostraron que, en general, un dominio con axiomas no puede reescribirse sin ellos salvo que el dominio o los planes se hagan exponencialmente más grandes Thiébaux et al., 2005.

13.3. Reglas que se llaman a sí mismas

La regla de `free` podría haberse escrito dentro de `pick`; el predicado derivado solo le puso nombre a la condición. La siguiente regla no puede escribirse dentro de ninguna precondición, porque no tiene un tamaño fijo.

La planta tiene ahora tres puertas: `d-pharmacy` entre el vestíbulo y la farmacia, `d-corridor` entre el vestíbulo y el pasillo, y `d-icu` entre la sala B y la UCI. Usamos los predicados de puertas del Capítulo 5, `(joins ?d ?a ?b)` y `(open ?d)`. Y cambiamos el nivel de detalle con el que se mueve el robot. Un robot mensajero real tiene un sistema de navegación que encuentra por sí mismo el camino por el edificio; el planificador de tareas no necesita elegir cada paso, solo saber *si* el robot puede llegar de una habitación a otra. Así que `move`, entre vecinas, se convierte en `go`, entre dos habitaciones cualesquiera tales que la segunda es alcanzable desde la primera (Listado 13.2).

```
  (:predicates
    (connected ?from ?to - location)
    (joins ?d - door ?a ?b - location)  ; a door between two rooms
    (open ?d - door)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (passable ?a ?b - location)     ; derived
    (reachable ?a ?b - location))   ; derived

  ; One step: the rooms are connected and every door between
  ; them is open (no door at all is fine too).
```

```
(:derived (passable ?a ?b - location)
  (and (connected ?a ?b)
       (forall (?d - door)
         (imply (or (joins ?d ?a ?b) (joins ?d ?b ?a))
                (open ?d)))))

; Any number of steps: one step, or one step and then the rest.
(:derived (reachable ?a ?b - location)
  (or (passable ?a ?b)
      (exists (?c - location)
        (and (passable ?a ?c) (reachable ?c ?b)))))

; The whole way in one action: navigation is not planned here.
(:action go
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (reachable ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

Listado 13.2: Alcanzabilidad a través de puertas abiertas: los predicados, las dos reglas y la acción `go` de `courier-reach-domain.pddl`.

Hay dos reglas, una construida sobre la otra.

`passable` es un paso: dos habitaciones conectadas tales que todas las puertas que hay entre ellas están abiertas. El `forall` con `imply` es el «todo ... que ...» del Capítulo 5; si no hay ninguna puerta entre dos habitaciones, se cumple para todas las cero que hay, y las habitaciones son transitables.

`reachable` es cualquier número de pasos, y su cuerpo menciona al propio `reachable`: `?b` es alcanzable desde `?a` si un paso lleva hasta allí, o si un paso lleva a alguna habitación `?c` desde la que `?b` es alcanzable. Esta es una **regla recursiva**. Es la definición de camino que se escribiría en matemáticas, y no tiene equivalente usando solo `exists`: una fórmula sin recursión puede hablar de caminos de dos pasos, o de tres, o de cualquier número fijo, pero no de caminos de la longitud que el mapa necesite en cada caso.

¿Cómo puede definir algo una regla que se refiere a sí misma? Construyendo desde cero. Empezamos sin ningún hecho `reachable`. Aplicamos la regla a todo lo que pueda derivar: eso da los pares unidos por un paso transitable. La aplicamos otra vez, ahora con esos hechos disponibles: salen los pares que están a dos pasos. Seguimos hasta que una ronda no añade nada nuevo. Lo que se ha derivado hasta entonces es cierto; todo lo demás es falso. La Figura 13.1 muestra las rondas en el mapa con la puerta de la UCI cerrada.

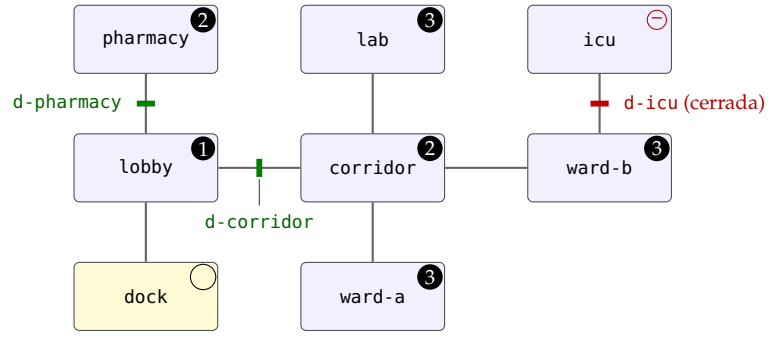


Figura 13.1.: Derivación de `(reachable dock ?b)` con dos puertas abiertas (en verde) y una cerrada (en rojo). El número de cada habitación es la ronda en la que se deriva el hecho para esa habitación: un paso transitable, después dos, después tres. Una cuarta ronda no añade nada, así que `(reachable dock icu)` es falso.

Tres problemas ponen a prueba el dominio. En `courier-reach-p01.pddl` todas las puertas están abiertas y los medicamentos van de la farmacia a la UCI:

```
$ kairos courier-reach-domain.pddl courier-reach-p01.pddl
; cost = 4
(go r1 dock pharmacy)
(pick r1 meds1 pharmacy)
(go r1 pharmacy icu)
(drop r1 meds1 icu)
```

Cuatro acciones donde el mensajero que avanza paso a paso necesita ocho. El segundo `go` recorre cuatro habitaciones y tres puertas en una sola acción, porque se ha derivado `(reachable pharmacy icu)`.

En `courier-reach-p02.pddl` las tres puertas empiezan cerradas:

```
$ kairos courier-reach-domain.pddl courier-reach-p02.pddl
; cost = 9
(go r1 dock lobby)
(open-door r1 d-corridor lobby corridor)
(open-door r1 d-pharmacy lobby pharmacy)
(go r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(go r1 pharmacy ward-b)
(open-door r1 d-icu ward-b icu)
(go r1 ward-b icu)
(drop r1 meds1 icu)
```

Este plan muestra lo que es un hecho derivado. Ninguna de sus acciones tiene `reachable` en sus efectos y, sin embargo, cada `open-door` cambia qué acciones `go` son aplicables a continuación. Después de las dos primeras, el robot puede ir de la farmacia directamente a la sala B; no puede ir más allá, por la tercera puerta, así que se detiene allí, la abre y sigue. El planificador trata las puertas como el medio y la alcanzabilidad como su consecuencia, que es como pensamos en ellas nosotros.

El tercer problema usa el predicado derivado en un objetivo, bajo una negación. La UCI debe recibir sus medicamentos y quedar después aislada otra vez del vestíbulo, con el robot de vuelta en la base de carga (`courier-reach-p03.pddl`):

```
  (:goal (and (item-at meds1 icu)
              (robot-at r1 dock)
              (not (reachable lobby icu)))))
```

```
$ kairos courier-reach-domain.pddl courier-reach-p03.pddl
; cost = 9
(go r1 dock pharmacy)
(pick r1 meds1 pharmacy)
(go r1 pharmacy dock)
(go r1 dock icu)
(drop r1 meds1 icu)
(go r1 icu dock)
(go r1 dock lobby)
(close-door r1 d-corridor lobby corridor)
(go r1 lobby dock)
; better plan 2
; cost = 7
(go r1 dock pharmacy)
(pick r1 meds1 pharmacy)
(go r1 pharmacy icu)
(drop r1 meds1 icu)
(go r1 icu ward-b)
(close-door r1 d-icu ward-b icu)
(go r1 ward-b dock)
```

El objetivo no dice qué puerta cerrar, y los dos planes cierran puertas distintas. El primero cierra la puerta del pasillo, desde el lado del vestíbulo, lo que separa el vestíbulo de media planta; el segundo cierra la puerta de la UCI al salir. Los dos hacen falso `(reachable lobby icu)`. Un objetivo sobre un predicado derivado dice *qué* debe cumplirse y deja el *cómo* al planificador.

Una regla recursiva define hechos de profundidad ilimitada, como los caminos de un grafo. Sus hechos son los que deriva la aplicación repetida de la regla, empezando desde cero.

Dos recursiones más: torres y circuitos

La misma forma de regla aparece siempre que una relación es la repetición de otra más simple. En el mundo de bloques, `(on ?a ?b)` dice que un bloque está directamente sobre otro. «En algún lugar por encima» es su repetición (Listado 13.3).

```
;; Blocks world with two derived predicates: (clear ?b) is the
;; absence of a block on ?b, and (above ?a ?b) looks through a
;; whole tower.
(define (domain blocks-above)
  (:requirements :strips :typing :negative-preconditions
                 :disjunctive-preconditions :equality
                 :existential-preconditions :derived-predicates)

  (:types block)

  (:predicates
    (on ?a ?b - block)        ; ?a lies directly on ?b
    (on-table ?a - block)
    (clear ?a - block)        ; derived
    (above ?a ?b - block))    ; derived

  (:derived (clear ?a - block)
    (not (exists (?b - block) (on ?b ?a))))

  ; ?a is above ?b if it lies on ?b, or on a block above ?b.
  (:derived (above ?a ?b - block)
    (or (on ?a ?b)
        (exists (?c - block) (and (on ?a ?c) (above ?c ?b)))))

  (:action move-to-block
    :parameters (?a ?from ?to - block)
    :precondition (and (on ?a ?from) (clear ?a) (clear ?to)
                       (not (= ?a ?to)))
    :effect (and (not (on ?a ?from)) (on ?a ?to)))

  (:action move-to-table
    :parameters (?a ?from - block)
    :precondition (and (on ?a ?from) (clear ?a))
    :effect (and (not (on ?a ?from)) (on-table ?a)))

  (:action move-from-table
    :parameters (?a ?to - block)
    :precondition (and (on-table ?a) (clear ?a) (clear ?to)
                       (not (= ?a ?to)))
    :effect (and (not (on-table ?a)) (on ?a ?to))))
```

`clear`, que todos los mundos de bloques de este libro mantenían a mano en cuatro efectos, es aquí la ausencia de un bloque encima, el mismo patrón que `free`. `above` sigue el patrón de `reachable`. Con ellos, el objetivo puede decir lo que importa y nada más. En `blocks-above-p01.pddl` la torre es `d` sobre `c` sobre `b` sobre `a`, y el objetivo es `(and (above a d) (clear a))`: `a` en algún lugar por encima de `d`, sin nada encima. No dice dónde acaban `b` y `c`.

Listado 13.3: El mundo de bloques con `clear` y `above` como predicados derivados.

```
$ kairos blocks-above-domain.pddl blocks-above-p01.pddl
; cost = 4
(move-to-table d c)
(move-to-table c b)
(move-to-table b a)
(move-from-table a d)
```

El planificador eligió la disposición más barata que satisface el objetivo: `a` directamente sobre `d`, y los otros dos en la mesa. Un objetivo con `on` nos habría obligado a elegir nosotros una disposición. El problema `blocks-above-p02.pddl` pide `a` por encima de `b` y `b` por encima de `a`; Kairos explora todas las disposiciones de los tres bloques y responde `unsolvable`.

El tercer dominio no tiene robot y casi no tiene acciones. Una instalación eléctrica tiene una fuente, puntos unidos por cables e interruptores en algunos de los cables. Un punto tiene corriente si es la fuente o si está enlazado con un punto que tiene corriente (Listado 13.4).

```
;; An electrical installation: current flows from the source
;; through wires and closed switches. Which points have power is
;; never stated: it follows from the switches.
(define (domain circuit)
  (:requirements :strips :typing :negative-preconditions
                 :disjunctive-preconditions
                 :existential-preconditions :derived-predicates)

  (:types point switch)

  (:predicates
    (source ?p - point)                 ; the mains come in here
    (wire ?a ?b - point)                ; a plain wire
    (switched ?s - switch ?a ?b - point) ; a wire with a switch
    (closed ?s - switch)                ; a closed switch conducts
    (linked ?a ?b - point)              ; derived
    (powered ?p - point))               ; derived

  ; Current can pass between two points, in either direction.
  (:derived (linked ?a ?b - point)
    (or (wire ?a ?b) (wire ?b ?a)
        (exists (?s - switch)
          (and (closed ?s)
               (or (switched ?s ?a ?b) (switched ?s ?b ?a))))))

  (:derived (powered ?p - point)
    (or (source ?p)
        (exists (?q - point) (and (powered ?q) (linked ?q ?p)))))

  (:action close-switch
    :parameters (?s - switch)
    :precondition (not (closed ?s))
    :effect (closed ?s))

  (:action open-switch
    :parameters (?s - switch)
    :precondition (closed ?s)
    :effect (not (closed ?s))))
```

Listado 13.4: Un circuito: qué puntos tienen corriente se deduce de los interruptores.

Las únicas acciones cierran y abren interruptores; todo lo interesante se deriva. En `circuit-p01.pddl` (Figura 13.2) el horno debe tener corriente mientras la cocina, donde trabaja un electricista, no tiene. Al principio, los dos interruptores de la línea de la cocina están cerrados y los demás, abiertos.

```
$ kairos circuit-domain.pddl circuit-p01.pddl
; cost = 5
(open-switch s2)
(open-switch s4)
(close-switch s3)
(close-switch s1)
(close-switch s5)
```

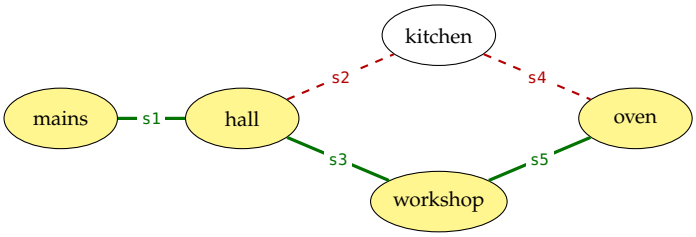


Figura 13.2.: El circuito de `circuit-p01` después del plan. Los interruptores cerrados están en verde continuo, y los abiertos, en rojo discontinuo; los puntos con corriente están sombreados. La corriente llega al horno a través del taller. Fíjate en `s4`: hubo que abrirlo, porque un interruptor cerrado conduce en los dos sentidos y el horno alimentaría la cocina.

El plan abre los dos interruptores de la cocina, no solo el del lado del recibidor. La regla de `linked` es simétrica, así que un horno con corriente detrás de un `s4` cerrado daría corriente a la cocina por detrás. El planificador encontró un hecho sobre los circuitos que un electricista con prisas olvida, y lo encontró en las reglas, no en ninguna acción.

Reglas y negación

La regla de `free` tiene una negación en su cuerpo, y también podría tenerla una regla nuestra que usara negado otro predicado derivado: «una habitación está aislada si no es alcanzable desde el vestíbulo». Lo que una regla no puede hacer es depender negativamente de *sí misma*, directamente o a través de otras reglas. El procedimiento que construye los hechos ronda a ronda solo tiene sentido si un hecho, una vez derivado, sigue derivado; una regla como

```
(:derived (odd ?l - lamp) (not (odd ?l)))
```

derivaría un hecho porque está ausente, y con ello haría falso su propio motivo. Kairos la rechaza: `derived predicate 'odd' depends negatively on itself`. Los conjuntos de reglas sin ciclos de este tipo se llaman **estratificados**: los predicados se pueden repartir en capas, cada capa usa libremente las inferiores y a sí misma solo en positivo, y las capas se calculan una tras otra. Todas las reglas de este capítulo están estratificadas en dos capas como mucho.

PDDL 2.2 es más estricto: no permite en absoluto un predicado derivado bajo una negación en el cuerpo de una regla. Kairos y los planificadores construidos sobre Fast Downward aceptan cualquier conjunto estratificado de reglas.

13.4. Qué hace un planificador con las reglas

Para el planificador, un estado sigue siendo un conjunto de hechos básicos. Los hechos derivados no se guardan en él: cada vez que la búsqueda llega a un estado nuevo, las reglas se ejecutan sobre sus hechos básicos hasta que no sale nada nuevo, y solo entonces se comprueban las precondiciones y el objetivo. La opción `-v` muestra el tamaño de ese trabajo para las puertas:

```
$ kairos courier-reach-domain.pddl courier-reach-p02.pddl -v
...
```

```
[   0.001s] [ground] grounded in 0.4 ms: 291 atoms, 0 numeric
     fluents, 92 actions, 0 durative actions, 518 axioms (1031
     Datalog facts, 28 rules, 0 instances pruned)
[   0.001s] [sas] SAS+ in 0.3 ms: 276 variables (2 from 3 mutex
     groups, 4 binary), 3 invariants from 15 candidates, 1 axiom
     layers
...
```

Nuestras dos reglas se convirtieron en 518 reglas instanciadas (el planificador las llama axiomas), construidas a partir de las instancias de las reglas para habitaciones y puertas concretas. Las cifras crecen deprisa. Con ocho habitaciones hay 64 pares `(reachable ?a ?b)`, y la regla recursiva puede derivar cada uno a través de cualquiera de ocho habitaciones intermedias. Una regla con tres variables sobre n objetos tiene del orden de n^3 instancias, y estas se evalúan en cada estado que visita la búsqueda. Los predicados derivados no salen gratis: trasladan trabajo del modelador al planificador, y en un mapa grande el modelador debería saber cuánto.

Hay un segundo coste, menos visible. Las heurísticas del Capítulo 3 estiman la distancia al objetivo mirando lo que añaden y borran las acciones. Un hecho derivado no lo añade ni lo borra ninguna acción, así que la heurística tiene que razonar a través de las reglas para ver que abrir una puerta acerca la UCI. Los planificadores difieren en lo bien que lo hacen. En los tres problemas de las puertas, la búsqueda expandió como mucho unos cientos de estados, así que aquí lo hizo bien; un modelo en el que todo lo interesante es derivado, como el circuito, le da a la heurística menos a lo que agarrarse.

¿Por qué no eliminar las reglas compilándolas?

Si una regla es solo un nombre para una condición, un planificador podría sustituir el nombre por la condición y olvidarse de la regla. Para `free` funciona: es lo que habríamos hecho al escribir el `exists` dentro de `pick`. Para una regla recursiva no hay ninguna condición que sustituir. La alternativa es añadir acciones que deriven los hechos paso a paso y obligar al planificador a ejecutarlas después de cada acción real; entonces los planes crecen con el número de pasos de derivación. El resultado de Thiébaux, Hoffmann y Nebel Thiébaux et al., 2005 dice que esto no es un accidente de una compilación concreta: cualquier forma de eliminar los axiomas hace el dominio o los planes exponencialmente más grandes en el peor caso. Por eso, los planificadores que admiten bien los predicados derivados, como Fast Downward Helmert, 2006, los conservan como reglas, organizadas en capas, y los evalúan en cada estado.

El tercer coste es el de toda característica que va más allá de STRIPS: menos planificadores. Un modelo que usa un predicado derivado solo como nombre, como con `free`, se reescribe en un minuto para un planificador que no los admita. Un modelo construido sobre una regla recursiva, no, y elegirlo es elegir los planificadores que podrán ejecutarlo.

Portabilidad. En la prueba que mantiene la documentación de Kairos (una regla recursiva, diez segundos por planificador), Kairos, LPG-td y LAMA devolvieron un plan válido; OPTIC, POPF, TFD, ENHSP y Metric-FF, no. Los predicados derivados son raros entre los planificadores temporales y numéricos.

Las reglas se instancian como las acciones y se evalúan en cada estado. Una regla con tres variables sobre n objetos cuesta del orden de n^3 reglas instanciadas.

13.5. Un mundo con calendario: literales iniciales temporizados

Dejamos las reglas y pasamos al reloj. El mensajero temporal del Capítulo 10 vive en un hospital donde no ocurre nada si no lo hace un robot.

Ahora la farmacia tiene horario de apertura. Lo que necesitamos decir es sencillo: en el instante 30 abre la farmacia. Nadie la abre, ninguna acción lo causa, ninguna condición tiene que cumplirse. Es un hecho sobre el futuro que se conoce hoy.

PDDL 2.2 lo escribe en el estado inicial del problema:

```
   (at 30 (is-open pharmacy))
```

Esto es un **literal inicial temporizado**: un instante y un literal. En ese instante el literal se hace cierto, haga lo que haga el plan. Un literal negativo, `(at 50 (not (is-open pharmacy)))`, se hace falso en su instante. Los instantes se cuentan desde el inicio del plan, en la misma unidad que las duraciones. El requisito es `:timed-initial-literals`, y no hay más sintaxis que esa. `(is-open ?l)` es un predicado corriente que el dominio declara y que las acciones usan en sus condiciones; lo único especial que tiene es que ninguna acción del dominio lo tiene en un efecto. Lo cambia el mundo, según el horario.

El Listado 13.5 muestra las dos acciones del dominio que miran el calendario. El dominio es el mensajero temporal con `(idle ?r)` y duraciones calculadas a partir de las distancias, y `drop` se ha convertido en `deliver`, que entrega un artículo en su destino.

```
38  ; Nobody hands over an item in a closed room: the room must be
39  ; open during the whole pick.
40  (:durative-action pick
41    :parameters (?r - robot ?i - item ?l - location)
42    :duration (= ?duration 2)
43    :condition (and
44      (at start (idle ?r))
45      (at start (item-at ?i ?l))
46      (at start (free ?r))
47      (over all (robot-at ?r ?l))
48      (over all (is-open ?l)))
49    :effect (and
50      (at start (not (idle ?r)))
51      (at start (not (item-at ?i ?l)))
52      (at start (not (free ?r)))
53      (at end (carrying ?r ?i))
54      (at end (idle ?r))))
55
56  ; Handing the item over at its destination, before its deadline.
57  (:durative-action deliver
58    :parameters (?r - robot ?i - item ?l - location)
59    :duration (= ?duration 2)
60    :condition (and
61      (at start (idle ?r))
62      (at start (carrying ?r ?i))
63      (at start (destination ?i ?l))
64      (over all (robot-at ?r ?l))
65      (over all (is-open ?l))
66      (at end (in-time ?i)))
67    :effect (and
68      (at start (not (idle ?r)))
69      (at start (not (carrying ?r ?i)))
70      (at end (item-at ?i ?l))
71      (at end (delivered ?i))
72      (at end (free ?r))
73      (at end (idle ?r)))))
```

Las dos piden que la habitación esté abierta `over all`: durante toda la acción, no solo cuando empieza. (La condición sobre `in-time` al final de `deliver` es para la sección siguiente.) Todas las habitaciones del problema `courier-hours-p01.pddl` están abiertas salvo la farmacia. No

Listado 13.5: Las acciones `pick` y `deliver` de `courier-hours-domain.pddl`: la habitación debe estar abierta durante toda la entrega.

figura entre los hechos iniciales, así que está cerrada, y recibe un literal temporizado:

```
    (is-open lobby) (is-open corridor) (is-open lab)
    (is-open ward-a) (is-open ward-b) (is-open icu) (is-open dock)
    (at 30 (is-open pharmacy))
```

```
$ kairos courier-hours-domain.pddl courier-hours-p01.pddl
; cost = 62.00399999999999
; makespan = 62.004
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
30.000: (pick r1 meds1 pharmacy) [2.000]
32.001: (move r1 pharmacy lobby) [6.000]
38.002: (move r1 lobby corridor) [10.000]
48.003: (move r1 corridor ward-a) [12.000]
60.004: (deliver r1 meds1 ward-a) [2.000]
```

Por primera vez en el libro, un plan tiene un tiempo muerto. El robot está en la farmacia en el instante 10,001 y no hace nada hasta el 30,000. No ha hecho falta ninguna acción «esperar»: en un plan temporal, esperar es la ausencia de acciones, y el planificador ha elegido como instante de inicio de `pick` aquel en que su condición se hace cierta. Sin el literal temporizado, este mismo problema duraba 42,006 segundos; el calendario cuesta 20 (Figura 13.3).

Un literal inicial temporizado hace un hecho cierto o falso en un instante fijo, haga lo que haga el agente. Los planes se adaptan colocando las acciones dentro de las ventanas, y esperando.

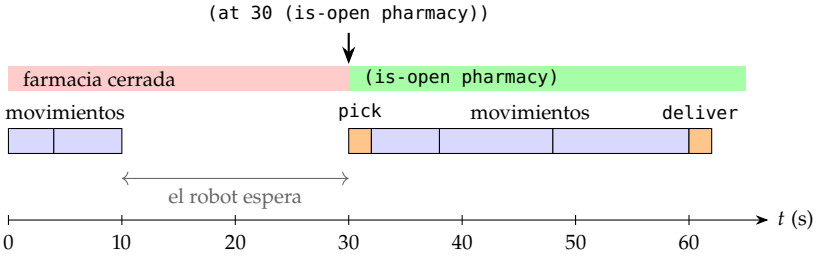


Figura 13.3.: El calendario (arriba) y el plan (abajo) de `courier-hours-p01`. La línea superior no la elige el planificador; la inferior, sí. Fíjate en el intervalo entre 10 y 30: el robot está parado en la puerta de la farmacia.

¿Cómo planifica esto Kairos? Con `-v` indica que la tarea la toma su motor M3, porque con literales iniciales temporizados «las acciones deben ocurrir dentro de ventanas temporales». Cada literal temporizado es un acontecimiento con un instante fijo, y el planificador añade a las restricciones temporales del plan (Capítulo 11) que las acciones que necesitan el hecho vayan después de él, y las que necesitan su ausencia, antes.

Portabilidad. En la misma prueba de antes, Kairos, OPTIC y POPF planificaron con literales iniciales temporizados; TFD, LPG-td, ENHSP, Metric-FF y LAMA, no. La característica necesita un planificador temporal, y no todos los planificadores temporales la tienen.

13.6. Ventanas y plazos

Dos literales temporizados sobre el mismo hecho, uno que lo hace cierto y otro posterior que lo hace falso, forman una **ventana**: un intervalo en el que algo es posible. En `courier-hours-p02.pddl` la farmacia solo está abierta de 30 a 50, y el laboratorio, al que debe ir una muestra de la sala A, cierra en el instante 90:

```
    (at 30 (is-open pharmacy))
    (at 50 (not (is-open pharmacy)))
    (at 90 (not (is-open lab)))
```

```
$ kairos courier-hours-domain.pddl courier-hours-p02.pddl
; cost = 86.00800000000001
; makespan = 86.008
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
30.000: (pick r1 meds1 pharmacy) [2.000]
32.001: (move r1 pharmacy lobby) [6.000]
38.002: (move r1 lobby corridor) [10.000]
48.003: (move r1 corridor ward-a) [12.000]
60.004: (deliver r1 meds1 ward-a) [2.000]
62.005: (pick r1 sample1 ward-a) [2.000]
64.006: (move r1 ward-a corridor) [12.000]
76.007: (move r1 corridor lab) [8.000]
84.008: (deliver r1 sample1 lab) [2.000]
```

Hay dos órdenes posibles para atender los dos artículos, y las ventanas solo dejan uno. Ir primero a por la muestra llevaría al robot a la farmacia en el instante 74, mucho después de que cerrara. Así que el robot espera a que abra la farmacia, como antes, y después tiene el tiempo justo: la muestra se entrega en el instante 86,008, menos de cuatro segundos antes de que cierre el laboratorio. Aquí importa la condición `over all`. Una entrega que empezara en 89 empezaría con el laboratorio abierto y terminaría con él cerrado, y el modelo la rechaza, como la rechazaría el hospital.

Plazos

Un **plazo** es media ventana: algo tiene que ocurrir antes de un instante dado. PDDL 2.2 no tiene ninguna construcción para ello, y no la necesita. El patrón es un hecho que significa «todavía se está a tiempo», cierto desde el principio y eliminado por un literal temporizado, y una acción que necesita el hecho cuando termina. En nuestro dominio, el hecho es `(in-time ?i)` y la acción es `deliver`, con su condición `(at end (in-time ?i))`. El problema `courier-hours-p03.pddl` tiene dos entregas: medicamentos para la sala A, que deben llegar antes del instante 50, y una comida para la sala B, que puede esperar:

```
    (in-time meds1) (in-time meal1)
    (at 50 (not (in-time meds1))))
```

```
$ kairos courier-hours-domain.pddl courier-hours-p03.pddl
; cost = 90.01200000000001
; makespan = 90.012
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
10.002: (pick r1 meds1 pharmacy) [2.000]
12.003: (move r1 pharmacy lobby) [6.000]
18.004: (move r1 lobby corridor) [10.000]
28.005: (move r1 corridor ward-a) [12.000]
40.006: (deliver r1 meds1 ward-a) [2.000]
42.007: (move r1 ward-a corridor) [12.000]
54.008: (move r1 corridor lobby) [10.000]
64.009: (pick r1 meal1 lobby) [2.000]
66.010: (move r1 lobby corridor) [10.000]
76.011: (move r1 corridor ward-b) [12.000]
88.012: (deliver r1 meal1 ward-b) [2.000]
```

Los medicamentos llegan en el instante 42,006, a tiempo. La comida estaba más cerca (está en el vestíbulo, que el robot cruza al principio), y un plan que la sirviera primero tiene exactamente la misma duración;

está en el fichero `courier-hours-mealfirst.plan`. Lo que distingue a los dos es el plazo:

```
$ kairos validate courier-hours-domain.pddl courier-hours-p03.pddl
     courier-hours-mealfirst.plan
courier-hours-mealfirst.plan:14: error: step 13 (deliver r1 meds1
     ward-a) at 88.012: at end condition false: (in-time meds1) is
     false [validate.precondition_failed]
 14 | 88.012: (deliver r1 meds1 ward-a) [2.000]
plan invalid
```

Un plazo que no se puede cumplir da un problema sin plan, y con literales temporizados Kairos a menudo puede demostrarlo. En el problema `courier-hours-p04.pddl` el plazo es 20, y los medicamentos no pueden estar en la sala A antes de 42:

```
$ kairos courier-hours-domain.pddl courier-hours-p04.pddl
note: no plan: the timed initial literals take away what the goal
     needs before it can be reached, even ignoring deletes and with
     the shortest durations: (delivered meds1) [orchestrator.
     deadlines]
kairos: unsolvable (exhausted)
```

Merece la pena leer la nota despacio. El planificador ha calculado el instante más temprano en que podría conseguirse `(delivered meds1)`, a la manera optimista de las heurísticas del Capítulo 3 (ignorando los borrados, con las duraciones más cortas), y ha visto que es posterior al instante en que desaparece `(in-time meds1)`. Si hasta el optimista llega tarde, nadie llega a tiempo, y eso es una demostración.

Un satélite y sus ventanas

Las ventanas son el pan de cada día de las misiones espaciales, donde el calendario lo escribe la mecánica orbital. Un satélite solo puede fotografiar una zona mientras la sobrevuela, y solo puede enviar sus imágenes mientras tiene a la vista una estación de tierra. Los dos intervalos se conocen con mucha antelación (Listado 13.6).

```
;; An observation satellite. A target can be photographed only
;; while the orbit passes over it, and images can be sent only
;; while a ground station is in sight: windows fixed by the orbit.
(define (domain satellite)
  (:requirements :typing :durative-actions
                 :timed-initial-literals)

  (:types target station)

  (:predicates
    (visible ?t - target)        ; the orbit passes over it now
    (in-sight ?s - station)      ; the station can be reached now
    (pointing ?t - target)
    (idle)                       ; one thing at a time
    (have-image ?t - target)
    (sent ?t - target))

  (:durative-action turn
    :parameters (?from ?to - target)
    :duration (= ?duration 5)
    :condition (and
      (at start (idle))
      (at start (pointing ?from)))
    :effect (and
```

```
      (at start (not (idle)))
      (at start (not (pointing ?from)))
      (at end (pointing ?to))
      (at end (idle))))

  ; The target must stay visible during the whole exposure.
  (:durative-action observe
    :parameters (?t - target)
    :duration (= ?duration 4)
    :condition (and
      (at start (idle))
      (over all (pointing ?t))
      (over all (visible ?t)))
    :effect (and
      (at start (not (idle)))
      (at end (have-image ?t))
      (at end (idle))))

  ; The station must stay in sight during the whole transmission.
  (:durative-action downlink
    :parameters (?t - target ?s - station)
    :duration (= ?duration 6)
    :condition (and
      (at start (idle))
      (at start (have-image ?t))
      (over all (in-sight ?s)))
    :effect (and
      (at start (not (idle)))
      (at end (sent ?t))
      (at end (idle)))))
```

Listado 13.6: Un satélite que observa y transmite dentro de ventanas.

Nada en el dominio menciona el tiempo, aparte de las duraciones. Las ventanas están en el problema, `satellite-p01.pddl`: el cráter es visible de 10 a 20, el volcán de 12 a 30, y la estación de Madrid está a la vista de 35 a 50. La cámara empieza apuntando al volcán.

```
   (at 10 (visible crater))
   (at 20 (not (visible crater)))
   (at 12 (visible volcano))
   (at 30 (not (visible volcano)))
   (at 35 (in-sight madrid))
   (at 50 (not (in-sight madrid))))
```

```
$ kairos satellite-domain.pddl satellite-p01.pddl
; cost = 47.001
; makespan = 47.001
0.000: (turn volcano crater) [5.000]
10.000: (observe crater) [4.000]
14.001: (turn crater volcano) [5.000]
19.002: (observe volcano) [4.000]
35.000: (downlink volcano madrid) [6.000]
41.001: (downlink crater madrid) [6.000]
```

El satélite hace primero lo que su orientación inicial no sugiere: deja el volcán y gira hacia el cráter, cuya ventana se cierra antes, y está listo antes de que se abra la ventana (Figura 13.4). Después vuelve a girar hacia el volcán, y las dos imágenes esperan en memoria a Madrid. Las dos transmisiones ocupan algo más de 12 de los 15 segundos del pase.

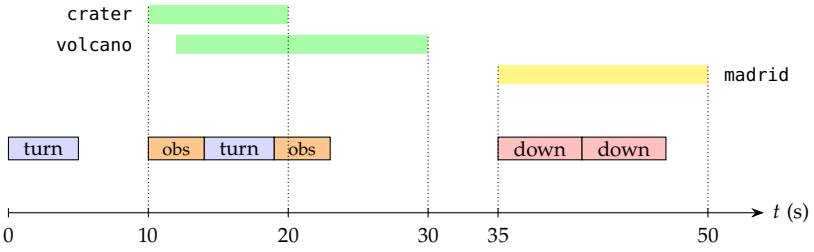


Figura 13.4.: Las ventanas de `satellite-p01` (arriba: cuándo es visible cada zona y cuándo está a la vista la estación) y el plan (abajo; *obs* es `observe` y *down* es `downlink`). Cada observación cae dentro de la ventana de su zona, y las dos transmisiones, dentro del pase de la estación.

En `satellite-p02.pddl` el cráter solo es visible durante tres segundos, y una exposición dura cuatro. Kairos responde `unsolvable`, con la misma nota que para el plazo: la ventana retira `(visible crater)` antes de que pueda alcanzarse `(sent crater)`.

13.7. Números en el calendario

Lo que el calendario hace con los hechos también puede hacerlo con los números. Durante el horario de visitas, los pasillos del hospital se llenan de gente y el robot tiene que ir despacio. Eso es un número, la velocidad, que cambia en instantes conocidos:

```
    (= (speed r1) 1)
    (at 20 (= (speed r1) 0.5))    ; a timed initial fluent
    (at 60 (= (speed r1) 1))
```

Un **fluente inicial temporizado** asigna un valor a un fluente numérico en un instante. No forma parte de PDDL 2.2, que solo programa literales; es una extensión posterior, con su propio requisito, `:timed-initial-fluents`, que aceptan algunos planificadores temporales. Kairos la incluye entre sus características:

```
$ kairos --features
...
  timed-initial-literals     full     :timed-initial-literals
  timed-initial-fluents      full     :timed-initial-fluents
...
```

El problema `courier-hours-p05.pddl` es la entrega única a la sala A, con la farmacia abierta todo el día y horario de visitas de 20 a 60. El dominio no cambia: `move` ya calculaba su duración a partir de `(speed ?r)`.

```
$ kairos courier-hours-domain.pddl courier-hours-p05.pddl
; cost = 54.006
; makespan = 54.006
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
10.002: (pick r1 meds1 pharmacy) [2.000]
12.003: (move r1 pharmacy lobby) [6.000]
18.004: (move r1 lobby corridor) [10.000]
28.005: (move r1 corridor ward-a) [24.000]
52.006: (deliver r1 meds1 ward-a) [2.000]
```

El último movimiento, de 12 metros, dura 24 segundos: empieza en el instante 28, cuando la velocidad es 0,5. El movimiento anterior es más interesante. Empieza en el 18,004, dos segundos antes de que llegue la multitud, y dura sus 10 segundos habituales, aunque la mayor parte

transcurre en horario de visitas. No es un error del planificador. Es lo que dice el modelo: la duración de una acción durativa se calcula cuando la acción empieza, con los valores de ese instante (Figura 13.5). Si un robot al que la multitud sorprende a mitad de camino debiera frenar, el modelo tiene que ser más fino: movimientos más cortos, o el cambio continuo del Capítulo 12 con un ritmo que dependa de la velocidad.

Un fluente inicial temporizado fija un número en un instante conocido. A las acciones que leen el número al empezar no les afecta un cambio que llega mientras se ejecutan.

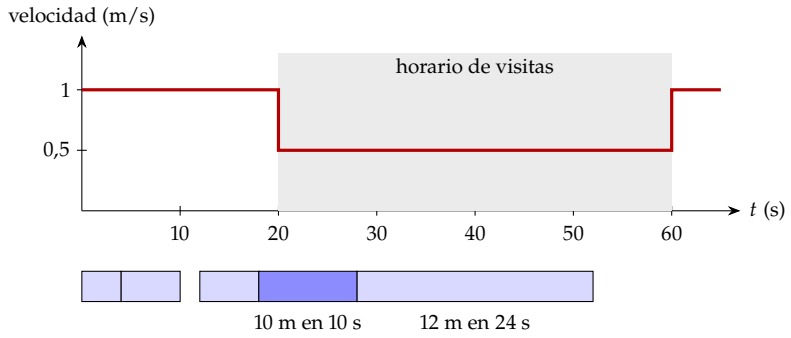


Figura 13.5.: La velocidad del robot, fijada por dos fluentes iniciales temporizados, y los movimientos del plan para el problema `p05`. Fíjate en el movimiento más oscuro: empieza en 18 con velocidad 1 y conserva la duración calculada entonces, aunque la velocidad baja en 20.

13.8. El método MODEL: consecuencias y calendario

Las dos mitades del capítulo tratan de cambios que no hace ninguna acción, y cada una añade una pregunta al método.

Preguntas nuevas para los pasos M y O

Mapear la situación. ¿Cambia el mundo en instantes conocidos, haga lo que haga el agente? Horarios de apertura, plazos, turnos, luz del día, pases de un satélite. Cada uno va a la ficha del modelo, en *Supuestos*, como un horario: qué cambia, cuándo, y que el agente no puede influir en ello. En PDDL se convierte en literales (o fluentes) iniciales temporizados en el problema, y en una condición en las acciones que dependen de ello: `over all` si la acción necesita el hecho todo el tiempo, `at end` si es un plazo.
Objetos y estado. Para cada propiedad de la tabla de estado: ¿es consecuencia de otras propiedades? Si conocer el resto del estado siempre dice su valor, es candidata a predicado derivado, y la tabla anota la regla en lugar de las acciones que la cambiarían. Después, pregúntate cómo se calcula la consecuencia: mirando un número fijo de hechos (como `free`) o siguiendo una cadena de longitud desconocida (como `reachable`).

El paso O en el mensajero con puertas

La tabla de estado de `courier-reach` tiene una clase nueva de fila.

Propiedad	Cambia cuando	Codificación
`(open ?d)`	un robot abre o cierra la puerta	básico, dinámico
`(joins ?d ?a ?b)`	nunca	básico, estático
`(passable ?a ?b)`	se deduce de `connected`, `joins`, `open`	derivado
`(reachable ?a ?b)`	se deduce de `passable`, en cualquier número de pasos	derivado, recursivo

Alternativas comparadas

El hecho `(free ?r)` se ha codificado ya de tres maneras, y las tres se ejecutan sobre el mismo problema (dos artículos de la farmacia a la sala A) con el mismo plan de 15 acciones:

- *Mantenido por efectos*: `courier-domain.pddl`, el dominio del Capítulo 4. `pick` borra el hecho, `drop` lo añade y el estado inicial lo declara.
- *Derivado*: `courier-free-domain.pddl`, Listado 13.1.
- *Escrito en línea*: `courier-inline-domain.pddl`, sin ningún predicado `free`; `pick` tiene la precondición `(not (exists (?j - item) (carrying ?r ?j)))`.

Tabla 13.1.: Tres codificaciones de «el robot no lleva nada». Los tamaños son los que da `kairos -v` para el problema con dos artículos.

	Mantenido por efectos	Derivado	Escrito en línea
Fidelidad	Correcto solo mientras todas las acciones y todos los estados iniciales mantengan acompasados los dos predicados	Correcto por construcción	Correcto por construcción
Tamaño	27 hechos, 46 acciones, sin reglas	27 hechos, 46 acciones, 1 regla	26 hechos, 46 acciones, sin reglas
Guía	La mejor: el hecho lo añaden y lo borran acciones, y eso lo ven las heurísticas	La heurística tiene que mirar a través de la regla	La heurística tiene que mirar a través de un `exists` negado
Claridad	Dos predicados para una idea; el invariante está en la cabeza del modelador	Una definición, con nombre, utilizable en cualquier parte	La condición se repite allí donde se necesita
Portabilidad	Cualquier planificador	Necesita `:derived-predicates`	Necesita precondiciones negativas y existenciales

Veredicto. Para un hecho tan simple como `free`, que usa una sola acción, mantenerlo mediante efectos sigue siendo una buena opción: cuesta dos efectos, lo acepta cualquier planificador y es lo que más le da a la búsqueda para trabajar. Por eso el mundo del mensajero de este libro lo conserva. La versión derivada se gana su sitio cuando el hecho se usa en muchas acciones, o cuando al dominio se le siguen añadiendo acciones nuevas y cada una es una ocasión de olvidar la contabilidad. Escribir la condición en línea es lo adecuado cuando se necesita una sola vez y el planificador no admite predicados derivados.

Para `reachable` no hay tabla que dibujar, porque una de las columnas está vacía: el hecho no puede mantenerse mediante efectos. La alternativa real es otro nivel de detalle, el `move` paso a paso por puertas transitables de `courier-steps-domain.pddl`. En el problema con todas las puertas cerradas da un plan de 11 acciones donde `go` necesita 9, con 392 reglas instanciadas en lugar de 518 y 42 acciones instanciadas en lugar de 92. La regla recursiva no es más barata. Es la opción correcta cuando la

navegación no es asunto del planificador, y permite que los objetivos y las condiciones hablen de la propia alcanzabilidad, como en «dejar la UCI aislada».

Un plazo: literal temporizado o restricción

Un plazo también puede escribirse sin tocar el dominio, con las restricciones sobre el plan de PDDL 3 que estudia el Capítulo 15. El problema `courier-hours-p06.pddl` es `p03` sin el literal temporizado y con una sección más:

```
(:constraints (within 50 (delivered meds1))))
```

Kairos lo resuelve con un plan válido que entrega primero los medicamentos, y su validador rechaza el plan que sirve primero la comida, con el mensaje `the plan violates the problem :constraints`. Aquí, las dos codificaciones aceptan los mismos planes. Difieren en los demás criterios. La restricción es más clara: dice «antes de 50» en una línea del problema, mientras que el literal temporizado necesita un predicado auxiliar, una condición en una acción y dos hechos por artículo. El literal temporizado es más portable entre los planificadores temporales, y puede expresar lo que `within` no puede: una ventana que se abre, se cierra y vuelve a abrirse, o un hecho que una acción necesita *todo el tiempo*. Usa `within` para un plazo simple sobre un objetivo cuando el planificador lo admita, y literales temporizados para todo lo que sea un horario del mundo.

13.9. Errores comunes

Escribir un predicado derivado

El hábito de mantener los hechos a mano sobrevive a la llegada de la regla: el modelador añade `:derived` y deja en su sitio un efecto antiguo, o un hecho inicial antiguo.

```
$ kairos check courier-free-wrong-domain.pddl
courier-free-wrong-domain.pddl:22:18: error: in action 'drop':
     derived predicate 'free' cannot be changed by an effect [
     semantic.derived_in_effect]
...
```

En un problema, el mensaje es `derived predicate 'free' cannot be initialized`. La corrección es siempre borrar la línea culpable: la regla ya lo dice.

Una regla que solo mira un paso más allá

Escribir `reachable` sin la recursión,

```
(:derived (reachable ?a ?b - location)
  (or (passable ?a ?b)
      (exists (?c - location)
        (and (passable ?a ?c) (passable ?c ?b)))))
```

da un modelo que funciona en todas las pruebas con habitaciones a dos pasos como mucho y falla en el primer pasillo largo, con un plan lleno de saltos cortos o sin plan. Ningún mensaje lo avisa: la

regla es legal y significa otra cosa. La batería de pruebas del paso E necesita un problema más profundo de lo que alcanzaría la regla sin recursión.

El hecho ya es cierto

Un literal temporizado que hace cierto un hecho no tiene ningún efecto si el hecho es cierto desde el principio. En `courier-hours-wrong-p01.pddl` el modelador declaró abiertas todas las habitaciones, también la farmacia, y después añadió su hora de apertura:

```
    (is-open pharmacy)
    (at 30 (is-open pharmacy))
```

```
$ kairos courier-hours-domain.pddl courier-hours-wrong-p01.pddl
; cost = 42.006
; makespan = 42.006
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
10.002: (pick r1 meds1 pharmacy) [2.000]
...
```

El robot recoge en el instante 10, veinte segundos antes de que abra la farmacia. El fichero es PDDL correcto y la comprobación lo acepta. Para cada hecho del calendario, decide su valor en el instante cero, y escribe el primer literal temporizado como el primer *cambio*.

Una ventana comprobada solo al inicio

Con `(at start (is-open ?l))` en lugar de `over all`, una acción puede empezar un instante antes de que cierre la habitación y terminar dentro de una habitación cerrada. Que eso esté mal depende del mundo: a un cliente que está dentro cuando cierra la tienda se le sigue atendiendo; un satélite cuya zona sale del campo de visión a mitad de la exposición se queda sin imagen. Decídelo a propósito, con las tres preguntas del Capítulo 10 (qué debe cumplirse al inicio, todo el tiempo y al final), y pruébalo con un problema cuya ventana sea más corta que la acción, como `satellite-p02.pddl`.

Un requisito olvidado

Los literales temporizados se escriben en el problema, pero el requisito va con los demás, y es fácil olvidarlo porque el dominio en sí no cambia. Kairos planifica de todos modos y avisa en la comprobación:

```
warning: uses timed-initial-literals but :requirements does not
     include :timed-initial-literals [semantic.
     undeclared_requirement]
```

El mismo aviso existe para `:derived-predicates`. Un problema puede declarar requisitos propios, como hace `courier-hours-p05.pddl` con `:timed-initial-fluents`.

13.10. Resumen

PDDL 2.2 dio a los hechos dos maneras nuevas de ser ciertos. Un predicado derivado es cierto porque lo dice una regla sobre el estado

actual; un literal inicial temporizado hace un hecho cierto o falso porque el reloj ha llegado a un instante escrito en el problema. Ninguno de los dos está bajo el control del agente, y los dos permiten que un modelo diga directamente algo que antes había que simular a base de contabilidad.

Tabla 13.2.: Lo que ha añadido este capítulo.

Construcción o idea	Qué recordar
`(:derived (p ?x) body)`	`p` se cumple exactamente cuando se cumple el cuerpo; requisito `:derived-predicates`
Dónde puede aparecer	Se lee en cualquier parte (condiciones, objetivos, otras reglas); nunca en efectos ni en el estado inicial
Varias reglas, una cabeza	Basta con cualquiera de ellas
Regla recursiva	Los hechos se construyen ronda a ronda desde cero; para caminos, torres, redes
Negación	Permitida si ningún predicado depende negativamente de sí mismo (estratificación)
Coste	Reglas instanciadas evaluadas en cada estado; heurísticas más débiles; menos planificadores
`(at 30 (p))` en `:init`	`p` se hace cierto en el instante 30, haga lo que haga el plan; `(at 50 (not (p)))` lo hace falso; requisito `:timed-initial-literals`
Ventana	Un literal que abre y otro que cierra; las acciones necesitan el hecho `over all`
Plazo	Un hecho que se retira al vencer el plazo y que necesita `at end` la acción que debe llegar a tiempo
`(at 20 (= (f) 0.5))`	Un fluente inicial temporizado: un número fijado en un instante conocido; una extensión, `:timed-initial-fluents`

13.11. Lecturas complementarias

El informe que define PDDL 2.2 Edelkamp y Hoffmann, 2004 es breve y presenta las dos construcciones, con las restricciones que les impuso la competición de 2004. La defensa de los predicados derivados, con la demostración de que no pueden eliminarse por compilación a bajo coste y un análisis de las reglas estratificadas, está en el artículo de Thiébaux, Hoffmann y Nebel Thiébaux et al., 2005. Cómo evalúa un planificador las reglas por capas durante la búsqueda se describe junto con Fast Downward Helmert, 2006. Para los literales iniciales temporizados en un planificador temporal, véanse POPF Coles et al., 2010 y OPTIC Benton et al., 2012, cuya búsqueda hacia delante ordena las acciones de un plan en torno a los acontecimientos fijos. La restricción `within` usada en la última comparativa pertenece a PDDL 3 A. Gerevini y Long, 2005 y al Capítulo 15. El libro de Haslum y colaboradores Haslum et al., 2019 trata las dos características de PDDL 2.2 con más ejemplos.

13.12. Ejercicios

Ejercicios de la Sección 13.1 — Hechos que se siguen de otros hechos

Ejercicio 13.1.1. Para cada propiedad, di si es básica o consecuencia de otras, y de cuáles: (a) una habitación está vacía (no hay en ella ningún robot ni ningún artículo); (b) una puerta está abierta; (c) un robot está sobrecargado (lleva más de lo que permite su capacidad);

(d) dos robots están en la misma habitación; (e) un artículo está en la farmacia.

Ejercicio 13.1.2. Un modelo nuevo de robot puede llevar dos artículos. En el dominio del Capítulo 4, ¿qué tendrían que hacer `pick` y `drop` para mantener correcto `(free ?r)`, con el significado «puede coger un artículo más»? Explica por qué esta contabilidad es más difícil que con un artículo, sin escribir el PDDL.

Ejercicios de la Sección 13.2 — Predicados derivados

Ejercicio 13.2.1. Añade al dominio del mensajero con tipos un predicado derivado `(tidy ?l)`: no hay ningún artículo en la habitación. Escribe un problema en el que haya dos artículos en el pasillo y el objetivo pida un pasillo despejado y un vestíbulo despejado, y lee el plan: ¿dónde ha dejado el planificador los artículos?

Ejercicio 13.2.2. Un compañero recibe el dominio del Listado 13.1 con su problema y añade `(free r1)` al estado inicial «por si acaso». Predice el mensaje de `kairos check` y explica por qué el lenguaje prohíbe lo que parece una redundancia inofensiva.

Ejercicio 13.2.3. En una habitación restringida puede entrar un robot que tenga tarjeta de acceso, y en cualquier otra habitación, cualquier robot. Escribe `(may-enter ?r ?l)` de dos maneras: con una regla y un `or`, y con dos reglas. ¿Cuál se lee mejor?

Ejercicios de la Sección 13.3 — Reglas que se llaman a sí mismas

Ejercicio 13.3.1. En el mapa de la Figura 13.1, supón que `d-corridor` está cerrada y las otras dos puertas están abiertas. Calcula por rondas las habitaciones `?b` para las que se cumple `(reachable ward-a ?b)`. ¿Está `(reachable ward-a ward-a)` entre ellas? ¿Importa eso para la acción `go`?

Ejercicio 13.3.2. Un jardín tiene una toma de agua y puntos unidos por tuberías, cada una con una válvula. El agua llega a un punto si es la toma o si una válvula abierta lo une con un punto que tiene agua. Modélalo, con acciones que abren y cierran válvulas, y resuelve este caso: el jardín puede recibir agua a través de la piscina o a través del cobertizo; dale agua mientras la piscina, que está en reparación, sigue seca.

Ejercicio 13.3.3. En el dominio de bloques del Listado 13.3, escribe el objetivo «`a`, `b` y `c` están en la misma torre, en cualquier orden» usando solo `above`. Después explica por qué `blocks-above-p02.pddl` es irresoluble, y por qué Kairos puede demostrarlo aquí cuando en el Capítulo 12 a menudo no podía.

Ejercicio 13.3.4. Explica qué falla en el par de reglas «una lámpara está encendida si no está apagada» y «una lámpara está apagada si no está encendida», las dos como predicados derivados. ¿Qué haría con ellas el procedimiento ronda a ronda?

Ejercicios de la Sección 13.4 — Qué hace un planificador con las reglas

Ejercicio 13.4.1. El dominio de las puertas existe con `go` y con `move` (los ficheros `courier-reach` y `courier-steps`). Ejecuta el problema `p02` de cada uno con `-v` y compara las líneas que empiezan por

`[ground]`. Explica de dónde sale la diferencia en el número de reglas y en el número de acciones.

Ejercicio 13.4.2. Estima cuántas reglas instanciadas tendría `reachable` en un hospital con 100 habitaciones. Propón un modelo para ese hospital en el que el robot siga sin planificar cada paso. (Pista: ¿abre el robot las puertas?)

Ejercicios de la Sección 13.5 — Un mundo con calendario

Ejercicio 13.5.1. La farmacia abre dos veces: de 0 a 8 y de 40 a 60. Escribe el problema (una entrega a la sala A, como en `p01`) y predice el *makespan* antes de ejecutarlo. ¿Podría el robot aprovechar la primera ventana?

Ejercicio 13.5.2. En el plan de `courier-hours-p01.pddl`, el robot va primero a la farmacia y espera allí. ¿Es válido también un plan en el que espera en la base de carga y llega justo en el instante 30? ¿Es mejor o peor según el makespan? Da una razón, ajena al modelo, para preferir uno de los dos.

Ejercicios de la Sección 13.6 — Ventanas y plazos

Ejercicio 13.6.1. En `courier-hours-p02.pddl`, ¿cuál es la hora de cierre más temprana del laboratorio con la que el problema sigue teniendo plan? ¿Cuál es la hora de apertura más tardía de la ventana de la farmacia (manteniendo su duración de 20 segundos) con la que todavía se puede atender el laboratorio, que cierra en 90?

Ejercicio 13.6.2. Un róver solo se desplaza y toma fotografías con luz de día. Un desplazamiento entre emplazamientos vecinos dura 30 segundos, y una fotografía, 10. Modélalo con un hecho `(daylight)` en el calendario: dos días de 100 segundos con una noche de 50 entre ellos. Hay tres emplazamientos en línea, la base, la duna y la roca; el vehículo empieza en la base y debe fotografiar la duna y la roca y volver. Lee el plan: ¿qué hace el vehículo durante la noche?

Ejercicio 13.6.3. En el problema del satélite, el plan observa primero el cráter. Escribe a mano el plan que observa primero el volcán y valídalo. ¿Qué condición falla, y en qué instante?

Ejercicios de la Sección 13.7 — Números en el calendario

Ejercicio 13.7.1. En `courier-hours-p05.pddl`, cambia el horario de visitas: la multitud llega en el instante 10 y se va en el 30. Predice qué debería hacer un robot con prisa; después ejecuta el planificador y lee todos los planes que imprime.

Ejercicio 13.7.2. Nombra dos magnitudes de un hospital, aparte de la velocidad de un robot, que cambien en instantes conocidos y que un planificador de entregas querría tener como fluentes iniciales temporizados.

Ejercicios de la Sección 13.8 — El método MODEL

Ejercicio 13.8.1. En el dominio del Listado 13.5, `(delivered ?i)` lo añade la acción `deliver`. ¿Podría ser un predicado derivado, «el artículo está en su destino»? Escribe la regla y di qué se perdería en lo que respecta al plazo.

Ejercicio 13.8.2. Clasifica cada cambio según quién lo causa, el agente (una acción), el estado (una regla) o el calendario (un literal temporizado): (a) la cafetería cierra a las cuatro; (b) una habitación se congestiona cuando hay en ella más de tres robots; (c) un robot cierra una puerta con llave; (d) empieza el turno de noche; (e) una sala está abastecida cuando tiene sus medicamentos y su comida.

Parte V. Preferencias y restricciones: PDDL 3

14. Objetivos blandos y preferencias

Todos los objetivos que hemos escrito hasta ahora eran órdenes. El planificador o los alcanzaba completos o respondía que no había plan, y un modelo con un objetivo de más era un modelo sin planes. Las peticiones reales rara vez son así. La lista de la mañana del mensajero dice: medicamentos a la sala A, medicamentos a la UCI, una muestra al laboratorio. La batería tiene 100 unidades y nadie ha reparado el cargador. Si las tres entregas necesitan más de 100, un planificador que lea la lista como un único objetivo responde `unsolvable` y el robot se queda toda la mañana en la base de carga, aunque podría haber atendido dos salas de tres.

PDDL 3.0 añade la construcción para esa situación: la **preferencia**, una condición que el plan debería cumplir y puede incumplir, a un precio que fija el modelador. Con preferencias, un problema tiene dos clases de objetivos, los que deben cumplirse y los que se desean, y la métrica del Capítulo 9 dice cuánto vale cada deseo. El planificador resuelve entonces un problema distinto: ya no «alcanzar el objetivo», sino «alcanzar lo exigido y renunciar a lo menos posible de lo deseado».

El capítulo parte de la lista que no cabe en la batería y convierte sus entregas en deseos. Después da a los deseos pesos distintos, los enfrenta a los metros que cuesta concederlos, escribe una sola preferencia para muchos objetos con `forall` y lleva las preferencias del objetivo a las precondiciones de una acción, donde hablan del camino y no del final. Otros dos mundos, el día de un turista y un satélite con más peticiones que tiempo, usan las mismas herramientas. Las últimas secciones miran qué hace un planificador con una preferencia, añaden los deseos al método MODEL y comparan una preferencia con sus dos vecinos: el objetivo estricto y el coste de acción.

14.1 Más deseos que batería 323
14.2 Una preferencia en el objetivo 324
14.3 Lo que vale cada deseo 327
Un deseo sobre el propio robot 328
Objetivos estrictos y blandos juntos 328
14.4 Deseos frente a esfuerzo 329
14.5 Una preferencia, muchas instancias 330
14.6 Deseos sobre el camino: preferencias en las precondiciones 331
14.7 Otros dos mundos . . . 333
El día de un turista . . . 333
Un satélite con más peticiones que tiempo . 335
14.8 Qué hace el planificador con una preferencia . . 336
14.9 El método MODEL: los deseos y su valor 338
Alternativas comparadas 338
14.10 Errores comunes 340
14.11 Resumen 341
14.12 Lecturas complementarias 341
14.13 Ejercicios 342

14.1. Más deseos que batería

Tomamos el mundo del mensajero con tipos del Capítulo 4 y la batería del Capítulo 8, en su forma más sencilla: cada desplazamiento consume tantas unidades de batería como metros recorre, y hoy no hay acción `charge`. La única línea nueva del dominio es su requisito, `:preferences`, que el primer problema todavía no usa.

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (>= (battery ?r) (distance ?from ?to)))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (decrease (battery ?r) (distance ?from ?to))))
```

El problema `courier-battery-p01.pddl` contiene la lista de la mañana. El robot `r1` empieza en la base de carga con una batería de 100; `meds1` y `meds2` esperan en la farmacia y `sample1` en la sala B (Figura 14.1). El objetivo lo pide todo:

```
(:goal (and (item-at meds1 ward-a)
            (item-at meds2 icu)
            (item-at sample1 lab)))
```

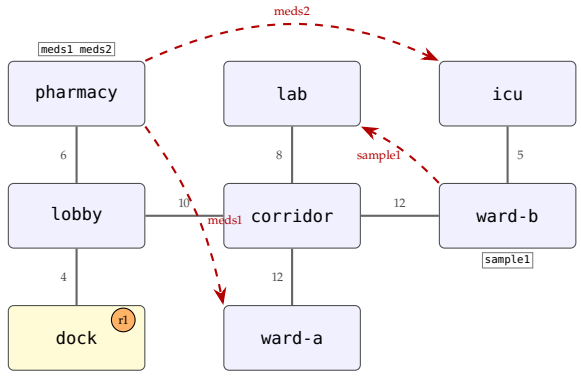


Figura 14.1.: La lista de la mañana. Las flechas discontinuas son las tres entregas pedidas; los números son metros, y cada metro cuesta una unidad de una batería de 100. El robot lleva un artículo cada vez.

```
$ kairos courier-battery-domain.pddl courier-battery-p01.pddl
note: no plan: every state reachable from the initial one was
     searched (37168 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

La respuesta es correcta e inútil. Correcta, porque ningún orden de las tres entregas cabe en 100 unidades: la Figura 14.2 muestra lo que consume cada subconjunto de ellas. Inútil, porque la pregunta que tiene el hospital es otra: qué entregas hacer con la batería que hay.

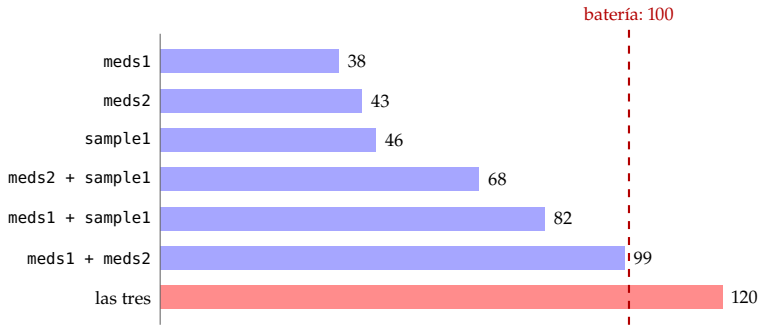


Figura 14.2.: Batería necesaria para hacer cada conjunto de entregas, partiendo de la base de carga. Dos entregas cualesquiera caben; las tres juntas necesitan 120.

Un problema con más objetivos de los que permiten sus recursos se llama **sobresuscrito**. Es el caso normal siempre que las peticiones vienen de varias personas y los recursos (batería, tiempo, combustible, memoria) vienen de un solo robot. El remedio clásico es borrar objetivos a mano hasta que aparezca un plan, pero eso obliga al modelador a hacer el trabajo del planificador: elegir *qué* objetivos borrar es justamente la parte difícil, y depende de las distancias, de la batería y del orden de los viajes.

Un objetivo es todo o nada. Cuando las peticiones superan los recursos, un modelo hecho solo de objetivos estrictos no tiene plan, y no dice nada de la mejor respuesta parcial.

14.2. Una preferencia en el objetivo

Una **preferencia** es una condición con nombre, que se escribe `(preference name condition)`. Dentro de un objetivo significa: esta condición debería cumplirse al final del plan, pero un plan en el que no se cumple sigue

siendo un plan. Un objetivo que es una preferencia se llama también **objetivo blando**; los normales son **objetivos estrictos**. El requisito es `:preferences`.

El problema `courier-battery-p02.pddl` es la lista de la mañana con sus tres entregas convertidas en objetivos blandos (Listado 14.1).

```
;; The same three deliveries, now as wishes of equal worth.
(define (problem courier-battery-p02)
  (:domain courier-battery)
  (:objects
    r1 - robot
    lobby corridor pharmacy lab ward-a ward-b icu dock - location
    meds1 meds2 sample1 - item)
  (:init
    (connected lobby corridor) (connected corridor lobby)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (connected lobby dock) (connected dock lobby)
    (connected corridor lab) (connected lab corridor)
    (connected corridor ward-a) (connected ward-a corridor)
    (connected corridor ward-b) (connected ward-b corridor)
    (connected ward-b icu) (connected icu ward-b)
    (= (distance lobby corridor) 10)
    (= (distance corridor lobby) 10)
    (= (distance lobby pharmacy) 6)
    (= (distance pharmacy lobby) 6)
    (= (distance lobby dock) 4) (= (distance dock lobby) 4)
    (= (distance corridor lab) 8) (= (distance lab corridor) 8)
    (= (distance corridor ward-a) 12)
    (= (distance ward-a corridor) 12)
    (= (distance corridor ward-b) 12)
    (= (distance ward-b corridor) 12)
    (= (distance ward-b icu) 5) (= (distance icu ward-b) 5)
    (robot-at r1 dock) (free r1)
    (= (battery r1) 100)
    (item-at meds1 pharmacy) (item-at meds2 pharmacy)
    (item-at sample1 ward-b))
  (:goal (and (preference meds-a (item-at meds1 ward-a))
              (preference meds-icu (item-at meds2 icu))
              (preference sample-lab (item-at sample1 lab))))
  (:metric minimize (+ (is-violated meds-a)
                       (is-violated meds-icu)
                       (is-violated sample-lab))))
```

Listado 14.1: Las tres entregas como preferencias, todas con el mismo valor.

Han cambiado dos cosas. En el objetivo, cada entrega va envuelta en una preferencia y tiene un nombre: `meds-a`, `meds-icu`, `sample-lab`. Y el problema tiene una `:metric`, que es donde se usan los nombres. La expresión `(is-violated meds-a)` es un número: 0 si el plan cumple la preferencia `meds-a`, 1 si no la cumple. La métrica suma las tres, así que su valor es el número de entregas que *no* se hacen, y al planificador se le pide que lo minimice.

Un plan **satisface** una preferencia del objetivo cuando su condición se cumple en el último estado, y la **viola** en caso contrario. La palabra es más fuerte que lo que significa: violar una preferencia no rompe ninguna regla, solo cuesta lo que dice la métrica. Como `is-violated` cuenta lo que salió mal, las métricas con preferencias son casi siempre `minimize`.

```
$ kairos courier-battery-domain.pddl courier-battery-p02.pddl
; cost = 3
; better plan 2
; cost = 2
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
```

```
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
; better plan 3
; cost = 1
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor ward-b)
(pick r1 sample1 ward-b)
(move r1 ward-b corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

La salida es la búsqueda *anytime* del Capítulo 9 en acción, y merece la pena leerla desde arriba. El primer plan tiene coste 3 y *ninguna acción*: no hay nada después de su línea de coste. Con solo objetivos blandos, no hacer nada es un plan válido, el peor: viola las tres preferencias. El segundo plan entrega `meds1` y cuesta 2. El tercero añade `sample1` y cuesta 1, y ahí la búsqueda se detiene sola: ningún plan viola menos de una preferencia, porque las tres entregas no caben. El coste que imprime Kairos es el valor de la métrica.

Con preferencias, el planificador siempre tiene un plan, el vacío, y la búsqueda se convierte en una búsqueda de planes mejores. El coste impreso es el valor de la métrica.

Un plan escrito a mano se valida como siempre, y `kairos validate` informa de la métrica que alcanza. El fichero `courier-battery-p03-lazy.plan` contiene las siete acciones que solo entregan `meds1`:

```
$ kairos validate courier-battery-domain.pddl courier-battery-p02.
    pddl courier-battery-p03-lazy.plan
plan valid, makespan 7, metric 2
```

Válido, con una métrica de 2: un plan perezoso no es un plan erróneo. Lo que cambian las preferencias es el sentido de «válido» frente a «bueno». La validez la siguen decidiendo las precondiciones y los objetivos estrictos; todo lo que tiene que ver con los deseos vive en la métrica.

Paso M: obligatorio o deseado

La ficha del modelo del mundo del mensajero tiene una sola línea de *Éxito* desde el Capítulo 2. Para un día sobresuscrito, esa línea se divide en dos:

- **Obligatorio**: lo que un plan tiene que conseguir para ser aceptable. Hoy, nada; mañana, quizá la muestra que espera el laboratorio.
- **Deseado**: lo que debería conseguirse si es posible, cada deseo con su valor.

La prueba consiste en imaginar el plan que deja fuera la condición. Si ese plan es *inservible*, la condición es obligatoria. Si solo es *peor*, es un deseo.

14.3. Lo que vale cada deseo

En `p02` los tres deseos valían lo mismo, y el planificador renunció a la entrega de la UCI. La supervisora de enfermería no está de acuerdo: la UCI va primero. La métrica es una expresión aritmética, así que cada `is-violated` puede multiplicarse por un **peso**, el precio de violar esa preferencia. El problema `courier-battery-p03.pddl` solo cambia la métrica:

```
(:metric minimize (+ (* 3 (is-violated meds-a))
                     (* 5 (is-violated meds-icu))
                     (* 1 (is-violated sample-lab))))
```

Dejar la UCI sin sus medicamentos cuesta ahora 5; la sala A, 3, y el laboratorio, 1. La Tabla 14.1 da la métrica de cada conjunto de entregas que cabe en la batería; el planificador tiene que encontrar la fila con el valor más pequeño.

Entregas hechas	Batería	`p02` (1, 1, 1)	`p03` (3, 5, 1)
ninguna	0	3	9
`meds1`	38	2	6
`meds2`	43	2	4
`sample1`	46	2	8
`meds2`, `sample1`	68	1	3
`meds1`, `sample1`	82	1	5
`meds1`, `meds2`	99	1	**1**

Tabla 14.1.: La métrica de cada conjunto de entregas posible, con pesos iguales y con los pesos de la supervisora. Con pesos iguales empatan tres conjuntos a 1; con pesos, solo uno es el mejor.

```
$ kairos courier-battery-domain.pddl courier-battery-p03.pddl
; cost = 9
; better plan 2
; cost = 4
...
; better plan 3
; cost = 3
...
; better plan 4
; cost = 1
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
...
(drop r1 meds1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lobby)
(move r1 lobby pharmacy)
(pick r1 meds2 pharmacy)
...
(move r1 ward-b icu)
(drop r1 meds2 icu)
```

Los costes 9, 4, 3 y 1 son cuatro filas de la tabla: nada, `meds2` solo, `meds2` con `sample1` y, por último, los dos medicamentos. El último plan llega a la UCI con una unidad de batería. El mismo dominio, el mismo estado inicial, los mismos deseos: solo ha cambiado su valor, y con él el plan.

Un deseo sobre el propio robot

Una preferencia puede envolver cualquier condición del objetivo, no solo entregas. A los técnicos les gustaría que el robot terminara la mañana en la base de carga, donde pueden revisar el cargador. Es un objetivo blando más, que `courier-battery-p04.pddl` escribe como `(preference parked (robot-at r1 dock))` y al que da un peso de 4:

```
  (:goal (and (preference meds-a (item-at meds1 ward-a))
              (preference meds-icu (item-at meds2 icu))
              (preference sample-lab (item-at sample1 lab))
              (preference parked (robot-at r1 dock))))
  (:metric minimize (+ (* 3 (is-violated meds-a))
                       (* 5 (is-violated meds-icu))
                       (* 1 (is-violated sample-lab))
                       (* 4 (is-violated parked))))
```

El mejor plan de `p03` terminaba en la UCI con una batería de 1: no puede volver a casa. Volver compite con las entregas por la misma batería, y el planificador tiene que sopesar una cosa y otra:

```
$ kairos courier-battery-domain.pddl courier-battery-p04.pddl
; cost = 9
...
; better plan 4
; cost = 3
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds2 pharmacy)
...
(drop r1 meds2 icu)
(move r1 icu ward-b)
(pick r1 sample1 ward-b)
...
(drop r1 sample1 lab)
(move r1 lab corridor)
(move r1 corridor lobby)
(move r1 lobby dock)
```

El robot atiende la UCI y el laboratorio y vuelve a la base de carga con 10 unidades. Renuncia a la sala A (coste 3) porque la alternativa, los dos medicamentos y un robot varado en la UCI, cuesta 1 + 4 = 5. Fíjate en que el plan vacío sigue costando 9 y no 13: el robot empieza en la base de carga, así que el plan que no hace nada cumple `parked`.

Objetivos estrictos y blandos juntos

Los objetivos estrictos y las preferencias se mezclan sin problema dentro del mismo `and`. Si el laboratorio tiene que recibir hoy su muestra, pase lo que pase con lo demás, el problema `courier-battery-p05.pddl` deja esa entrega fuera de toda preferencia:

```
  (:goal (and (item-at sample1 lab)
              (preference meds-a (item-at meds1 ward-a))
              (preference meds-icu (item-at meds2 icu))))
  (:metric minimize (+ (* 3 (is-violated meds-a))
                       (* 5 (is-violated meds-icu))))
```

Ahora el plan vacío no es un plan. Kairos imprime tres planes, de coste 8, 5 y 3; el primero solo entrega la muestra y el último añade `meds2`. El

plan perezoso de antes, validado frente a `p05`, es ahora `plan invalid`: el objetivo `(item-at sample1 lab)` no se cumple al final.

Un peso es el precio de violar una preferencia. La métrica suma los precios de todo aquello a lo que el plan renuncia, y los objetivos estrictos quedan fuera del regateo.

Cómo elegir los pesos

Solo importa la relación entre los pesos: (3, 5, 1) y (30, 50, 10) describen el mismo problema. Tres patrones cubren la mayoría de las necesidades.

- **Pesos iguales**: maximizar el número de deseos concedidos.
- **Valor**: cada peso es lo que vale el deseo en una unidad común (euros, minutos, metros). Dos deseos pequeños pueden entonces ganar a uno grande, y eso es lo que se pretende.
- **Prioridad estricta**: un deseo de una clase superior tiene que ganar a *cualquier número* de deseos de las clases inferiores. Da a cada clase un peso mayor que la suma de todos los pesos que tiene por debajo. Con pesos 5, 3 y 1, la UCI gana a la sala A y al laboratorio juntos ($5 > 3 + 1$).

14.4. Deseos frente a esfuerzo

Hasta ahora solo se renunciaba a un deseo cuando no cabía. Pero un deseo puede caber y aun así no compensar el esfuerzo: nadie envía a un robot a 60 metros para entregar una servilleta. Para modelarlo, el esfuerzo y los deseos tienen que estar en la misma métrica, en la misma unidad.

El dominio `courier-cost` se olvida de la batería y usa los costes de acción del Capítulo 9: un desplazamiento suma sus metros a `total-cost`, y un `pick` o un `drop` suma 1. Sus artículos tienen los subtipos `medicine`, `sample` y `meal`. En `courier-cost-p01.pddl` hay dos deseos, un medicamento para la sala A y una comida para la UCI, que esperan los dos en la farmacia:

```
(:goal (and (preference meds-a (item-at meds1 ward-a))
            (preference meal-icu (item-at meal1 icu))))
(:metric minimize (+ (total-cost)
                     (* 60 (is-violated meds-a))
                     (* 20 (is-violated meal-icu))))
```

La métrica se lee como una frase: los metros recorridos, más 60 si no llega el medicamento, más 20 si no llega la comida. Los pesos están ahora *en metros*: el medicamento vale un viaje de hasta 60, y la comida, uno de hasta 20.

```
$ kairos courier-cost-domain.pddl courier-cost-p01.pddl
; cost = 80
; better plan 2
; cost = 60
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

No hacer nada cuesta $60 + 20 = 80$. Entregar el medicamento cuesta 40 (38 metros y dos manipulaciones) y ahorra 60: el plan baja a $40 +$

20 = 60. Entregar también la comida sumaría 63 al coste (volver a la farmacia y subir hasta la UCI) para ahorrar 20, así que el planificador deja la comida donde está y demuestra que no hay ningún plan más barato. Nada impedía esa entrega; simplemente era un mal negocio. En `courier-cost-p02.pddl` la comida vale 100, y el mismo dominio da un plan de coste 103 que hace las dos entregas: los planes no cambiaron, cambió su precio.

Cuando los costes y las violaciones comparten la métrica, un deseo solo se concede si vale más que el esfuerzo que añade. Los pesos tienen que estar en la unidad del coste.

Una métrica que enfrenta el valor de lo que se consigue con el coste de conseguirlo se llama métrica de **beneficio neto**. Tiene una consecuencia que al principio sorprende: con una métrica de beneficio neto no hay razón para esperar que el planificador consiga «todo lo posible». Consigue lo que compensa.

14.5. Una preferencia, muchas instancias

Mañana la farmacia envía cinco medicamentos a la UCI, y la semana que viene, veinte. Escribir una preferencia por artículo funciona, pero el fichero del problema crece con la lista, y los objetivos universales del Capítulo 5 resolvieron eso para los objetivos estrictos. Las preferencias tienen la misma herramienta: un `forall` *alrededor* de una preferencia.

```
(:goal (forall (?m - medicine)
         (preference meds-icu (item-at ?m icu))))
(:metric minimize (+ (total-cost)
                     (* 50 (is-violated meds-icu))))
```

Este objetivo de `courier-cost-p03.pddl` declara una preferencia por medicamento, todas con el nombre `meds-icu`. Son deseos independientes: el plan puede cumplir unos y violar otros. Y `(is-violated meds-icu)` ya no es 0 o 1: es el *número* de medicamentos que no están en la UCI al final. El problema tiene dos medicamentos y una comida en la farmacia (la comida no es un `medicine`, así que ninguna preferencia habla de ella). El problema declara `:universal-preconditions` para el `forall`.

```
$ kairos courier-cost-domain.pddl courier-cost-p03.pddl
; cost = 100
; better plan 2
; cost = 95
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds2 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds2 icu)
```

El plan vacío cuesta 100: dos violaciones a 50 cada una. El mejor plan entrega *un* medicamento: 45 de esfuerzo más una violación, 95. El segundo medicamento necesitaría 68 más (la vuelta y otra vez la subida) para ahorrar 50. Al planificador le da igual qué medicamento lleve: las dos instancias valen lo mismo.

La posición del `forall` importa. El otro orden, `(preference all-meds (forall (?m - medicine) (item-at ?m icu)))`, es un deseo distinto: *una* preferencia cuya condición es «todos los medicamentos están en la UCI». Se viola una vez, falten los medicamentos que falten, y un plan que

entrega uno de dos no obtiene nada a cambio. La Tabla 14.2 pone las dos formas una junto a otra.

	`forall` fuera	`forall` dentro
Se escribe	`(forall (?m) (preference p ...))`	`(preference p (forall (?m) ...))`
Preferencias	una por objeto	una
`(is-violated p)`	número de objetos que fallan	1 si falla algún objeto
Éxito parcial	cuenta	no cuenta

Tabla 14.2.: Un `forall` alrededor de una preferencia y un `forall` dentro de ella son dos deseos distintos.

Un forall alrededor de una preferencia crea una instancia por objeto con un solo nombre, e is-violated cuenta las instancias que fallan.

La misma cuenta se hace siempre que un nombre se usa más de una vez. Dos preferencias escritas por separado con el mismo nombre, por ejemplo `(preference meds (item-at meds1 ward-a))` y `(preference meds (item-at meds2 icu))`, se suman: `(is-violated meds)` puede valer 0, 1 o 2. Un nombre identifica una *clase* de deseos con un mismo precio, no un deseo concreto.

14.6. Deseos sobre el camino: preferencias en las precondiciones

Todos los deseos vistos hasta ahora trataban del último estado. Algunos deseos tratan de cómo llega el plan hasta él. En el vestíbulo esperan visitas, y el hospital preferiría que los robots no lo cruzaran con muestras biológicas. No está prohibido: si no hay otro camino, la muestra tiene que llegar igualmente al laboratorio. Un objetivo blando no puede decir esto, porque al final del plan el robot ya no está en el vestíbulo, haya hecho lo que haya hecho por el camino.

PDDL 3.0 permite una preferencia dentro de la precondición de una acción. La acción sigue siendo aplicable cuando la preferencia no se cumple; aplicarla en ese caso cuenta como una violación. El dominio `courier-lobby` añade una a `move`, con `lobby` declarado como constante del dominio (Capítulo 4):

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
    (preference clean-lobby
      (or (not (= ?to lobby))
          (not (exists (?s - sample) (carrying ?r ?s))))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (increase (total-cost) (distance ?from ?to))))
```

Las dos primeras condiciones son estrictas, como siempre. La tercera es el deseo: «el destino no es el vestíbulo, o el robot no lleva ninguna muestra». Un desplazamiento que entra en el vestíbulo con una muestra es legal y viola `clean-lobby`.

En una preferencia de precondición, `(is-violated clean-lobby)` es el número de veces que *cualquier* acción del plan se aplicó sin que se cumpliera la preferencia. Un robot que cruza el vestíbulo tres veces con muestras la viola tres veces.

En el mapa fijo de la planta, este deseo no deja elección, porque el vestíbulo es el único paso entre la farmacia y el resto. Por eso los problemas de

esta sección abren una puerta trasera entre la farmacia y el laboratorio, de 30 metros, donde el camino por el vestíbulo tiene 24 (Figura 14.3). Una muestra espera en la farmacia y tiene que llegar (objetivo estricto) al laboratorio. En `courier-lobby-p01.pddl` cada cruce tiene un precio de 10 metros:

```
(:goal (item-at sample1 lab))
(:metric minimize (+ (total-cost)
                     (* 10 (is-violated clean-lobby))))
```

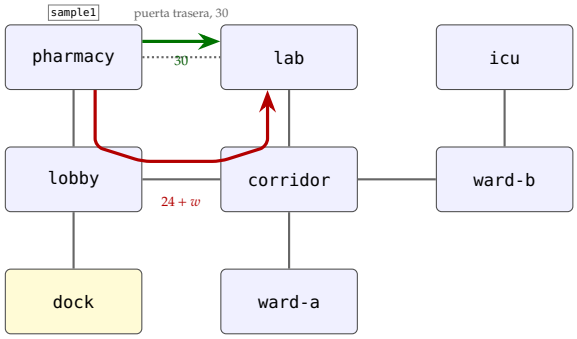


Figura 14.3.: Dos caminos para llevar la muestra de la farmacia al laboratorio. El rojo es más corto, pero entra en el vestíbulo con una muestra y paga el peso w de `clean-lobby`. El verde es la puerta trasera que esta sección añade al mapa.

```
$ kairos courier-lobby-domain.pddl courier-lobby-p01.pddl
; cost = 42
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 sample1 pharmacy)
(move r1 pharmacy lab)
(drop r1 sample1 lab)
```

El robot da la vuelta: 10 metros hasta la farmacia, 30 por la puerta trasera y dos manipulaciones, 42 en total. El camino por el vestíbulo costaría $36 + 10 = 46$. En `courier-lobby-p02.pddl` el mismo cruce tiene un precio de 3, y la respuesta se invierte:

```
$ kairos courier-lobby-domain.pddl courier-lobby-p02.pddl
; cost = 42
...
; better plan 2
; cost = 39
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 sample1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

El segundo plan viola la preferencia una vez, en su cuarta acción, y aun así es el mejor: $36 + 3 = 39$. La Figura 14.4 dibuja los dos planes en función del peso: se cruzan en $w = 6$.

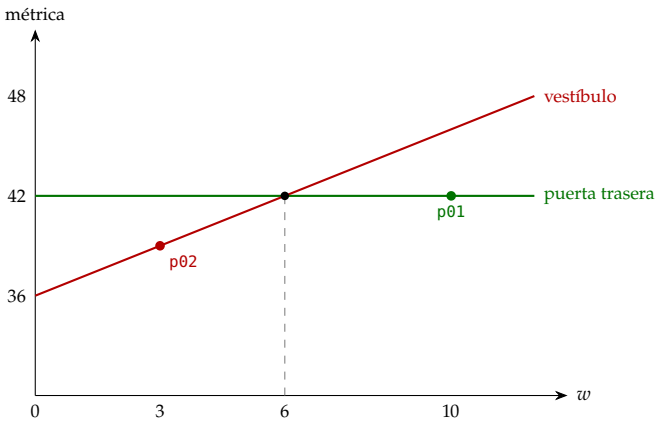


Figura 14.4.: La métrica de las dos rutas a medida que crece el peso w de `clean-lobby`. Por debajo de 6, el planificador cruza el vestíbulo; por encima de 6, toma la puerta trasera. Los puntos son los planes que devolvió Kairos para los dos problemas.

Cuando no hay forma de dar la vuelta, la preferencia se paga tantas veces como se incumple. `courier-lobby-p03.pddl` no tiene puerta trasera y tiene dos muestras. Kairos devuelve un único plan de 15 acciones y coste 106: 86 de esfuerzo y dos entradas en el vestíbulo con una muestra, a 10 cada una.

Una preferencia en una precondición nunca bloquea la acción. Se viola una vez cada vez que la acción se aplica sin que se cumpla, y el plan paga por cada una.

Una preferencia en una precondición no es una precondición

Si se lee `courier-lobby-domain.pddl` deprisa, la tercera condición de `move` parece una norma del hospital. No lo es: el planificador la incumplirá siempre que salga más barato, y la incumplirá gratis si la métrica no menciona `clean-lobby`. Si una condición no debe incumplirse nunca, escríbela sin `preference`. Con la norma del vestíbulo como precondición estricta y sin puerta trasera (`courier-lobby-hard-p02.pddl`), la muestra no puede salir de la farmacia:

```
$ kairos courier-lobby-hard-domain.pddl courier-lobby-hard-p02.
    pddl
note: no plan: every state reachable from the initial one was
    searched (18 states) and none satisfies the goal. [
    orchestrator.why]
kairos: unsolvable (exhausted)
```

14.7. Otros dos mundos

El día de un turista

Un turista tiene 240 minutos antes de la cena y cuatro lugares que visitar alrededor del hotel. Caminar de un sitio a otro lleva tiempo, cada visita lleva tiempo, y un autobús une dos sitios cualesquiera en 5 minutos. El turista tiene que estar de vuelta en el hotel para cenar, le gustaría verlo todo y preferiría ir andando. El Listado 14.2 es el dominio: planificación numérica sin más, con un hecho `(used-bus)` que recuerda si se cogió el autobús.

```
;; A tourist's day: walk or take the bus between places, visit
;; sights, with a limited number of minutes.
(define (domain tourist)
  (:requirements :strips :typing :negative-preconditions
                 :numeric-fluents :preferences)

  (:types place - object
```

```
        sight - place)

  (:predicates
    (at ?p - place)               ; where the tourist is
    (seen ?s - sight)
    (used-bus))                   ; took the bus at least once

  (:functions
    (minutes-left)
    (walk-time ?from ?to - place)
    (visit-time ?s - sight))

  (:action walk
    :parameters (?from ?to - place)
    :precondition (and (at ?from)
                       (>= (minutes-left) (walk-time ?from ?to)))
    :effect (and (not (at ?from)) (at ?to)
                 (decrease (minutes-left) (walk-time ?from ?to))))

  ; the bus takes 5 minutes between any two places
  (:action bus
    :parameters (?from ?to - place)
    :precondition (and (at ?from) (>= (minutes-left) 5))
    :effect (and (not (at ?from)) (at ?to) (used-bus)
                 (decrease (minutes-left) 5)))

  (:action visit
    :parameters (?s - sight)
    :precondition (and (at ?s)
                       (>= (minutes-left) (visit-time ?s)))
    :effect (and (seen ?s)
                 (decrease (minutes-left) (visit-time ?s)))))
```

Listado 14.2: El dominio del turista: los minutos son el recurso que se agota.

El objetivo de `tourist-p01.pddl` tiene las tres clases de condición de este capítulo: un objetivo estricto, deseos positivos con valores distintos y el deseo de que algo *no* haya ocurrido.

```
(:goal (and (at hotel)
            (preference see-museum (seen museum))
            (preference see-cathedral (seen cathedral))
            (preference see-market (seen market))
            (preference see-tower (seen tower))
            (preference on-foot (not (used-bus)))))
(:metric minimize (+ (* 5 (is-violated see-museum))
                     (* 3 (is-violated see-cathedral))
                     (* 2 (is-violated see-market))
                     (* 4 (is-violated see-tower))
                     (* 1 (is-violated on-foot))))
```

Solo las visitas suman 220 minutos, así que los cuatro lugares no caben a pie. Kairos imprime cuatro planes, de coste 14, 5, 4 y 3; el último es:

```
$ kairos tourist-domain.pddl tourist-p01.pddl
...
; better plan 4
; cost = 3
(walk hotel cathedral)
(visit cathedral)
(walk cathedral museum)
(visit museum)
(walk museum tower)
(visit tower)
(bus tower hotel)
```

El turista ve los tres lugares que más valen, se salta el mercado (2) y coge el autobús una vez (1), para la vuelta: a pie, desde la torre el hotel está a 30 minutos, y solo le quedan 5. El deseo `on-foot` es un objetivo blando

sobre un hecho que registra la historia. Cuesta lo mismo para un viaje en autobús que para cuatro, porque el último estado solo sabe si `(used-bus)` es cierto. Un turista al que le moleste *cada* viaje pondría la preferencia en la precondición de `bus`.

Para un turista que de verdad detesta los autobuses, `tourist-p02.pddl` sube el peso de `on-foot` a 6, más que cualquier lugar por separado. El mejor plan cuesta entonces 4: museo, mercado y catedral, todo a pie, y la torre queda para otro día.

Un satélite con más peticiones que tiempo

Las preferencias no están atadas a las acciones instantáneas. Un satélite de observación pasa sobre una región durante 60 segundos. En ese tiempo puede girar su cámara hacia un blanco (un `slew`, cuya duración depende del ángulo) y tomar una imagen (10 segundos, durante los cuales tiene que seguir apuntando al blanco y la región tiene que estar a la vista). Hay cinco peticiones, dos de ellas urgentes. El Listado 14.3 usa las acciones durativas del Capítulo 10, y el problema cierra la pasada con un literal inicial temporizado del Capítulo 13: `(at 60 (not (in-pass)))`.

```
;; A satellite over a region: it turns to a target and takes its
;; image, and only while the pass over the region lasts.
(define (domain satellite)
  (:requirements :strips :typing :durative-actions
                 :numeric-fluents :timed-initial-literals
                 :preferences)

  (:types
    direction - object
    target - direction
    urgent routine - target)

  (:predicates
    (pointing ?d - direction)     ; where the camera looks
    (steady)                      ; the satellite is not turning
    (in-pass)                     ; the region is in view
    (imaged ?t - target))

  (:functions
    (slew-time ?from ?to - direction))  ; seconds to turn

  (:durative-action slew
    :parameters (?from ?to - direction)
    :duration (= ?duration (slew-time ?from ?to))
    :condition (and (at start (pointing ?from))
                    (at start (steady)))
    :effect (and (at start (not (pointing ?from)))
                 (at start (not (steady)))
                 (at end (pointing ?to))
                 (at end (steady))))

  (:durative-action take-image
    :parameters (?t - target)
    :duration (= ?duration 10)
    :condition (and (at start (steady)) (over all (pointing ?t))
                    (over all (in-pass)))
    :effect (and (at start (not (steady)))
                 (at end (steady))
                 (at end (imaged ?t)))))
```

Listado 14.3: El dominio del satélite: acciones durativas dentro de una pasada que se acaba.

Los blancos tienen dos subtipos, y el objetivo tiene una preferencia por clase, cada una bajo un `forall`:

```
  (:goal (and
```

```
    (forall (?t - urgent)
      (preference urgent-done (imaged ?t)))
    (forall (?t - routine)
      (preference routine-done (imaged ?t)))))
  (:metric minimize (+ (* 5 (is-violated urgent-done))
                       (is-violated routine-done))))
```

Con tres blancos rutinarios, un peso de 5 hace de la clase urgente una prioridad estricta ($5 > 3 \times 1$).

```
$ kairos satellite-domain.pddl satellite-p01.pddl
; cost = 13
; makespan = 0.000
; better plan 2
; cost = 3
; makespan = 47.003
0.000: (slew nadir fire) [12.000]
12.001: (take-image fire) [10.000]
22.002: (slew fire flood) [15.000]
37.003: (take-image flood) [10.000]
; better plan 3
; cost = 2
; makespan = 57.005
0.000: (slew nadir fire) [12.000]
12.001: (take-image fire) [10.000]
22.002: (slew fire harbor) [9.000]
31.003: (take-image harbor) [10.000]
41.004: (slew harbor flood) [6.000]
47.005: (take-image flood) [10.000]
```

El segundo plan toma imágenes de los dos blancos urgentes y se detiene a los 47 segundos sin nada más a su alcance. El tercero es más astuto: ir de `fire` a `flood` *pasando por* `harbor` lleva los mismos 15 segundos de giro que ir directamente, y la parada intermedia deja sitio para una imagen rutinaria (Figura 14.5). Dos peticiones rutinarias quedan sin atender, y el coste es 2. El coste y el *makespan* son ahora números distintos: la métrica cuenta violaciones, no tiempo.

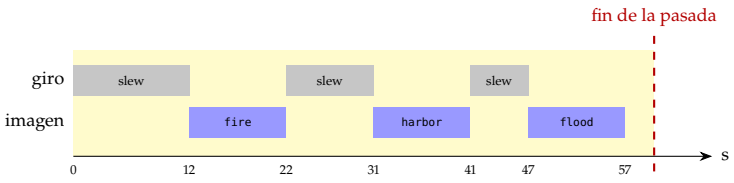


Figura 14.5.: El mejor plan para el satélite: tres imágenes dentro de los 60 segundos de la pasada. La imagen rutinaria de `harbor` cabe porque el puerto está en el camino de un blanco urgente al otro.

14.8. Qué hace el planificador con una preferencia

Un planificador clásico busca un estado que cumpla el objetivo. Una preferencia no forma parte de esa pregunta, así que los planificadores reescriben el problema en otro que sí saben buscar. La reescritura habitual, debida a Keyder y Geffner Keyder y Geffner, 2009, convierte un objetivo blando en acciones normales con costes (Figura 14.6):

- Una acción nueva *termina* el plan. Exige los objetivos estrictos y, una vez aplicada, ninguna acción del dominio vuelve a ser aplicable.
- Después del final, cada preferencia tiene que quedar resuelta por una de dos acciones nuevas: **recoger**, que exige que se cumpla la

condición de la preferencia y no cuesta nada, o **renunciar**, que no exige nada y cuesta el peso de la preferencia.
- El objetivo nuevo es que todas las preferencias estén resueltas.

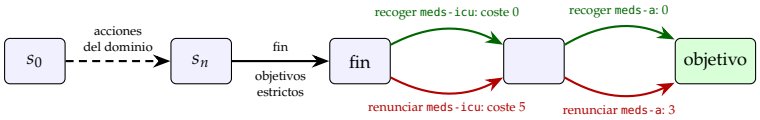


Figura 14.6.: Objetivos blandos compilados en costes. Cuando termina el plan real, cada preferencia se recoge gratis, si su condición se cumple, o se renuncia a ella al precio de su peso. El camino más barato hasta el objetivo nuevo es el mejor plan del problema original.

El resultado es un problema solo con objetivos estrictos y costes de acción, que puede resolver cualquier planificador del Capítulo 9 que tenga en cuenta los costes, y cuya heurística «ve» los deseos: un estado desde el que todavía se puede recoger `meds-icu` parece más barato que uno desde el que hay que renunciar a ella. Las acciones de recoger y renunciar se quitan del plan antes de imprimirlo.

Kairos sigue este esquema. `kairos --features` describe hasta dónde:

```
$ kairos --features
...
  preferences                full     :preferences
     compiled in goals (also under forall), at the top of
     preconditions (four per action) and over constraints; counted
     elsewhere (more, or nested), where only improving plans sees
     their cost
...
```

Dicho con palabras: las preferencias en los objetivos, solas o bajo un `forall`, se compilan como en la figura. Las preferencias escritas directamente en el `and` de una precondición también se compilan, haciendo copias de la acción (una que cumple la preferencia y otra que la viola y paga), hasta cuatro preferencias por acción. Cualquier otra preferencia (una quinta en la misma acción, o una anidada más adentro de una condición) no se compila, sino que se *cuenta*: la búsqueda no ve su coste, cada plan encontrado se valora exactamente y el bucle *anytime* se queda solo con los planes que mejoran. La respuesta es del mismo tipo; puede tardar más en ser buena. Las preferencias «sobre restricciones» son el tema del Capítulo 15.

Desde fuera se ven dos consecuencias. La búsqueda empieza por el plan más barato de *encontrar*, que con objetivos blandos suele ser el vacío, y mejora a partir de ahí. Y cuando una búsqueda secuencial termina sola, como en todos los ejemplos del mensajero de este capítulo, el último plan es óptimo para la métrica: `kairos -v` informa de `solved (proven optimal)`. En problemas temporales como el del satélite, la búsqueda termina sin esa demostración.

PDDL3 y la quinta competición internacional

Las preferencias y las restricciones de trayectoria del capítulo siguiente llegaron juntas en PDDL3, el lenguaje que Alfonso Gerevini y Derek Long definieron en 2005 para la quinta Competición Internacional de Planificación, celebrada en 2006 (A. Gerevini & Long, 2005). Hasta entonces la competición había medido la calidad del plan por su longitud, su coste o su *makespan*; PDDL3 permitió que un problema dijera qué objetivos y qué propiedades del plan eran deseables, y cuánto. El informe de aquella competición (A. E. Gerevi-

ni et al., 2009) describe el lenguaje, los dominios de prueba escritos para él y cómo se las arreglaron con ellos los planificadores de la época.

14.9. El método MODEL: los deseos y su valor

Las preferencias cambian el primer paso del método más que ningún otro. Hasta ahora, «qué cuenta como resuelto» tenía una sola respuesta. A partir de ahora, el paso M pide una lista.

Preguntas nuevas para el paso M (Mapear la situación)

- **¿Qué objetivos son obligatorios y cuáles son deseos?** Para cada condición de éxito, imagina el plan que la deja fuera: inservible (objetivo estricto) o solo peor (preferencia).
- **¿Cuánto vale cada deseo?** En qué unidad, y comparado con qué: solo con los demás deseos, o también con el esfuerzo (distancia, tiempo, dinero) del plan.
- **¿Hay algún deseo sobre el camino y no sobre el final?** Algo que no debería ocurrir durante el plan, aunque no deje rastro en el último estado.
- **¿Cuál es el plan cuando no se concede nada?** Si el plan vacío es aceptable, dilo; si no lo es, algún objetivo es estricto y falta en la lista.

La ficha del modelo gana una tabla de *Deseos* bajo *Éxito*: nombre, condición, peso y el motivo del peso.

Los pasos O y D apenas cambian: una preferencia usa los predicados y las acciones que ya están ahí. Solo se tocan cuando un deseo trata del camino y nada en el estado lo registra, como con `(used-bus)`: entonces el paso O añade el hecho, y el paso D hace que la acción lo active. El paso E gana dos pruebas propias. La primera consiste en ejecutar el problema con todos los pesos a cero o sin métrica y comprobar que el plan obtenido es el que se espera cuando nadie desea nada. La segunda es la *prueba del precio*: para cada deseo, sube su peso hasta que cambie el plan y comprueba que el peso al que cambia tiene sentido (la Figura 14.4 es una prueba de este tipo).

Alternativas comparadas

Un deseo sobre el final tiene dos codificaciones, y un deseo sobre el camino, tres. Comparamos el segundo caso con la norma del vestíbulo, con la puerta trasera abierta. Los tres modelos están en los ejemplos del capítulo, y los tres se ejecutaron sobre el mismo problema.

Precondición estricta (`courier-lobby-hard`). La condición se escribe sin `preference`: un robot con una muestra no puede entrar en el vestíbulo.

Preferencia (`courier-lobby`). El modelo de la Sección 14.6, con peso 10.

Coste de acción (`courier-lobby-cost`). Sin preferencia: `move` tiene un efecto condicional (Capítulo 6) que suma 10 a `total-cost` cuando entra en el vestíbulo con una muestra.

```
                (when (and (= ?to lobby)
                           (exists (?s - sample)
                             (carrying ?r ?s)))
                  (increase (total-cost) 10))))
```

Los tres devuelven las mismas cinco acciones por la puerta trasera, con coste 42. Se diferencian en todo lo demás (Tabla 14.3).

Tabla 14.3.: Tres codificaciones de «no cruzar el vestíbulo con una muestra», juzgadas según los cinco criterios.

	Precondición estricta	Preferencia	Coste de acción
Fidelidad	Pierde soluciones: sin la puerta trasera no hay ningún plan	Exacta: permitido, a un precio	Exacta, los mismos planes y los mismos precios
Tamaño	No crece	Una copia de `move` por preferencia compilada, más las acciones de recoger y renunciar de las preferencias del objetivo	No crece; un efecto condicional
Guía	La mejor: los desplazamientos prohibidos se podan	Buena si se compila: la heurística ve el precio	Depende del planificador: Kairos no usa un coste condicional para guiar la búsqueda
Claridad	Una norma, que se lee como una norma	El deseo y su precio están separados: el dominio dice qué se desea, y cada problema, cuánto	El precio queda enterrado en el dominio, el mismo para todos los problemas
Portabilidad	Precondiciones negativas y disyuntivas	`:preferences`: Kairos, OPTIC, POPF y ENHSP entre los planificadores probados	`:conditional-effects` y `:action-costs` juntos

La celda de Guía de la tercera columna sale de haberlo ejecutado. Kairos resuelve `courier-lobby-cost-p01.pddl`, pero añade una nota:

```
$ kairos courier-lobby-cost-domain.pddl courier-lobby-cost-p01.pddl
; cost = 42
...
note: the search minimizes the number of actions; the metric of the
     problem is reported for each plan but does not guide the search
      yet [orchestrator.metric]
```

Un coste que depende del estado no se le puede dar a la búsqueda como coste de una acción, así que la búsqueda recurre al plan más corto, que aquí resulta ser el mejor.

Veredicto. Usa un objetivo o una precondición *estrictos* cuando el plan que incumple la condición es inservible: seguridad, física, la ley. Una preferencia con un peso enorme no los sustituye: sigue permitiendo la violación, y obliga a la búsqueda a comparar números enormes con números pequeños. Usa una *preferencia* cuando incumplir la condición es aceptable a un precio, y sobre todo cuando el precio cambia de un problema a otro: los pesos viven en el fichero del problema, y el dominio se escribe una sola vez. Usa un *coste de acción* cuando lo que hay que evitar es la propia acción, siempre, sea cual sea el estado (un peaje, un ascensor lento), y no una condición bajo la que se aplica. Y cuando un deseo sobre el final solo compite con otros deseos, el objetivo blando es la única de las tres formas que puede expresarlo: ni un objetivo estricto ni un coste de acción pueden decir «entrega todos los que puedas».

14.10. Errores comunes

Preferencias sin métrica

Incumplir una preferencia que la métrica no menciona no cuesta nada. El fichero `courier-battery-nometric.pddl` es `p02` sin su `:metric`:

```
$ kairos courier-battery-domain.pddl courier-battery-nometric.
    pddl
; cost = 0
```

Un plan vacío de coste 0, y ningún aviso: el PDDL es correcto. Sin métrica, Kairos minimiza el número de acciones, y si no hay que alcanzar ningún objetivo obligatorio, el plan más corto es no hacer nada. *Corrección*: toda preferencia aparece en la métrica, o no debería estar en el modelo. Lo mismo le ocurre a una preferencia sin nombre, como `(preference (item-at meds1 ward-a))`: la sintaxis la permite, pero nada puede referirse a ella.

Un nombre en la métrica que no es de ninguna preferencia

Un error tipográfico en un nombre lo detecta `kairos check`, y esa es la razón para ejecutarlo primero:

```
$ kairos check courier-battery-domain.pddl courier-battery-typo.
    pddl
courier-battery-typo.pddl:36:24: error: no preference named '
    sample-lap' [semantic.unknown_preference]
 36 |                        (is-violated sample-lap))))
    |                        ^
invalid
```

El error contrario, un nombre mal escrito de la misma manera *a la vez* en el objetivo y en la métrica, no es un error en absoluto. Repasa los nombres una vez más.

Maximizar las violaciones

`is-violated` cuenta lo que salió mal. Escribir `maximize` porque «queremos el mayor número de entregas» pide el plan que más falla:

```
$ kairos courier-battery-domain.pddl courier-battery-maximize.
    pddl
; cost = 3
note: the search minimizes the number of actions; the metric of
    the problem is reported for each plan but does not guide
    the search yet [orchestrator.metric]
```

El plan vacío viola las tres preferencias y ya es el mejor plan posible. *Corrección*: usa `minimize` sobre una suma de violaciones. Para maximizar un beneficio, escríbelo como una constante menos las violaciones.

Quedarse con el primer plan

Con objetivos blandos, el primer plan suele ser el vacío. Si se pide un solo plan, se obtiene ese:

```
$ kairos courier-battery-domain.pddl courier-battery-p03.pddl --
    no-anytime
; cost = 9
```

Corrección: deja que la búsqueda *anytime* siga y quédate con el último plan, o dale una meta con `--target-cost`. Un sistema que llama al planificador con un límite de tiempo corto tiene que saber que lo que recibe puede estar lejos del mejor.

Pesos en la unidad equivocada

En una métrica de beneficio neto, los pesos se comparan con el coste de las acciones. Un peso de 3 para un medicamento, copiado de un problema en el que solo se comparaban deseos, significa «vale tres metros» en `courier-cost`, y el robot nunca se moverá por él. El síntoma es un plan que hace menos de lo esperado y es `proven optimal`. *Corrección*: haz la prueba del precio de la Sección 14.9, y escribe junto a cada peso en qué se mide.

14.11. Resumen

Una preferencia es una condición con nombre que un plan debería cumplir y puede incumplir. En un objetivo habla del último estado; en una precondición, de cada aplicación de una acción. La métrica da un precio a cada violación, y el planificador busca el plan que menos paga. Con eso, un problema que pide más de lo que permiten sus recursos tiene respuesta: el mejor subconjunto de lo pedido.

Construcción	Significado
`:preferences`	El requisito
`(preference p C)` en el objetivo	Objetivo blando: `C` debería cumplirse en el último estado
`(preference p C)` en una precondición	`C` debería cumplirse cada vez que se aplica la acción; nunca la bloquea
`(forall (?x) (preference p C))`	Una preferencia por objeto, todas llamadas `p`
`(is-violated p)`	En la métrica: cuántas preferencias llamadas `p` se violan (objetivo), o cuántas veces (precondición)
`(* w (is-violated p))`	Un peso: el precio de cada violación
`(+ (total-cost) ...)`	Beneficio neto: deseos frente a esfuerzo, en una sola unidad

Tabla 14.4.: Las construcciones de este capítulo.

Para el modelador, la lección está en el paso M: separar lo obligatorio de lo deseado, y dar a cada deseo un valor que se pueda defender. Las preferencias sobre el camino han abierto una puerta que el capítulo siguiente cruza: el Capítulo 15 introduce condiciones sobre todos los estados que visita un plan, estrictas o como preferencias.

14.12. Lecturas complementarias

La definición de las preferencias, con su sintaxis y el significado exacto de `is-violated`, está en el informe que presentó PDDL3 (A. Gerevini & Long, 2005); las secciones 2 y 3 cubren lo que ha usado este capítulo. El artículo sobre la quinta competición (A. E. Gerevini et al., 2009) añade

los dominios de prueba con preferencias, buenas fuentes de ideas de modelado, y los resultados de los planificadores. El libro de Haslum et al. (2019) trata las preferencias junto con las restricciones de trayectoria y discute su compilación en costes. OPTIC (Benton et al., 2012) es un planificador temporal construido en torno a las preferencias, y su artículo explica qué las hace difíciles cuando interviene el tiempo. Para ver cómo las trata Kairos, consulta su documentación (Martín Rico et al., 2026).

14.13. Ejercicios

Ejercicios de la Sección 14.1 — Más deseos que batería

Ejercicio 14.1.1. Sin el planificador, y con las distancias de la Figura 14.1, calcula la batería que necesita `r1` para entregar `meds2` en la UCI y después `sample1` en el laboratorio, partiendo de la base de carga. Compara tu número con la Figura 14.2.

Ejercicio 14.1.2. Encuentra, con Kairos, la batería inicial más pequeña con la que `courier-battery-p01.pddl` tiene plan. Antes de ejecutar nada, calcula a mano la batería que necesitan dos órdenes distintos de las tres entregas. ¿Qué orden sigue el plan de Kairos, y por qué es el barato?

Ejercicios de la Sección 14.2 — Una preferencia en el objetivo

Ejercicio 14.2.1. Supón que el objetivo de `courier-battery-p02.pddl` gana una cuarta preferencia, `(preference at-lobby (robot-at r1 lobby))`, que se añade a la métrica con peso 1: al turno de día le gustaría encontrarse el robot en el vestíbulo. ¿Cuál es el coste del plan vacío? ¿Y el del mejor plan? Responde primero y ejecútalo después.

Ejercicios de la Sección 14.3 — Lo que vale cada deseo

Ejercicio 14.3.1. Con ayuda de la Tabla 14.1, elige pesos para los tres deseos de `p03` de modo que el mejor plan entregue `meds1` y `sample1` y renuncie a la UCI. Compruébalo con Kairos.

Ejercicio 14.3.2. Un hospital tiene tres clases de entregas: 2 urgentes, 4 normales y 10 opcionales. Da un peso a cada clase de modo que las clases sean prioridades estrictas: ningún número de entregas opcionales vale lo que una entrega normal, y ningún número de normales y opcionales vale lo que una urgente.

Ejercicio 14.3.3. En `courier-battery-p04.pddl` el deseo `parked` tiene peso 4. ¿Con qué pesos termina en la base de carga el mejor plan? Razona con la tabla de la sección y prueba después un peso a cada lado de tu respuesta.

Ejercicios de la Sección 14.4 — Deseos frente a esfuerzo

Ejercicio 14.4.1. En `courier-cost-p01.pddl` la comida vale 20 y no se entrega; en `p02` vale 100 y sí se entrega. Encuentra el peso exacto en el que el planificador cambia de opinión, y explica qué devuelve Kairos cuando el peso es justo ese.

Ejercicios de la Sección 14.5 — Una preferencia, muchas instancias

Ejercicio 14.5.1. En `courier-cost-p03.pddl`, sube el peso de `meds-icu` a 70. Predice el coste del mejor plan y el valor de `(is-violated meds-icu)` en él; después, ejecútalo.

Ejercicio 14.5.2. Reescribe el objetivo de `courier-cost-p03.pddl` con el `forall` dentro de la preferencia. Sin ejecutarlo: ¿cuál es la métrica del plan que entrega un medicamento? ¿Qué planes pueden ser óptimos ahora?

Ejercicios de la Sección 14.6 — Deseos sobre el camino

Ejercicio 14.6.1. Según la Figura 14.4, las dos rutas empatan cuando el peso de `clean-lobby` es 6. Ejecuta `courier-lobby-p01.pddl` con pesos 6 y 7 y explica las dos salidas.

Ejercicio 14.6.2. El turno de noche. Los pacientes duermen en la sala A y en la sala B, y el hospital preferiría que los robots no entraran en esas salas de noche, carguen algo o no. Escribe un dominio `courier-night` a partir de `courier-cost`, con un predicado estático `(quiet ?l - location)` y una preferencia en `move`. Escribe después un problema en el que `r1` tenga que llevar `meds2` de la farmacia a la UCI y volver a la base de carga, con un precio de 25 metros por cada entrada en una sala tranquila. ¿Cuántas veces se viola la preferencia en el mejor plan, y por qué no pueden ser menos?

Ejercicio 14.6.3. Un estudiante modela la norma del vestíbulo como un objetivo blando: `(preference clean-lobby (not (robot-at r1 lobby)))`. Explica qué significa este deseo y por qué no expresa la norma.

Ejercicios de la Sección 14.7 — Otros dos mundos

Ejercicio 14.7.1. En `tourist-p01.pddl`, estar de vuelta en el hotel es un objetivo estricto. Conviértelo en una preferencia `dinner` con peso 1 y ejecuta el problema. ¿Qué hace ahora el turista, y qué dice eso del peso?

Ejercicio 14.7.2. Amplía `satellite-p01.pddl`: la pasada dura 80 segundos y hay un cuarto blanco rutinario, `volcano`, a 5 segundos de giro de cualquier otra dirección. ¿Cuántas peticiones quedan sin atender en el mejor plan que encuentra Kairos?

Ejercicios de la Sección 14.9 — El método MODEL: los deseos y su valor

Ejercicio 14.9.1. Aplica el paso L al deseo «terminar la mañana en la base de carga» de `courier-battery-p04.pddl`. Compara tres codificaciones según los cinco criterios: un objetivo estricto, la preferencia del capítulo, y ningún objetivo, con un robot al que simplemente se manda a casa con un segundo problema después. ¿Cuándo es adecuada cada una?

Ejercicio 14.9.2. El paso M a partir de una historia. «Esta noche tengo tres horas. El informe de laboratorio hay que entregarlo mañana (70 minutos). Debería hacer los ejercicios de álgebra (50), me gustaría leer el capítulo para el viernes (40), ir al gimnasio (45) y poner la lavadora (15). Después del gimnasio estoy cansado, y cuando

estoy cansado trabajo mal, salvo que descanse 20 minutos.» Escribe la parte de *Éxito* de la ficha del modelo con su tabla de deseos, y después un dominio y un problema. Deja que el planificador decida la tarde y lee su plan.

15. Restricciones sobre todo el plan

Un objetivo juzga un plan por su último estado, y una precondición juzga una acción en el momento en que se aplica. Muchas reglas de un lugar de trabajo real no son ni lo uno ni lo otro. «Nada de residuos en la UCI» no trata de dónde acaban las cosas: los residuos acaban en la base de carga en cualquier caso. «Desinfectarse antes de entrar en el laboratorio» no trata de una acción: relaciona algo que ocurre ahora con algo que tiene que haber ocurrido antes. «Pasar por la base de carga como mucho una vez» lleva la cuenta. La preferencia en una precondición del Capítulo 14 miró por primera vez el camino, pero solo a través de la ventana de una única acción. Un hospital que entrega su reglamento al modelador espera que todo plan lo respete del primer paso al último.

PDDL 3.0 tiene una sección para el reglamento: `:constraints`. Una **restricción de trayectoria** es una condición sobre la secuencia completa de estados por la que pasa un plan, escrita con uno de diez operadores, desde `always` y `sometime` hasta plazos como `within` y ventanas como `hold-during`. Un plan que alcanza el objetivo e incumple una restricción no es un plan. Y una restricción envuelta en una preferencia se convierte en una regla que se puede incumplir pagando un precio.

El capítulo empieza con un plan que incumple una regla que el modelo no tenía forma de expresar. Después define la secuencia de estados de un plan, sobre la que descansa todo lo demás, y presenta los operadores en tres grupos: lo que debe ser cierto siempre o alguna vez, lo que debe ocurrir en cierto orden y lo que debe ocurrir antes de cierto instante. Una sección pone los diez uno junto a otro, cada uno con su significado exacto, con un plan que lo cumple y otro que lo viola. Vienen luego las restricciones en el dominio y en el problema, las restricciones como preferencias, un róver y un taller, qué hace un planificador con una restricción y la comparativa que pide el método MODEL: una restricción, precondiciones en todas las acciones o un autómata escrito a mano.

15.1 Una regla que el plan incumple 345
15.2 Los estados de un plan: always 347
15.3 Lo que debe ocurrir, y en qué orden 348
sometime y at end . . . 348
at-most-once 349
sometime-before y sometime-after 349
15.4 Reglas con reloj 350
Contar pasos 351
Contar segundos 352
15.5 Los diez operadores, uno junto a otro 353
15.6 Dónde viven las reglas, y reglas que ceden 355
En el dominio o en el problema 355
Restricciones como preferencias 355
15.7 Dos mundos más 356
Un róver y un cráter . . 356
Un taller con un horno . 357
15.8 Qué hace el planificador con una restricción . . . 358
15.9 El método MODEL: reglas sobre el plan completo 359
Alternativas comparadas 360
15.10 Errores comunes 361
15.11 Resumen 363
15.12 Lecturas complementarias 363
15.13 Ejercicios 364

15.1. Una regla que el plan incumple

El mensajero de este capítulo tiene una cesta. Puede llevar cualquier número de artículos, así que el predicado `(free ?r)` del dominio con tipos desaparece y `pick` pierde esa precondición. Los artículos tienen tres subtipos, `medicine`, `sample` y `waste`. El robot también puede desinfectarse la pinza donde haya un dispensador de gel, y coger cualquier cosa se la vuelve a ensuciar. El Listado 15.1 es el dominio completo, `courier-rules`: STRIPS clásico con tipos. Declara `:constraints` porque sus problemas las usarán.

```
;; The courier world for a chapter about rules. The robot has a
;; basket: it carries any number of items, so (free ?r) is gone.
;; It can sanitize its gripper; picking an item dirties it again.
(define (domain courier-rules)
  (:requirements :strips :typing :constraints)

  (:types
    robot location item - object
    medicine sample waste - item)

```

```
  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (has-sanitizer ?l - location)   ; a gel dispenser is here
    (sanitized ?r - robot))         ; the gripper is clean

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to)))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (item-at ?i ?l))
    :effect (and (not (item-at ?i ?l)) (not (sanitized ?r))
                 (carrying ?r ?i)))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l)))

  (:action sanitize
    :parameters (?r - robot ?l - location)
    :precondition (and (robot-at ?r ?l) (has-sanitizer ?l))
    :effect (sanitized ?r)))
```

Listado 15.1: El dominio del capítulo: un mensajero con una cesta y un dispensador de gel. Nada en él menciona una regla.

En `courier-rules-p01.pddl` una caja de medicamentos y una bolsa de residuos esperan en la farmacia. Los medicamentos van a la UCI, los residuos a la base de carga, y el robot debe terminar en la base de carga.

```
$ kairos courier-rules-domain.pddl courier-rules-p01.pddl
; cost = 14
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 waste1 pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
(move r1 icu ward-b)
(move r1 ward-b corridor)
(move r1 corridor lobby)
(move r1 lobby dock)
(drop r1 waste1 dock)
```

Catorce acciones, el plan más corto que existe, y ningún hospital lo aceptaría: el robot lo carga todo de una vez y entra en la unidad de cuidados intensivos con una bolsa de residuos en la cesta. El modelo no tiene ningún fallo que el paso E hubiera podido encontrar. Todas las acciones son legales, se alcanza el objetivo y no se ha supuesto nada falso. Lo que falta es una regla del hospital, *ningún robot está nunca en la UCI mientras lleva residuos*, y la pregunta es dónde se puede escribir una regla así.

En el objetivo no: en el último estado los residuos están en la base de carga. Una precondición de `move` podría prohibir entrar en la UCI con residuos, pero los residuos también se pueden recoger *dentro* de la UCI, así que `pick` necesita su propia precondición, y lo mismo cada acción

que añada una versión posterior del dominio. La regla es una sola frase sobre estados, y quedaría repartida entre todas las acciones que pudieran producir uno de esos estados.

Algunas reglas no tratan del final del plan ni de una acción: tratan de cada estado del camino. Los objetivos y las precondiciones solo pueden expresarlas a trozos.

15.2. Los estados de un plan: always

Un plan de n acciones, aplicado desde el estado inicial, pasa por $n + 1$ estados:

$$S_0, S_1, S_2, \ldots, S_n$$

donde S_0 es el estado inicial y S_i es el estado tras la i-ésima acción. Esta secuencia es la **trayectoria** del plan. Fíjate en que incluye los dos extremos: el estado inicial forma parte de la trayectoria, y el último también.

El plan anterior tiene 14 acciones y una trayectoria de 15 estados. La Figura 15.1 dibuja dos hechos a lo largo de ella: si el robot está en la UCI y si lleva los residuos. La regla del hospital se incumple en los estados en los que los dos son ciertos, S_8 y S_9.

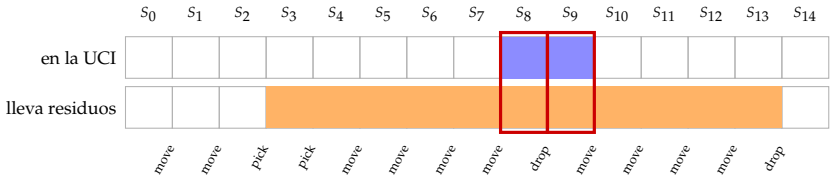


Figura 15.1.: La trayectoria del plan de 14 acciones. Cada columna es un estado; una celda rellena indica que el hecho se cumple en él. Un objetivo solo mira S_{14}. La regla «nunca en la UCI con residuos» mira todas las columnas, y falla en las dos enmarcadas en rojo.

Una **restricción de trayectoria** es una condición sobre la trayectoria. La más sencilla dice que una condición se cumple en todos los estados: `(always C)`. Las restricciones van en una sección propia, `:constraints`, que en un problema se escribe después del objetivo. El requisito también se llama `:constraints`. `courier-rules-p02.pddl` es `p01` más la regla:

```
(:goal (and (item-at meds1 icu) (item-at waste1 dock)
            (robot-at r1 dock)))
(:constraints
  (forall (?r - robot ?w - waste)
    (always (not (and (robot-at ?r icu) (carrying ?r ?w))))))
```

El `forall` crea una restricción por cada robot y cada artículo de tipo `waste`; una sección con varias restricciones las une con `and`. La condición dentro de `always` es una condición corriente, de las que se usan en los objetivos: aquí, una conjunción negada, y por eso el problema declara también `:negative-preconditions` y `:universal-preconditions`.

```
$ kairos courier-rules-domain.pddl courier-rules-p02.pddl
; cost = 16
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 waste1 pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby dock)
(drop r1 waste1 dock)
(move r1 dock lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
(move r1 icu ward-b)
```

```
(move r1 ward-b corridor)
(move r1 corridor lobby)
(move r1 lobby dock)
```

El robot sigue cargando los dos artículos en la farmacia, pero ahora se deshace de los residuos en la base de carga antes de subir a la UCI. El plan tiene dos acciones más: ese es el coste de la regla. El dominio no ha cambiado ni un carácter.

El validador también comprueba las restricciones. El plan corto de `p01`, guardado como `courier-rules-p02-bad.plan`, es válido para `p01` y no lo es para `p02`:

```
$ kairos validate courier-rules-domain.pddl courier-rules-p02.pddl
     courier-rules-p02-bad.plan
courier-rules-p02-bad.plan: error: the plan violates the problem :
     constraints [validate.constraint_violated]
plan invalid
```

La trayectoria de un plan es la secuencia de sus estados, incluido el inicial. Una restricción always debe cumplirse en todos ellos.

Como S_0 pertenece a la trayectoria, un `always` que es falso en el estado inicial no puede cumplirse nunca: el problema es irresoluble antes de la primera acción. Cuando un problema con restricciones no tiene plan, el estado inicial es el primer sitio donde mirar.

15.3. Lo que debe ocurrir, y en qué orden

`always` es el operador de las prohibiciones. Otros cuatro hablan de cosas que deben ocurrir en algún momento, de cuántas veces y en qué orden. Cada uno se muestra aquí sobre un problema pequeño de `courier-rules`; los planes que cumplen y violan cada uno se reúnen en la Sección 15.5.

sometime y at end

`(sometime C)` pide que `C` se cumpla en al menos un estado de la trayectoria. En `courier-rules-p03.pddl` el robot recoge residuos de la sala B, y la enfermera de la UCI quiere verlo pasar durante la ronda:

```
(:goal (item-at waste1 dock))
(:constraints
  (sometime (robot-at r1 icu))))
```

El plan de Kairos tiene 10 acciones en lugar de 8: por el camino entra en la UCI y vuelve a salir. Un `sometime` es un objetivo que no tiene por qué durar. Ningún objetivo podría decir esto: al final el robot está en la base de carga, no en la UCI.

`(at end C)` pide que `C` se cumpla en el último estado. Eso es lo que hace un objetivo, y el operador existe para que objetivos y restricciones puedan escribirse, y preferirse, en la misma sección. Una condición escrita en `:constraints` sin ningún operador significa lo mismo.

at-most-once

`(at-most-once C)` pide que `C` se haga cierta como mucho una vez: puede cumplirse durante un tramo de estados consecutivos, de cualquier longitud, y nunca más cuando ese tramo termina. También puede no cumplirse nunca. La puerta de la base de carga es lenta y ruidosa, y los técnicos quieren que se use una vez por ronda. En `courier-rules-p04.pddl` el robot empieza en el vestíbulo, hay residuos en la sala A que esperan para ir a la base de carga y medicamentos en la base de carga que esperan para ir a la sala A:

```
  (:constraints
    (at-most-once (robot-at r1 dock))))
```

El plan corto iría primero a por los medicamentos (la base de carga está a un paso), los entregaría y volvería con los residuos: 11 acciones y dos visitas. Con la restricción, Kairos devuelve 12 acciones: el robot va a la sala A con la cesta vacía, trae los residuos y, en una sola visita a la base de carga, los deja y recoge los medicamentos.

El tramo puede ser largo. El robot se queda en la base de carga durante tres estados (llega, deja, recoge) y eso es una sola visita. Lo que está prohibido es irse y volver.

sometime-before y sometime-after

Dos operadores relacionan dos condiciones.

`(sometime-before A B)`: siempre que `A` se cumple en un estado, `B` se cumplió en algún estado *anterior*. «Antes de hacer `A`, tienes que haber hecho `B`». Si `A` no se cumple nunca, no se pide nada.

`(sometime-after A B)`: siempre que `A` se cumple en un estado, `B` se cumple en ese estado o en alguno *posterior*. «Si haces `A`, tienes que hacer `B` después».

El laboratorio pide que los robots se desinfecten antes de entrar. El problema `courier-rules-p05.pddl` envía una muestra de la sala A al laboratorio con esa regla:

```
  (:goal (item-at sample1 lab))
  (:constraints
    (sometime-before (robot-at r1 lab) (sanitized r1))))
```

```
$ kairos courier-rules-domain.pddl courier-rules-p05.pddl
; cost = 8
(move r1 dock lobby)
(move r1 lobby corridor)
(sanitize r1 corridor)
(move r1 corridor ward-a)
(pick r1 sample1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

El plan tiene la acción `sanitize` que la regla obligaba a poner. Pero fíjate en dónde está. El robot se desinfecta en el pasillo a la *ida*, después coge la muestra, lo que vuelve a ensuciar la pinza, y entra en el laboratorio con la pinza sucia. La restricción se cumple al pie de la letra: `(sanitized r1)`

se cumplió en un estado anterior al primer estado en el laboratorio. Lo que está mal escrito es la regla.

El paso E: la prueba de la trampa para una regla

La cuarta prueba del paso E tienta al planificador a hacer trampa, y una regla es una ocasión más para hacerla. Para cada restricción, busca la forma más barata de cumplir su letra sin hacer lo que se quería decir. `sometime-before` pide «en algún momento anterior», por lejano que sea y pase lo que pase entre medias. Si el hospital quiere decir «limpio *al* entrar», la regla trata de un único estado, y el operador para eso es `always`: `(always (imply (robot-at r1 lab) (sanitized r1)))`. El Ejercicio 15.3.3 lo ejecuta.

`sometime-after` es la imagen especular. En `courier-rules-p06.pddl`, un robot que ha estado en la UCI debe ir después a la base de carga para que lo desinfecten:

```
(:goal (item-at meds1 icu))
(:constraints
  (sometime-after (robot-at r1 icu) (robot-at r1 dock))))
```

El objetivo se alcanza con el `drop` en la UCI, tras 7 acciones. El plan de Kairos tiene 11: cuatro movimientos más llevan al robot a la base de carga, aunque ningún objetivo lo pide. La restricción ha creado en mitad del plan una obligación que el plan debe saldar antes de poder terminar.

sometime: al menos una vez. at-most-once: un tramo como mucho. sometime-before: solo después de que haya ocurrido B. sometime-after: si ocurre A, después viene B.

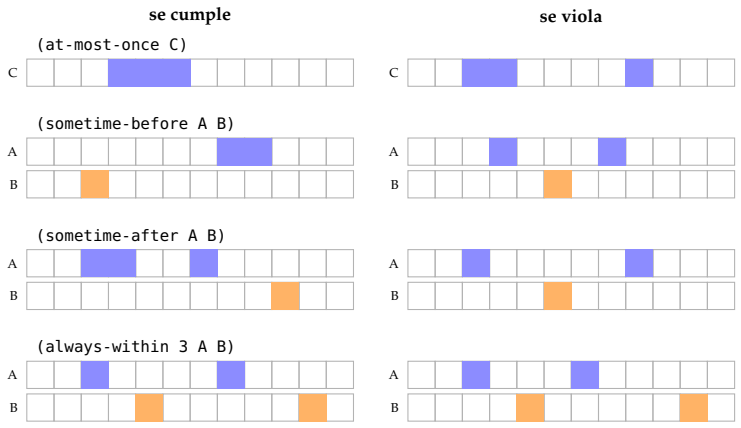


Figura 15.2.: Cuatro operadores como patrones sobre una trayectoria de 12 estados (el tiempo avanza hacia la derecha; una celda rellena indica que la condición se cumple). A la derecha: `C` vuelve tras una interrupción; la primera `A` no tiene ninguna `B` antes; la última `A` no tiene ninguna `B` después; la segunda `A` espera 4 pasos a su `B`.

15.4. Reglas con reloj

Los cuatro operadores restantes llevan números: plazos y ventanas. Para darles significado, cada estado de la trayectoria necesita un instante. Escribimos t_i para el instante en que empieza el estado S_i, con $t_0 = 0$.

En un plan temporal (Capítulo 10) los estados cambian en los acontecimientos, los inicios y finales de las acciones durativas, y t_i es el instante del i-ésimo acontecimiento, en la unidad de las duraciones. En un plan secuencial no hay reloj, y Kairos usa la posición en el plan: el estado S_i está en el instante i. Un plazo de 7 en un plan secuencial significa «dentro de las 7 primeras acciones».

- `(within t C)`: `C` se cumple en algún estado de instante t o anterior. Un plazo.

- `(always-within t A B)`: siempre que `A` se cumple en un estado, `B` se cumple en ese estado o en uno posterior que esté como mucho t después. Un plazo que empieza a correr cada vez que `A` es cierta.
- `(hold-during t1 t2 C)`: `C` se cumple en todos los estados cuyo instante es mayor o igual que t_1 y menor que t_2. Una ventana.
- `(hold-after t C)`: `C` se cumple en todos los estados cuyo instante es posterior a t. Una ventana que nunca se cierra.

Contar pasos

Cuatro problemas de `courier-rules` los usan, con el tiempo medido en pasos.

`p07`: los medicamentos deben estar en la sala A en 7 pasos como mucho, `(within 7 (item-at meds1 ward-a))`, mientras que una bolsa de residuos que está en el vestíbulo también tiene que ir a la base de carga. Siete pasos es exactamente lo que necesita la entrega desde la base de carga, así que el robot debe dejar los residuos para después, aunque pasa junto a ellos en el primer paso.

`p08`: una muestra que se ha recogido debe estar en el laboratorio como mucho 3 pasos después, `(always-within 3 (carrying r1 sample1) (item-at sample1 lab))`. Desde la sala A eso permite los dos movimientos y dejarla, y nada más: ninguna parada por el camino.

`p09`: el pasillo se friega del paso 3 al paso 6, `(hold-during 3 6 (not (robot-at r1 corridor)))`, y los medicamentos tienen que ir de la farmacia a la sala A, pasando por el pasillo.

```
$ kairos courier-rules-domain.pddl courier-rules-p09.pddl
; cost = 9
(move r1 dock lobby)
(move r1 lobby pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

El plan directo tiene 7 acciones y está en el pasillo en el paso 5 (Figura 15.3). Un plan secuencial no puede esperar, porque el tiempo solo pasa cuando se aplica una acción. Así que el planificador *inventa* una espera: la tercera y la cuarta acción van al vestíbulo y vuelven a la farmacia, dos pasos que no hacen otra cosa que dejar correr el reloj. Después el robot entra en el pasillo en el paso 7, cuando la ventana ya se ha cerrado.

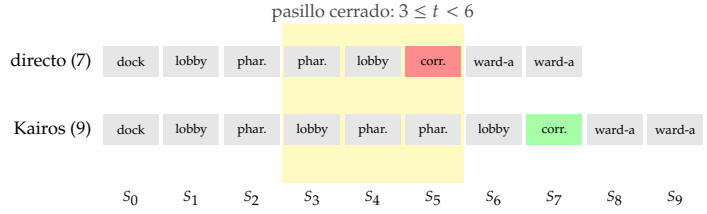


Figura 15.3.: Dónde está el robot en cada estado de dos planes para `courier-rules-p09`. La ventana cubre los estados de los instantes 3, 4 y 5. El plan directo está en el pasillo en S_5; el plan de Kairos malgasta dos movimientos y entra en él en S_7.

`p10`: dos bolsas de residuos en el vestíbulo y un toque de queda: a partir del paso 4 el robot está en la base de carga, `(hold-after 4 (robot-at r1 dock))`. Los estados posteriores al instante 4 son S_5, S_6, etc. Llevar las

bolsas de una en una devuelve al robot al vestíbulo en el paso 5; el plan de Kairos lleva las dos bolsas en un solo viaje, seis acciones, y está en la base de carga desde el paso 4.

Contar segundos

Con acciones durativas, los números son instantes del reloj. El dominio `courier-timed` es el mismo mensajero con las acciones durativas del Capítulo 10: un movimiento dura su distancia en segundos, coger o dejar algo dura 2 segundos, y `(idle ?r)` limita al robot a una acción cada vez. `courier-timed-p01.pddl` pide los dos recados de `p07` con un plazo en el reloj:

```
  (:goal (and (item-at meds1 ward-a) (item-at waste1 dock)))
  (:constraints
    (within 45 (item-at meds1 ward-a)))
  (:metric minimize (total-time)))
```

```
$ kairos courier-timed-domain.pddl courier-timed-p01.pddl
; cost = 73.004
; makespan = 73.004
0.000: (move r1 dock lobby) [4.000]
4.001: (pick r1 waste1 lobby) [2.000]
6.002: (move r1 lobby pharmacy) [6.000]
12.003: (pick r1 meds1 pharmacy) [2.000]
14.004: (move r1 pharmacy lobby) [6.000]
20.005: (move r1 lobby corridor) [10.000]
30.006: (move r1 corridor ward-a) [12.000]
42.007: (drop r1 meds1 ward-a) [2.000]
45.001: (move r1 ward-a corridor) [12.000]
57.002: (move r1 corridor lobby) [10.000]
67.003: (move r1 lobby dock) [4.000]
71.004: (drop r1 waste1 dock) [2.000]
```

Los medicamentos están en la sala A en el instante 44,007, cuando termina el `drop`, dentro del plazo. Esta vez el robot sí recoge los residuos por el camino: le cuesta 2 segundos, y el plazo de 45 deja margen para ellos. Llevar primero los residuos a la base de carga entregaría los medicamentos en el segundo 54; ese plan está en `courier-timed-waste-first.plan`, y `kairos validate` lo rechaza.

Una ventana en segundos se escribe igual. En `courier-timed-p02.pddl` el pasillo está cerrado a los robots del segundo 20 al segundo 40, `(hold-during 20 40 (not (robot-at r1 corridor)))`. El mejor plan que encuentra Kairos lleva primero los residuos a la base de carga y entra en el pasillo después de la ventana:

```
$ kairos courier-timed-domain.pddl courier-timed-p02.pddl
...
; makespan = 64.003
0.000: (move r1 dock lobby) [4.000]
4.001: (pick r1 waste1 lobby) [2.000]
6.002: (move r1 lobby dock) [4.000]
10.003: (drop r1 waste1 dock) [2.000]
12.004: (move r1 dock lobby) [4.000]
20.001: (move r1 lobby pharmacy) [6.000]
26.002: (pick r1 meds1 pharmacy) [2.000]
28.003: (move r1 pharmacy lobby) [6.000]
40.001: (move r1 lobby corridor) [10.000]
50.002: (move r1 corridor ward-a) [12.000]
62.003: (drop r1 meds1 ward-a) [2.000]
```

Un plan temporal puede esperar: el robot simplemente se queda parado en el vestíbulo entre 34 y 40. El plan es válido, pero no es el más rápido: el de `courier-timed-waste-first.plan` hace los mismos recados sin las pausas, en 54 segundos, y también es válido. El Capítulo 10 ya advertía de que el *makespan* de un plan temporal es bueno, no mínimo demostrado.

En un plan secuencial, el instante de un estado es su posición; en un plan temporal, el reloj. within es un plazo; always-within, un plazo que vuelve a empezar; hold-during y hold-after son ventanas.

El Capítulo 13 modeló un plazo con un literal inicial temporizado: un hecho que deja de ser cierto al vencer el plazo y que exige la acción de entrega. `within` dice lo mismo en una línea y sin tocar el dominio, y además funciona en planes secuenciales, que no tienen un reloj al que pueda atenerse un literal temporizado. El literal temporizado lo admiten más planificadores. La Sección 15.10 muestra un caso en el que los dos no se comportan igual.

15.5. Los diez operadores, uno junto a otro

La Tabla 15.1 da el significado de los diez operadores sobre una trayectoria $S_0, \ldots, S_n$ cuyos estados empiezan en los instantes $t_0 = 0 \le t_1 \le \ldots \le t_n$. Escribimos $S_i \models C$ para «la condición `C` se cumple en el estado S_i».

Restricción	Se cumple en la trayectoria cuando
`(at end C)`	$S_n \models C$
`(always C)`	$S_i \models C$ para todo i
`(sometime C)`	$S_i \models C$ para algún i
`(at-most-once C)`	si $S_i \models C$, existe $j \ge i$ tal que C se cumple en $S_i, \ldots, S_j$ y en ningún estado posterior a S_j
`(sometime-before A B)`	para todo i con $S_i \models A$ existe $j < i$ con $S_j \models B$
`(sometime-after A B)`	para todo i con $S_i \models A$ existe $j \ge i$ con $S_j \models B$
`(within t C)`	$S_i \models C$ para algún i con $t_i \le t$
`(always-within t A B)`	para todo i con $S_i \models A$ existe $j \ge i$ con $S_j \models B$ y $t_j - t_i \le t$
`(hold-during t1 t2 C)`	$S_i \models C$ para todo i con $t_1 \le t_i < t_2$
`(hold-after t C)`	$S_i \models C$ para todo i con $t_i > t$

Tabla 15.1.: Las restricciones de trayectoria de PDDL 3.0 como condiciones sobre la secuencia de estados. Las dos últimas filas se enuncian tal como las comprueba Kairos, sobre los estados que empiezan dentro de la ventana; la Sección 15.10 trata los bordes.

Hay tres detalles de la tabla que se pasan por alto con facilidad. En `sometime-before`, el índice j es *estrictamente* menor que i: una `B` en el mismo estado que `A` no está antes. En `sometime-after` y `always-within` es $j \ge i$: basta con `B` en el mismo estado. Y todo operador que empieza por «si se cumple `A`» lo cumple un plan en el que `A` no se cumple nunca, incluido el plan vacío.

La Tabla 15.2 es la misma lista convertida en experimento. Cada fila es un problema de `courier-rules` con una restricción y dos planes escritos a mano: `kairos validate` acepta todos los ficheros `-ok.plan` y rechaza todos los ficheros `-bad.plan` como planes que violan las restricciones del problema. El plan malo es siempre un plan válido del mismo problema sin su restricción.

Problema y restricción	`-ok.plan`	`-bad.plan`
`p02` `always` no en la UCI con residuos	deja los residuos en la base de carga y luego va a la UCI	entra en la UCI con los dos artículos (S_8, S_9)
`p03` `sometime` en la UCI	entra en la UCI (S_4)	recoge los residuos y vuelve
`p04` `at-most-once` en la base de carga	una visita (de S_6 a S_8)	dos visitas (S_1, S_2 y S_{10}, S_{11})
`p05` `sometime-before` laboratorio, desinfectado	desinfectado en S_6, en el laboratorio en S_7	nunca desinfectado
`p06` `sometime-after` UCI, base de carga	termina en la base de carga (S_{11})	termina en la UCI
`p07` `within 7` medicamentos en la sala A	entregados en S_7	entregados en S_{11}
`p08` `always-within 3` lleva, en el laboratorio	recogida en S_4, en el laboratorio en S_7	se para a desinfectarse: de S_4 a S_8
`p09` `hold-during 3 6` no en el pasillo	entra en el pasillo en S_7	en el pasillo en S_5
`p10` `hold-after 4` en la base de carga	en la base de carga desde S_4	de vuelta en el vestíbulo en S_5

Tabla 15.2.: Un problema por operador, cada uno con un plan que cumple su restricción y un plan que la viola. Los ficheros son `courier-rules-pNN.pddl`, `courier-rules-pNN-ok.plan` y `courier-rules-pNN-bad.plan`; los 18 planes están en las pruebas del capítulo. (`at end` es el objetivo).

```
$ kairos validate courier-rules-domain.pddl courier-rules-p04.pddl
     courier-rules-p04-ok.plan
plan valid, makespan 12
$ kairos validate courier-rules-domain.pddl courier-rules-p04.pddl
     courier-rules-p04-bad.plan
courier-rules-p04-bad.plan: error: the plan violates the problem :
     constraints [validate.constraint_violated]
plan invalid
```

El mensaje nombra la sección, no la restricción. Con varias restricciones en un problema, la forma de encontrar la incumplida es validar el plan frente a copias del problema que conserven una restricción cada una.

Lo que los operadores no pueden decir

Los diez operadores son un vocabulario pequeño y fijo. En PDDL 3.0 no se pueden anidar: no existe `(always (sometime C))`, y las condiciones que llevan dentro son condiciones corrientes sobre un estado. Así que «*cada vez* que el robot toca residuos, se desinfecta antes de la *siguiente* vez que entra en el laboratorio» no tiene traducción directa: `sometime-before` olvida lo que pasó entre medias, como mostró `p05`. Estas reglas se escriben añadiendo un hecho que recuerda (la pinza está `(sanitized r1)` o no) y restringiendo ese hecho con `always`. Los operadores son, de hecho, un fragmento de la *lógica temporal lineal*, la lógica que Amir Pnueli propuso en 1977 Pnueli, 1977 para enunciar propiedades de los programas sobre sus ejecuciones; la lógica completa permite el anidamiento que PDDL 3.0 deja fuera.

15.6. Dónde viven las reglas, y reglas que ceden

En el dominio o en el problema

Una sección `:constraints` puede escribirse en el problema, como hasta ahora, o en el dominio, después de los predicados y antes de las acciones. El significado es el mismo; el alcance, no. Una restricción del dominio vale para todos los problemas de ese dominio. Un plan debe cumplir las dos secciones.

La regla de la UCI no es la regla de hoy: es la del hospital. `courier-safe` es `courier-rules` con la regla trasladada al dominio. Para nombrar allí la UCI, el dominio la declara como constante:

```
(:constants icu - location)
...
(:constraints
  (forall (?r - robot ?w - waste)
    (always (not (and (robot-at ?r icu) (carrying ?r ?w))))))
```

`courier-safe-p01.pddl` solo enuncia los recados de `p02` y obtiene el mismo plan de 16 acciones. Quien escriba el próximo problema no puede olvidar la regla.

Ese es también el peligro. Una regla del dominio se aplica a problemas que su autor no imaginó. El paso E pide una prueba que debe tener plan, y aquí hay una que debería tenerlo: `courier-safe-p02.pddl` es la misma mañana con la bolsa de residuos tirada en la propia UCI.

```
$ kairos courier-safe-domain.pddl courier-safe-p02.pddl
note: no plan: every state reachable from the initial one was
     searched (278 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

La regla dice que ningún robot está nunca en la UCI llevando residuos, y recoger la bolsa en la UCI produce justo ese estado. Leída al pie de la letra, la regla prohíbe limpiar la UCI. El hospital quería decir «no lleves residuos *a* la UCI», que es otra frase, y encontrar la diferencia era precisamente el objeto de la prueba.

El paso M: de quién es la regla

Para cada regla de la historia, la ficha del modelo registra su **alcance**: una ley del mundo (nada puede incumplirla: va en precondiciones y efectos), una norma de la organización (todos los problemas: una restricción del dominio) o una condición de hoy (este problema: una restricción del problema). Y su **fuerza**: ¿debe cumplirse, o convendría que se cumpliera?

Restricciones como preferencias

Una restricción se puede envolver en la `preference` del Capítulo 14. Se convierte entonces en una **restricción blanda**: un plan puede violarla, y `(is-violated name)` vale 1 en la métrica si la trayectoria no la cumple y 0 si la cumple. Bajo un `forall`, cuenta cada instancia.

`courier-rules-p11.pddl` tiene la mañana de `p02` con la regla estricta de la UCI y dos deseos sobre el tiempo que tiran en direcciones opuestas:

```
(:constraints
  (and
    (forall (?r - robot ?w - waste)
      (always (not (and (robot-at ?r icu) (carrying ?r ?w)))))
    (preference fast-meds (within 8 (item-at meds1 icu)))
    (preference early-waste (within 6 (item-at waste1 dock)))))
(:metric minimize (+ (* 3 (is-violated fast-meds))
                     (* 2 (is-violated early-waste)))))
```

Ocho pasos es lo que necesitan los medicamentos si el robot va directo a la UCI; seis, lo que necesitan los residuos si va directo a la base de carga. Los dos no pueden ir primero. Kairos imprime tres planes, de coste 5, 3 y 2. El último entrega primero los medicamentos, en 8 pasos, y los residuos en el paso 16: viola `early-waste` y paga 2. Los tres respetan la regla estricta.

Las restricciones blandas son la manera de escribir un reglamento con excepciones: las reglas que protegen a los pacientes siguen siendo estrictas, y las de conveniencia reciben un precio.

Una restricción del dominio obliga a todos los problemas; una del problema, solo a ese problema. Envuelta en una preferencia, una restricción se puede violar y se paga en la métrica.

15.7. Dos mundos más

Un róver y un cráter

Un róver se desplaza entre cuatro puntos de paso y recoge muestras. Su dominio, `rover-domain.pddl`, tiene dos acciones, `drive` y `collect`. Su misión es conseguir las muestras de la llanura y del cráter (Figura 15.4). Sin reglas, `rover-p01.pddl` se resuelve en cuatro acciones pasando por la llanura. El control de misión de `rover-p02.pddl` añade tres reglas sobre el cráter, ninguna de las cuales conoce el dominio:

```
(:constraints (and
  ; look at the crater from the ridge before driving into it
  (sometime-before (at crater) (at ridge))
  ; drive into the crater only once
  (at-most-once (at crater))
  ; after the crater, come back to the base to be cleaned
  (sometime-after (at crater) (at base))))
```

```
$ kairos rover-domain.pddl rover-p02.pddl
; cost = 6
(drive base ridge)
(drive ridge crater)
(collect crater)
(drive crater plain)
(collect plain)
(drive plain base)
```

Cada regla ha dejado su huella. El róver da la vuelta por la cresta (primera regla), entra en el cráter una sola vez y sale por el otro lado (segunda) y vuelve a la base aunque el objetivo no lo pide (tercera). La primera regla merece una segunda mirada: el dominio no tiene ningún hecho «el cráter se ha inspeccionado», y no ha hecho falta. La restricción habla de *haber estado* en la cresta, que es historia, y la trayectoria guarda la historia.

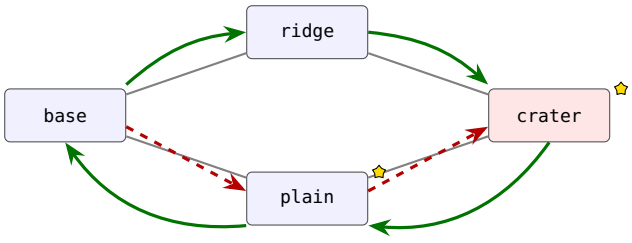


Figura 15.4.: La misión del róver: dos muestras (estrellas). En rojo discontinuo, el plan de cuatro acciones sin reglas. En verde, el plan con las tres reglas: por la cresta, una sola pasada por el cráter y vuelta a la base.

Un taller con un horno

Un taller cuece piezas en un horno: abrir la puerta, cargar, cerrar, calentar (lo que cuece todo lo que haya dentro), enfriar, abrir, descargar. Dos reglas de seguridad pertenecen al taller, no a ninguna orden de trabajo concreta, así que son restricciones del dominio (Listado 15.2).

```
;; A workshop with one oven: parts go in, the oven heats and bakes
;; them, cools, and the parts come out. Two safety rules are
;; constraints of the domain: they hold for every problem.
(define (domain workshop)
  (:requirements :strips :typing :negative-preconditions
                 :conditional-effects :universal-preconditions
                 :constraints)

  (:types part)

  (:predicates
    (outside ?p - part)
    (in-oven ?p - part)
    (baked ?p - part)
    (door-open)
    (hot))

  (:constraints (and
    ; never have the door open while the oven is hot
    (always (not (and (hot) (door-open))))
    ; one heating cycle per shift
    (at-most-once (hot))))
```

Listado 15.2: El taller: dos reglas de seguridad como restricciones del dominio.

Las acciones que siguen solo tienen las precondiciones de la física: una pieza se carga por una puerta abierta, el horno se calienta cuando está frío. En particular, nada en `heat` ni en `open-door` menciona a la otra acción. El problema `workshop-p01.pddl` pide tres piezas cocidas y fuera del horno:

```
$ kairos workshop-domain.pddl workshop-p01.pddl
; cost = 11
(open-door)
(load lid)
(load gear)
(load bracket)
(close-door)
(heat)
(cool)
(open-door)
(unload lid)
(unload gear)
(unload bracket)
```

El `close-door` antes de `heat` y el `cool` antes del segundo `open-door` vienen de la primera regla; las tres piezas en una sola tanda vienen de la segunda. Sin las restricciones, el mismo dominio da un plan de 8 acciones que calienta con la puerta abierta y descarga de un horno caliente. Un plan

precipitado que cuece las piezas una a una (`workshop-p01-hasty.plan`) lo rechaza el validador, y esta vez el mensaje señala al dominio: `the plan violates the domain :constraints`.

15.8. Qué hace el planificador con una restricción

Una búsqueda hacia delante construye un plan estado a estado. Para respetar una restricción no necesita la trayectoria entera: necesita recordar, en cada estado, justo lo suficiente del pasado para saber si la restricción todavía puede cumplirse. Esa memoria es un pequeño autómata, un **monitor**, y cada operador tiene el suyo (Figura 15.5). Para `always` no hay nada que recordar: la condición se comprueba en cada estado. Para `sometime` basta un bit, «ya ha ocurrido». Para `at-most-once` hay tres situaciones: todavía no, durante el tramo, después de él.

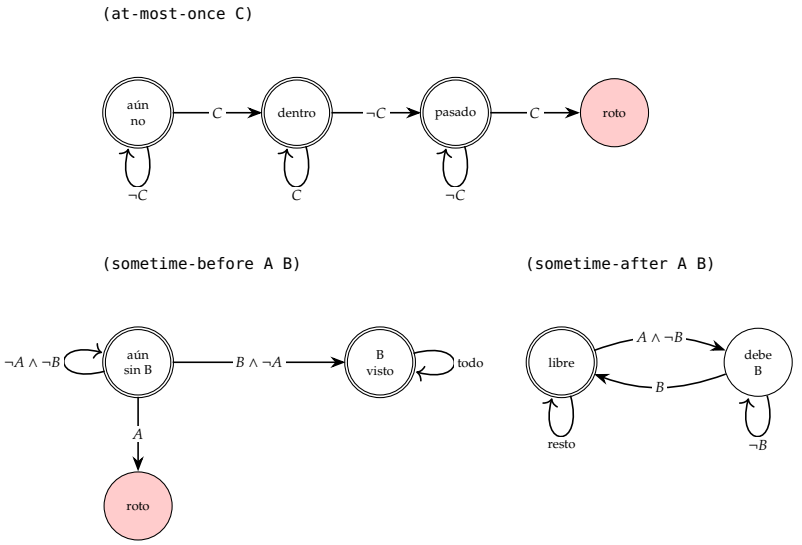


Figura 15.5.: Monitores de tres operadores. Cada estado leído de la trayectoria hace avanzar al monitor por una flecha. Un plan solo puede terminar en un círculo doble; de un estado rojo no se sale nunca. El monitor de `sometime-after` no tiene estado rojo: una obligación siempre se puede saldar todavía, pero el plan no puede terminar mientras la tenga pendiente.

Un planificador puede ejecutar los monitores junto a la búsqueda, o escribirlos dentro del problema. Kairos hace lo segundo: compila las restricciones antes de buscar. Lo que una restricción prohíbe se convierte en una precondición de todas las acciones, ya que todo estado salvo el último es el estado anterior a alguna acción; lo que debe recordar se convierte en un hecho nuevo, que mantienen al día efectos condicionales añadidos a todas las acciones; y lo que debe ser cierto cuando termina el plan se añade al objetivo. Para las restricciones con instantes en planes secuenciales añade un contador de pasos. En los planes temporales, los plazos y las ventanas se convierten en marcas que se activan en sus instantes, el mismo mecanismo que los literales iniciales temporizados del Capítulo 13.

```
$ kairos check courier-rules-domain.pddl courier-rules-p02.pddl
valid, plannable
  typing: full
  negative-preconditions: full
  disjunctive-preconditions: full
  universal-preconditions: full
  trajectory-constraints: full (all, hard or as preferences; always-
     within in temporal plans with a clock)
```

Compilar tiene dos consecuencias para el modelador. La búsqueda y su heurística ven las reglas como precondiciones y objetivos corrientes, así que una regla poda la búsqueda en lugar de rechazar planes ya terminados. Y el coste se paga en tamaño: cada restricción añade condiciones a *todas* las acciones, y un `forall` sobre dos tipos añade un monitor por cada par de objetos.

Pocos planificadores aceptan esta parte del lenguaje. De los que se comparan en la documentación de Kairos (Martín Rico et al., 2026), solo Kairos y OPTIC devolvieron planes válidos para un problema con una restricción de trayectoria; POPF devolvió un plan que la incumplía, y los demás planificadores rechazaron el fichero. Eso pesa en el criterio de Portabilidad, más abajo.

Reglas para la búsqueda antes que reglas para el plan

Las fórmulas temporales entraron en la planificación antes que PDDL3, con otro propósito. TLPlan, el planificador de Fahiem Bacchus y Froduald Kabanza, publicado en 2000 Bacchus y Kabanza, 2000, permitía al modelador escribir fórmulas de lógica temporal que decían qué planes parciales no merecía la pena extender, y podaba la búsqueda con ellas: consejos para el planificador, no requisitos sobre la respuesta. PDDL3 convirtió ese mismo tipo de fórmula en parte del propio problema: algo que un plan debe cumplir, lo encuentre quien lo encuentre y como lo encuentre (A. Gerevini & Long, 2005).

15.9. El método MODEL: reglas sobre el plan completo

Preguntas nuevas para los pasos M y D

Paso M (Mapear la situación). Enumera las reglas de la historia, aparte de sus objetivos. Para cada una:

- **¿Trata de una acción o del plan completo?** «El robot no puede coger lo que no está» trata de una acción. «Nunca en la UCI con residuos» trata de todos los estados.
- **¿Mira un estado o la historia?** Una regla sobre un estado cada vez es un `always`. Una regla con «antes», «después», «una vez» o «a más tardar» necesita memoria.
- **¿De quién es la regla, y puede ceder?** Dominio o problema; estricta o preferencia.

Paso D (Dinámica). Cuando una acción recibe una precondición, pregúntate si describe lo que la acción *necesita* para ser posible o lo que alguien *quiere* del plan. Lo primero es dinámica. Lo segundo es una regla, y puede ser mejor escribirla una vez, como restricción, que repetirla en cada acción que podría incumplirla.

La ficha del modelo gana una tabla de reglas: la frase tal como la dice el cliente, su alcance, su fuerza y el operador elegido, con la prueba de la trampa del paso E aplicada a cada una (Sección 15.3).

Alternativas comparadas

Una regla sobre el plan completo tiene tres codificaciones. Las tres se han escrito y ejecutado para esta sección.

Una restricción. Los modelos del capítulo: `courier-safe` para la regla de la UCI, `courier-rules-p04` para la regla de la base de carga.

Precondiciones en todas las acciones. Es posible cuando la regla mira un estado cada vez. El dominio `courier-icu-pre` escribe la regla de la UCI en las dos acciones que pueden producir el estado prohibido:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
    ; do not enter the ICU with waste
    (or (not (= ?to icu))
        (not (exists (?w - waste) (carrying ?r ?w)))))
  ...
(:action pick
  :parameters (?r - robot ?i - item ?l - location)
  :precondition (and (robot-at ?r ?l) (item-at ?i ?l)
    ; do not pick waste up inside the ICU
    (or (not (= ?l icu))
        (not (exists (?w - waste) (= ?i ?w)))))
```

Devuelve el mismo plan de 16 acciones que la restricción, y el mismo `unsolvable` cuando los residuos están en la UCI. Pero solo es correcto porque se encontraron *las dos* acciones. El dominio `courier-icu-leaky` pone la precondición solo en `move`, la primera en la que piensa cualquiera, y en ese problema encuentra un plan:

```
$ kairos courier-icu-leaky-domain.pddl courier-icu-leaky-p02.pddl
; cost = 14
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(pick r1 waste1 icu)
(drop r1 meds1 icu)
(move r1 icu ward-b)
...
```

Quiera o no el hospital ese plan, el modelo ya no dice lo que su autor creía que decía.

Un autómata con predicados extra. Hace falta sin `:constraints` cuando la regla mira la historia. El modelador escribe a mano el monitor de la Figura 15.5. Para «la base de carga como mucho una vez», el dominio `courier-dock-auto` añade un hecho `(left-dock ?r)`, lo activa cuando el robot sale de la base de carga y prohíbe entrar en ella una vez activado:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
    (or (not (= ?to dock)) (not (left-dock ?r))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (when (= ?from dock) (left-dock ?r))))
```

Con los recados de `p04` devuelve las mismas 12 acciones que `(at-most-once (robot-at r1 dock))`.

Tabla 15.3.: Tres codificaciones de una regla sobre el plan completo, juzgadas con los cinco criterios.

	Restricción	Precondiciones en todas las acciones	Autómata con predicados extra
Fidelidad	Dice la regla una vez, sobre todos los estados; no tiene fugas. Puede ser *demasiado* fuerte (residuos en la UCI)	Exacta solo si no se olvida ninguna acción, ni ahora ni en versiones posteriores; solo reglas de estado	Exacto si el autómata es correcto y el estado inicial lo inicializa bien; cualquier regla
Tamaño	El planificador añade los monitores: condiciones en todas las acciones	Solo las acciones afectadas	Uno o dos hechos por regla y objeto, efectos en las acciones afectadas
Guía	Buena en Kairos, que compila a las otras dos	Buena: precondiciones simples	Buena: hechos simples
Claridad	La mejor: la regla es una línea, aparte de la dinámica, y puede hacerse blanda	La regla queda dispersa y mezclada con la física	La peor: la regla queda oculta en la contabilidad auxiliar
Portabilidad	Mala: pocos planificadores aceptan `:constraints`	Buena: precondiciones ADL	Buena: ADL con efectos condicionales

Veredicto. Escribe primero la regla como restricción, aunque el modelo final no pueda usarla: es la especificación, y `kairos validate` con la restricción es la prueba de cualquier otra codificación. Conserva la restricción cuando el planificador la admita, cuando las reglas cambien de un problema a otro o cuando algunas deban ser blandas. Pasa una regla a precondiciones cuando trate de un estado, pocas acciones puedan incumplirla y el modelo deba ejecutarse en planificadores clásicos. Escribe el autómata cuando la regla trate de la historia y el planificador no tenga `:constraints`; después valida sus planes frente a la versión con la restricción.

15.10. Errores comunes

Olvidar que el estado inicial cuenta

`courier-rules-once-start.pddl` pide pasar por la base de carga como mucho una vez, con el robot empezando *en* la base de carga y residuos que llevar a ella:

```
$ kairos courier-rules-domain.pddl courier-rules-once-start.pddl
note: no plan: every state reachable from the initial one was
     searched (245 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

S_0 forma parte de la trayectoria, así que la única visita se gasta antes de la primera acción, y volver es una segunda visita. Lo mismo vale para `sometime-before`: una `B` cierta en S_0 cuenta para toda `A` posterior, pero una `A` cierta en S_0 no tiene nada antes y viola la restricción de inmediato. *Remedio*: decide qué significa la regla para el estado en que empieza el día, y escríbela pensando en él.

Un operador fuera de su sección

Los operadores solo existen dentro de `:constraints`. En un objetivo se leen como predicados que nadie ha declarado (`courier-rules-ingoal.pddl`):

```
$ kairos check courier-rules-domain.pddl courier-rules-ingoal.
    pddl
courier-rules-ingoal.pddl:20:15: warning: 'sometime' is not
    declared: never true [dialect.undeclared_predicate]
 20 |               (sometime (robot-at r1 icu)))))
    |               ^
valid, plannable
  typing: full
```

Un aviso, no un error; después el planificador responde `unsolvable`, porque el objetivo contiene un hecho que nunca es cierto. *Remedio*: lleva la condición a `:constraints` y lee los avisos de `kairos check`.

Una ventana a la que el plan nunca llega

`courier-rules-short.pddl` entrega medicamentos en la sala A con un toque de queda, `(hold-after 10 (robot-at r1 dock))`. El plan devuelto tiene 7 acciones y termina en la sala A. Es válido: la restricción habla de los estados posteriores al instante 10, y este plan no tiene ninguno. Una ventana restringe los estados que caen dentro de ella; no hace que el plan dure hasta que se abra. *Remedio*: si el robot debe estar en la base de carga, dilo también en el objetivo.

Quieto durante toda una ventana

En un plan temporal, un estado empieza en un acontecimiento y dura hasta el siguiente. Kairos decide una ventana por los instantes en que los estados *empiezan*. El plan `courier-timed-standing.plan` lleva al robot al pasillo en el segundo 14 y lo deja allí parado hasta el segundo 45, sin ningún acontecimiento entre medias:

```
$ kairos validate courier-timed-domain.pddl courier-timed-p02.
    pddl courier-timed-standing.plan
plan valid, makespan 105.01, metric 105.01
```

Ningún estado empieza entre 20 y 40, así que `(hold-during 20 40 ...)` no tiene nada que comprobar, aunque el robot está en el pasillo cerrado durante toda la ventana. *Remedio*: cuando una ventana deba cumplirse en todo instante, dale a sus bordes un acontecimiento propio. En `courier-timed-p03.pddl` dos literales iniciales temporizados (Capítulo 13) activan un hecho `(closed corridor)` en 20 y lo desactivan en 40, y la regla pasa a ser un `always`:

```
(at 20 (closed corridor)) (at 40 (not (closed corridor)))
...
(forall (?l - location)
  (always (not (and (closed ?l) (robot-at r1 ?l)))))
```

Ahora un estado empieza en el segundo 20 con el robot dentro, y el mismo plan se rechaza con `plan invalid`.

Pasos tomados por segundos

`(within 45 ...)` en un problema secuencial le da al plan 45 *acciones*, lo que casi nunca limita nada, y el plan encontrado parece ignorar la restricción. En un problema temporal, `(within 7 ...)` significa 7 segundos. *Remedio*: un número en una restricción tiene la unidad del plan: posiciones si el dominio tiene acciones instantáneas, la unidad de tiempo de las duraciones si las tiene durativas. Si un modelo secuencial necesita tiempo real, antes necesita acciones durativas.

Una regla más fuerte que el mundo

Una restricción recorta planes, y con una regla de más los recorta todos. El síntoma es `unsolvable` en un problema que parecía fácil, como con los residuos en la UCI. *Remedio*: el procedimiento de depuración del Capítulo 7 con un paso más al principio: quita las restricciones de una en una hasta que aparezca un plan, y después valida ese plan frente a cada restricción por separado para ver cuál incumple, y dónde.

15.11. Resumen

Un plan pasa por una secuencia de estados, su trayectoria, y PDDL 3.0 permite que un modelo imponga condiciones sobre toda ella. La sección `:constraints`, en el dominio o en el problema, las contiene; el requisito se llama igual.

Operador	Dice	Regla típica
`always`	en todos los estados	seguridad: nunca esto
`sometime`	en algún estado	pasar por un sitio, presentarse
`at end`	en el último estado	un objetivo
`at-most-once`	un tramo como mucho	no volver
`sometime-before A B`	`A` solo después de `B`	un requisito previo
`sometime-after A B`	tras `A`, `B`	una obligación
`within t`	a más tardar en t	un plazo
`always-within t A B`	tras `A`, `B` en t como mucho	un tiempo de reacción
`hold-during t1 t2`	en los estados de una ventana	un cierre
`hold-after t`	en los estados posteriores a t	un toque de queda

Tabla 15.4.: Los diez operadores y la clase de regla que escribe cada uno.

Las restricciones son estrictas salvo que se envuelvan en una preferencia. Mantienen el reglamento fuera de la dinámica: el dominio dice lo que se puede hacer; las restricciones, lo que está permitido hacer. Donde el planificador no las acepta, las mismas reglas se escriben como precondiciones o como un autómata hecho a mano, y la restricción se queda como su especificación y su prueba.

15.12. Lecturas complementarias

El informe sobre PDDL3 (A. Gerevini & Long, 2005) define las restricciones de trayectoria con su semántica sobre secuencias temporizadas de estados; compara sus definiciones de `hold-during` y `hold-after` con las últimas filas de la Tabla 15.1. El artículo sobre la quinta competición (A. E. Gerevini et al., 2009) presenta los dominios escritos con restricciones y cómo los trataron los planificadores participantes. Las restricciones de trayectoria y su compilación en autómatas ocupan una sección de

Haslum et al. (2019). El validador VAL (Howey et al., 2004) es la referencia habitual para comprobar planes frente a la semántica de PDDL. OPTIC (Benton et al., 2012) es el otro planificador de la comparación que maneja restricciones, y preferencias sobre ellas, en problemas temporales.

15.13. Ejercicios

Ejercicios de la Sección 15.1 — Una regla que el plan incumple

Ejercicio 15.1.1. Para cada regla, di si puede escribirse como objetivo, como precondición de una acción o si necesita el plan completo: (a) el robot termina en la base de carga; (b) el robot nunca lleva más de tres artículos; (c) una muestra nunca se deja en el pasillo, ni siquiera un momento; (d) el robot no puede dejar lo que no lleva; (e) el robot pasa por la farmacia al menos una vez; (f) la sala A se visita antes que la sala B.

Ejercicios de la Sección 15.2 — Los estados de un plan: always

Ejercicio 15.2.1. El servicio de higiene pide más: los residuos y los medicamentos nunca deben estar juntos en la cesta. Añade esa restricción al problema `courier-rules-p01.pddl`, con un `forall` sobre robots, residuos y medicamentos, y ejecútalo. ¿En qué se diferencia el plan del de `p02`?

Ejercicio 15.2.2. Sin ejecutar nada: ¿qué responde Kairos si la restricción de `courier-rules-p02.pddl` se sustituye por `(always (not (robot-at r1 dock)))`? ¿Por qué?

Ejercicios de la Sección 15.3 — Lo que debe ocurrir, y en qué orden

Ejercicio 15.3.1. Añade a `courier-rules-p03.pddl` la restricción de que ninguna sala se visite dos veces: `(forall (?l - location) (at-most-once (robot-at r1 ?l)))`. Predice la respuesta a partir del mapa y después ejecútalo.

Ejercicio 15.3.2. ¿Qué significan estas restricciones? (a) `(sometime-before (robot-at r1 lab) (robot-at r1 lab))`; (b) `(sometime-after (robot-at r1 icu) (robot-at r1 icu))`; (c) `(sometime-before (carrying r1 meds1) (sanitized r1))` en un problema cuyo estado inicial contiene `(sanitized r1)`.

Ejercicio 15.3.3. Sustituye la restricción de `courier-rules-p05.pddl` por `(always (imply (robot-at r1 lab) (sanitized r1)))`. El plan de la sección se ha guardado como `courier-rules-p05-early.plan`: comprueba que ahora no es válido y lee el plan nuevo. ¿Qué requisito necesita `imply`?

Ejercicios de la Sección 15.4 — Reglas con reloj

Ejercicio 15.4.1. En `courier-rules-p07.pddl`, cambia el plazo a 6 pasos. Explica la respuesta de Kairos contando acciones.

Ejercicio 15.4.2. En `courier-timed-p01.pddl`, ¿hasta dónde se puede apretar el plazo? Suma las duraciones de las acciones que necesita la entrega, prueba un plazo de 43 y explica por qué no se puede cumplir uno de 42 aunque las duraciones sumen 42.

Ejercicio 15.4.3. Explica la diferencia entre `(within 5 (sanitized r1))` y `(always-within 5 (robot-at r1 icu) (sanitized r1))`. Describe con palabras un plan que cumpla la primera y viole la segunda.

Ejercicios de la Sección 15.5 — Los diez operadores, uno junto a otro

Ejercicio 15.5.1. Toma las 16 acciones de `courier-rules-p02-ok.plan` y escribe su trayectoria: dónde está el robot y qué lleva en $S_0, \ldots, S_{16}$. Decide a mano cuáles de las siguientes restricciones cumple: (a) `(sometime (robot-at r1 lab))`; (b) `(at-most-once (robot-at r1 pharmacy))`; (c) `(sometime-before (robot-at r1 icu) (carrying r1 meds1))`; (d) `(within 8 (item-at meds1 icu))`; (e) `(always-within 5 (carrying r1 waste1) (item-at waste1 dock))`; (f) `(hold-after 12 (not (robot-at r1 icu)))`. Después compruébalo con `kairos validate`.

Ejercicios de la Sección 15.6 — Dónde viven las reglas, y reglas que ceden

Ejercicio 15.6.1. El taller de la Sección 15.7 tiene en su dominio «un ciclo de calentamiento por turno». Da una razón para llevar esa regla al problema, y otra para mantener la otra regla en el dominio.

Ejercicio 15.6.2. En el problema `courier-rules-p11.pddl`, da al deseo `early-waste` un peso de 4. Predice el plan y su coste, y después ejecútalo.

Ejercicios de la Sección 15.7 — Dos mundos más

Ejercicio 15.7.1. Quita la tercera regla de `rover-p02.pddl` y predice el plan. ¿Cuál de sus acciones se debe todavía a una regla?

Ejercicio 15.7.2. Las piezas no deben pasar mucho tiempo en el horno. Añade esta restricción al problema `workshop-p01.pddl` y encuentra el menor `N` para el que hay plan:

```
(forall (?p - part)
  (always-within N (in-oven ?p) (outside ?p)))
```

¿Qué exige la restricción sobre el orden en que se descargan las piezas?

Ejercicios de la Sección 15.9 — El método MODEL: reglas sobre el plan completo

Ejercicio 15.9.1. Escribe la primera regla del róver, «la cresta antes que el cráter», como un autómata con predicados extra, en un dominio `rover-auto` sin `:constraints`. Hazlo general: en el problema, cualquier punto de paso puede marcarse como peligroso y cualquiera como mirador. Compara su plan con el de la restricción, y di qué hace tu codificación si el róver *empieza* en la cresta.

Ejercicio 15.9.2. El paso M aplicado a una historia. «En nuestra cocina, el cocinero se lava las manos después de tocar carne cruda y antes de tocar cualquier otra cosa. El horno nunca se queda encendido sin nadie en la cocina. Cada plato se prueba antes de servirlo. La puerta de la nevera se abre el menor número de veces posible. Todo se sirve antes de las 14:00». Para cada regla, da su alcance, su fuerza y el operador (o la razón por la que no basta un único operador).

16. Fluentes de objeto

16.1 Un robot en dos habitaciones 367
16.2 Una función cuyo valor es un objeto 368
16.3 Fluentes como términos 370
16.4 Sin valor: indefinido . . 371
16.5 Dos mundos más 372
Bloques: un fluente en lugar de cuatro predicados 372
Colorear un mapa 373
16.6 Variables de estado: lo que ve el planificador . 374
16.7 El método MODEL: un valor o muchos 376
Alternativas comparadas 376
16.8 PDDL 3 de un vistazo . 377
16.9 Errores comunes 378
16.10 Resumen 379
16.11 Lecturas complementarias 379
16.12 Ejercicios 379

Desde el Capítulo 2, la posición del mensajero ha sido un predicado, `(robot-at ?r ?l)`. Un predicado es una relación: para cada robot y cada habitación es verdadero o falso, y nada en él dice que un robot esté en *una* habitación. Que un robot tenga exactamente una posición es un hecho del mundo que el modelo mantiene a base de disciplina: cada acción que añade un `robot-at` borra otro. El Capítulo 7 dio nombre al patrón, el invariante de a lo sumo uno, y puso su fallo, el borrado olvidado, entre los antipatrones. Cuando la disciplina falla, el planificador no se queja. Usa el robot, que ahora está en dos habitaciones.

PDDL 3.1 ofrece una forma de decirlo una sola vez, en la declaración. Un **fluente de objeto** es una función cuyo valor es un objeto: `(location-of ?r)` es *la* habitación del robot `?r`. Tiene un único valor en cada momento, una acción lo cambia asignándole uno nuevo, y el valor antiguo desaparece sin que nadie lo borre. El Capítulo 8 presentó funciones cuyos valores son números; este capítulo les da objetos como valores, con las mismas palabras clave: `:functions`, `=` y `assign`.

El capítulo empieza por el error y después reescribe el mundo del mensajero con dos fluentes de objeto. Usa los fluentes como términos dentro de otras expresiones, se encuentra con el fluente sin valor, `undefined`, y visita el mundo de bloques y un mapa que hay que colorear. Después mira lo que los planificadores construyen a partir de un modelo, variables de estado con dominios finitos, que es justo lo que un fluente de objeto declara directamente. El método MODEL gana una pregunta para el paso O y una comparativa entre predicado y fluente. Una última sección cierra la Parte V con las características de PDDL 3 en una tabla.

16.1. Un robot en dos habitaciones

Tomemos el dominio con tipos del Capítulo 4 y un viaje de ida y vuelta: llevar los medicamentos de la farmacia a la sala A y volver a la base de carga (`courier-p02.pddl`). El plan tiene diez acciones: siete para la entrega y tres movimientos para volver.

Ahora, un despiste de los que comete todo modelador. La acción `move` ha perdido una línea en `courier-twin-domain.pddl`:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (robot-at ?r ?to))            ; the delete is missing
```

`kairos check` no encuentra nada mal, porque no hay nada mal en el PDDL. El planificador encuentra un plan:

```
$ kairos courier-twin-domain.pddl courier-twin-p02.pddl
; cost = 6
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 lobby corridor)
```

```
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Seis acciones para un viaje de ida y vuelta que necesita diez. Leamos la cuarta: el robot coge los medicamentos en la farmacia y sale *del vestíbulo*. Nunca volvió sobre sus pasos, porque nunca se marchó: tras dos movimientos, `(robot-at r1 dock)`, `(robot-at r1 lobby)` y `(robot-at r1 pharmacy)` son verdaderos los tres. Y el plan no tiene viaje de vuelta, porque el robot sigue en la base de carga (Figura 16.1). El validador acepta el plan: comprueba el plan frente al modelo, y el problema es el modelo.

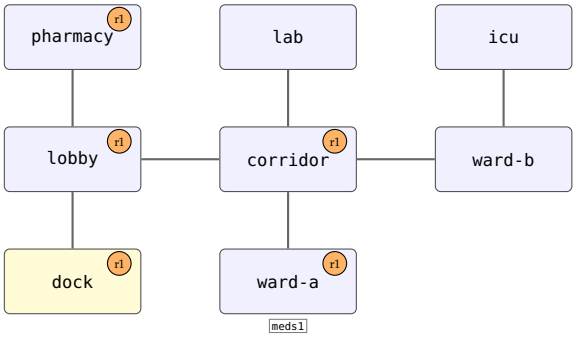


Figura 16.1.: El último estado del plan de seis acciones en el dominio con el borrado olvidado. El robot está en cinco habitaciones a la vez. Nada en un predicado dice que este estado sea imposible.

La prueba de la trampa del paso E caza este error si alguien lee el plan. Pero la causa es más profunda que una línea que falta. El predicado `robot-at` puede describir 2^8 situaciones para un robot en esta planta, cualquier subconjunto de las ocho habitaciones, y solo 8 de ellas tienen sentido. El modelo confía en que sus acciones lo mantengan dentro de esas 8. Una representación en la que las otras 248 ni siquiera pudieran escribirse no necesitaría la disciplina.

Un predicado deja que un robot esté en cualquier número de habitaciones. Que esté en exactamente una es un invariante que solo mantienen los efectos de las acciones, y un borrado olvidado lo rompe sin hacer ruido.

16.2. Una función cuyo valor es un objeto

Un **fluente de objeto** se declara en `:functions`, como los fluentes numéricos del Capítulo 8, con un añadido: detrás de la función va el tipo de su valor.

```
(:functions
  (location-of ?r - robot) - location   ; the robot's room
  (holder ?i - item) - place)           ; a room, or a robot
```

`(location-of ?r)` recibe un robot y da un lugar. Para cada robot tiene, en cada estado, como mucho un valor. El requisito es `:object-fluents`. Tres construcciones usan un fluente, y las tres son ya conocidas:

- **Valor inicial**, en `:init`: `(= (location-of r1) dock)`.
- **Comprobación**, en una precondición o en un objetivo: `(= (location-of ?r) ?from)` es verdadero cuando el valor del fluente es el objeto `?from`. Es la igualdad del Capítulo 4, así que el dominio declara también `:equality`.
- **Cambio**, en un efecto: `(assign (location-of ?r) ?to)` da al fluente un valor nuevo. El antiguo queda sustituido. No hay nada que borrar.

El Listado 16.1 es el mundo del mensajero reescrito. La posición de un robot es `(location-of ?r)`. Para los artículos, el dominio con tipos tenía dos predicados, `item-at` y `carrying`, con un segundo invariante oculto: un artículo está en una habitación o en la pinza de un robot, nunca en las dos cosas y nunca en ninguna. Un único fluente lo dice, `(holder ?i)`: lo que sostiene el artículo. Su valor puede ser una habitación o un robot, así que el dominio da a los dos tipos un supertipo común, `place` (Capítulo 4).

```
;; The courier world with object fluents: where a robot is and who
;; holds an item are functions with one value, not predicates.
(define (domain courier-of)
  (:requirements :strips :typing :equality :object-fluents)

  (:types
    robot location - place          ; what can hold an item
    item)

  (:predicates
    (connected ?from ?to - location)
    (free ?r - robot))

  (:functions
    (location-of ?r - robot) - location   ; the robot's room
    (holder ?i - item) - place            ; a room, or a robot

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (= (location-of ?r) ?from)
                       (connected ?from ?to))
    :effect (assign (location-of ?r) ?to))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (= (location-of ?r) ?l) (= (holder ?i) ?l)
                       (free ?r))
    :effect (and (assign (holder ?i) ?r) (not (free ?r))))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (= (location-of ?r) ?l) (= (holder ?i) ?r))
    :effect (and (assign (holder ?i) ?l) (free ?r))))
```

Listado 16.1: El mundo del mensajero con fluentes de objeto. `robot-at`, `item-at` y `carrying` han desaparecido.

Comparemos cada acción con su versión del Capítulo 4. `move` tenía dos efectos, un borrado y una adición; ahora tiene una asignación, y no hay ninguna línea que olvidar. `pick` tenía tres efectos sobre el artículo y el robot; ahora el artículo cambia de poseedor, de la habitación al robot. `(free ?r)` sigue siendo un predicado: es una propiedad que es verdadera o falsa, no un valor (el Ejercicio 16.2.2 la elimina).

El problema `courier-of-p01.pddl` da los valores iniciales y comprueba uno en el objetivo:

```
    (= (location-of r1) dock) (free r1)
    (= (holder meds1) pharmacy))
  (:goal (= (holder meds1) ward-a)))
```

```
$ kairos courier-of-domain.pddl courier-of-p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
```

```
(drop r1 meds1 ward-a)
```

El plan es el del Capítulo 4, acción por acción: el mundo no ha cambiado, ha cambiado su descripción. El viaje de ida y vuelta (`courier-of-p02.pddl`) lleva diez acciones, como debe ser. Y el error de la sección anterior no puede escribirse: no hay manera de asignar un segundo valor a `(location-of r1)` y conservar el primero (Figura 16.2).

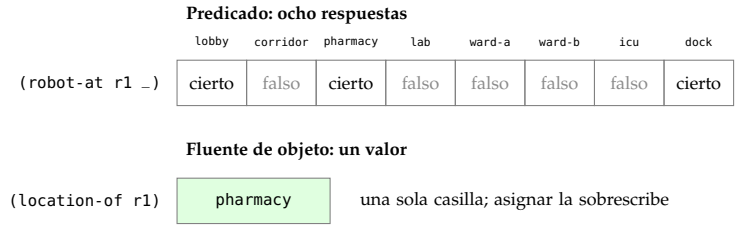


Figura 16.2.: La misma pregunta, «¿dónde está `r1`?», hecha a un predicado y a un fluente. El predicado responde una vez por habitación, y tres respuestas pueden ser `true`, como en el estado al que llegó el dominio con el error. El fluente tiene una sola casilla.

Un fluente de objeto es una función con un objeto como valor: se declara con su tipo, recibe un valor en el estado inicial, se comprueba con = y se cambia con assign. Un valor cada vez, por construcción.

Las funciones no son predicados

Un fluente nunca aparece solo como condición, y no recibe su valor como argumento. La costumbre de años de `robot-at` produce `(location-of r1 dock)`. En el estado inicial de `courier-of-oldstyle.pddl` solo da un aviso:

```
$ kairos check courier-of-domain.pddl courier-of-oldstyle.pddl
courier-of-oldstyle.pddl:16:5: warning: in the initial state:
     unknown predicate 'location-of'; the fact is ignored [
     semantic.unknown_predicate]
 16 |     (location-of r1 dock) (free r1)
    |     ^
valid, plannable
...
```

El hecho se ignora, el robot no empieza en ninguna parte y el planificador responde `unsolvable`. *Solución*: un fluente siempre se compara o se asigna: `(= (location-of r1) dock)`.

16.3. Fluentes como términos

En `(= (location-of ?r) ?from)`, el parámetro `?from` tiene un único cometido: llevar la posición del robot de la precondición al resto de la acción. Con un predicado no había otra forma de nombrar «la habitación donde está `?r`». Con un fluente sí la hay: la expresión `(location-of ?r)` *es* esa habitación, y puede escribirse en cualquier sitio donde quepa un objeto, como argumento de un predicado o a cualquiera de los dos lados de una igualdad.

El dominio `courier-terms` lo usa en dos acciones:

```
; no ?from: the robot leaves from wherever it is
(:action move
  :parameters (?r - robot ?to - location)
  :precondition (connected (location-of ?r) ?to)
  :effect (assign (location-of ?r) ?to))

; no ?l: the item must be held by the room the robot is in
(:action pick
  :parameters (?r - robot ?i - item)
  :precondition (and (= (holder ?i) (location-of ?r)) (free ?r))
```

```
    :effect (and (assign (holder ?i) ?r) (not (free ?r))))
```

`move` se lee así: la habitación del robot está conectada con `?to`. `pick`, así: el poseedor del artículo es la habitación del robot. Las dos frases se parecen más a como describiríamos la acción con palabras que sus versiones con un parámetro de más. `drop` conserva su parámetro `?l`, que nombra la habitación que pasa a ser el poseedor.

```
$ kairos courier-terms-domain.pddl courier-terms-p01.pddl
; cost = 7
(move r1 lobby)
(move r1 pharmacy)
(pick r1 meds1)
(move r1 lobby)
(move r1 corridor)
(move r1 ward-a)
(drop r1 meds1 ward-a)
```

El plan es el mismo y sus líneas son más cortas: una acción se nombra por sus parámetros, y estas acciones tienen menos. Eso también es una ganancia para el planificador. El Capítulo 3 mostró que los parámetros multiplican las acciones instanciadas. `pick` con `?l` tenía una acción instanciada por cada robot, artículo y habitación; sin él, una por cada robot y artículo. En este problema, Kairos instancia 30 acciones de `courier-of` y 23 de `courier-terms`, y la diferencia crece con el número de habitaciones y de artículos.

Un fluente puede escribirse donde se espera un objeto. Los parámetros cuyo único cometido era nombrar un valor desaparecen, y con ellos, acciones instanciadas.

El precio lo paga quien lee el plan: `(move r1 lobby)` ya no dice desde dónde. Para un robot que ejecuta el plan eso no hace daño, porque sabe dónde está. Para una persona que lo depura, el parámetro redundante era documentación.

16.4. Sin valor: indefinido

Un fluente al que no se le da valor en `:init` no tiene ninguno: está **indefinido**. Un efecto también puede quitarle el valor, con la palabra clave `undefined` en una asignación. Las dos son herramientas legítimas: algunas propiedades solo tienen valor parte del tiempo.

El dominio `courier-cargo` conserva `carrying` como predicado y convierte la habitación de un artículo en un fluente, `(room-of ?i)`. Mientras un robot lleva el artículo, el artículo no está en ninguna habitación:

```
(:action pick
  :parameters (?r - robot ?i - item ?l - location)
  :precondition (and (= (location-of ?r) ?l) (= (room-of ?i) ?l)
                     (free ?r))
  :effect (and (carrying ?r ?i) (not (free ?r))
               (assign (room-of ?i) undefined)))
```

`drop` vuelve a asignar la habitación. El problema `courier-cargo-p01.pddl`, con dos entregas, se resuelve en 11 acciones, como se esperaba.

Un fluente indefinido no es igual a nada. Una comprobación como `(= (room-of ?i) ?l)` no puede cumplirse mientras se lleva el artículo, sea cual sea `?l`, así que nadie puede coger un artículo de la pinza de otro robot. Lo mismo ocurre al final del plan. El plan `courier-cargo-p01-carried.plan` se detiene una acción antes, con la muestra todavía en la pinza, en el laboratorio:

```
$ kairos validate courier-cargo-domain.pddl courier-cargo-p01.pddl
    courier-cargo-p01-carried.plan
courier-cargo-p01-carried.plan: error: evaluating the goal: (room-of
     sample1) is undefined [validate.evaluation]
plan invalid
```

undefined no es un valor con el que comparar

`undefined` puede asignarse, pero no comprobarse: no existe `(= (room-of ?i) undefined)`. El dominio `courier-cargo-undef` lo intenta en una precondición:

```
$ kairos check courier-cargo-undef-domain.pddl
courier-cargo-undef-domain.pddl:32:40: error: in action 'drop':
     unknown object or constant 'undefined' [semantic.
     unknown_object]
 32 |                      (= (room-of ?i) undefined))
    |                                      ^
invalid
```

Solución: cuando el modelo tiene que *preguntar* si una propiedad tiene valor, no uses `undefined` para el que falta. Mantén un hecho que lo diga, como hace aquí `(carrying ?r ?i)`, o da al fluente un valor para ese caso, como hace `holder` con el robot.

Esa es la decisión de diseño que separa los dos modelos del mensajero de este capítulo. En `courier-of`, el fluente `holder` es *total*: todo artículo tiene siempre un poseedor, una habitación o un robot, y ninguna comprobación se encuentra nunca con un valor indefinido. En `courier-cargo`, el fluente `room-of` es *parcial*, y una segunda pieza del estado, `carrying`, cubre el hueco; dos piezas del estado que deben concordar vuelven a ser un invariante. Prefiere los fluentes totales. Usa `undefined` cuando «sin valor» sea una situación real por la que nada necesite preguntar: una región que todavía no tiene color, en la sección siguiente.

Un fluente sin valor está indefinido. Se le puede asignar undefined, pero nunca compararlo con él. Un fluente que siempre tiene valor es más fácil de mantener correcto.

16.5. Dos mundos más

Bloques: un fluente en lugar de cuatro predicados

El mundo de bloques del Capítulo 2 tenía `on`, `on-table`, `clear` y `holding`, y la mayoría de los efectos de sus cuatro acciones existían para mantenerlos acompasados. Pero todo el estado de una mesa con bloques es un hecho por bloque: sobre qué descansa. El Listado 16.2 modela exactamente eso, `(on-top-of ?b)`, cuyo valor es otro bloque o la mesa. La mesa es una constante del dominio, de un tipo `surface`, y `block` y `surface` comparten el supertipo `support`.

```
;; Blocks world with one object fluent: what each block is on.
;; There is no (clear ?b) and no (holding ?b): a block is clear
;; when no block is on top of it: that is read from the fluent.
(define (domain blocks-of)
  (:requirements :strips :typing :equality :negative-preconditions
                 :existential-preconditions :object-fluents)

  (:types block surface - support)

  (:constants table - surface)

  (:functions
```

```
    (on-top-of ?b - block) - support)   ; what ?b rests on

  ; move a clear block onto another clear block
  (:action stack
    :parameters (?b ?to - block)
    :precondition (and
      (not (= ?b ?to))
      (not (exists (?c - block) (= (on-top-of ?c) ?b)))
      (not (exists (?c - block) (= (on-top-of ?c) ?to))))
    :effect (assign (on-top-of ?b) ?to))

  ; move a clear block to the table
  (:action unstack
    :parameters (?b - block)
    :precondition (and
      (not (= (on-top-of ?b) table))
      (not (exists (?c - block) (= (on-top-of ?c) ?b))))
    :effect (assign (on-top-of ?b) table)))
```

Listado 16.2: El mundo de bloques con un único fluente de objeto. «Despejado» no se guarda: se lee del fluente con una condición existencial.

Un bloque está despejado cuando ningún bloque lo tiene como apoyo: `(not (exists (?c - block) (= (on-top-of ?c) ?b)))`. Con eso, el dominio necesita dos acciones, y cada una tiene un solo efecto. Para invertir una torre de tres (`blocks-of-p01.pddl`):

```
$ kairos blocks-of-domain.pddl blocks-of-p01.pddl
; cost = 3
(unstack c)
(stack b c)
(stack a b)
```

No hay ningún estado alcanzable en este modelo en el que un bloque esté sobre dos cosas, o despejado y cubierto a la vez: no hay nada que pueda desacompasarse. El precio está en las precondiciones, que ahora cuantifican sobre todos los bloques, y en la granularidad: la mano que sostenía un bloque entre dos acciones ha desaparecido (el Capítulo 7 discutió esa decisión).

Colorear un mapa

Hay que dar a cada región de un mapa un color, de modo que dos regiones con frontera común no compartan color. Una región tiene exactamente un color, o ninguno todavía: un fluente de objeto que empieza indefinido (Listado 16.3).

```
;; Coloring a map: every region gets one color, and two regions
;; that share a border never get the same one.
(define (domain coloring)
  (:requirements :strips :typing :equality :negative-preconditions
                 :disjunctive-preconditions
                 :universal-preconditions :object-fluents)

  (:types region color)

  (:predicates
    (border ?a ?b - region))

  (:functions
    (color-of ?r - region) - color)   ; undefined until painted

  (:action paint
    :parameters (?r - region ?c - color)
    :precondition (forall (?n - region)
                    (imply (border ?r ?n)
                           (not (= (color-of ?n) ?c))))
```

```
    :effect (assign (color-of ?r) ?c)))
```

Listado 16.3: Coloreado de mapas. El problema no da a `color-of` ningún valor inicial: todas las regiones empiezan sin color.

La precondición de `paint` mira todas las vecinas: ninguna tiene el color `?c`. Una vecina sin color pasa la comprobación, porque un fluente indefinido no es igual a `?c`. El objetivo de `coloring-p01.pddl` pide que cada región tenga *algún* color, que es la manera en que una condición dice «está definido» sin la comparación prohibida:

```
(:goal (forall (?r - region)
         (exists (?c - color) (= (color-of ?r) ?c)))))
```

El mapa tiene cuatro regiones en anillo alrededor de una quinta (Figura 16.3).

```
$ kairos coloring-domain.pddl coloring-p01.pddl
; cost = 5
(paint west red)
(paint east red)
(paint center green)
(paint south blue)
(paint north blue)
```

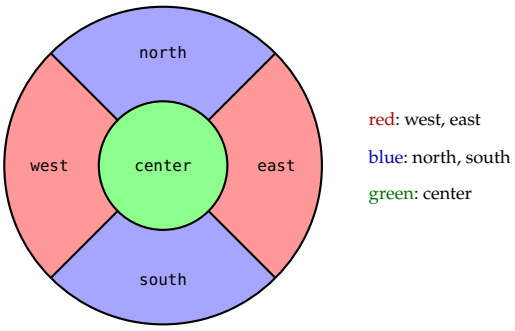


Figura 16.3.: El mapa de `coloring-p01` con los colores del plan. Las regiones opuestas del anillo no se tocan y pueden compartir color; el centro toca las cuatro y necesita el tercero.

Con dos colores (`coloring-p02.pddl`), Kairos recorre los 94 estados alcanzables y responde `unsolvable`, que aquí es una pequeña demostración: este mapa no se puede colorear con dos colores. El coloreado de mapas es un problema de satisfacción de restricciones, y un planificador no es la mejor herramienta para resolverlo. Como modelo, sin embargo, muestra el encaje: una variable por región, un valor por variable, una condición entre vecinas.

16.6. Variables de estado: lo que ve el planificador

El Capítulo 3 describió un estado como un conjunto de hechos. Esa es la visión del lenguaje. La mayoría de los planificadores modernos trabajan con otra: un estado es una asignación de valores a una lista fija de **variables de estado**, cada una con un dominio finito de valores. «La posición de `r1`» es una variable con ocho valores; «el poseedor de `meds1`» es una variable con nueve (Figura 16.4).

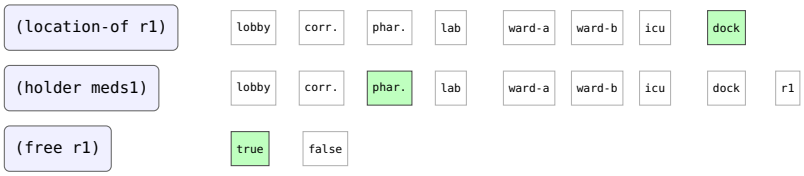


Figura 16.4.: El estado inicial de `courier-of-p01` como tres variables de estado. Cada fila es una variable con su dominio; el valor en verde es el actual. Un estado es una elección por fila: $8 \times 9 \times 2 = 144$ combinaciones, frente a 2^{18} conjuntos de hechos en el modelo con predicados.

Un planificador que recibe predicados tiene que *descubrir* las variables. Analiza las acciones, demuestra que ciertos hechos nunca pueden ser verdaderos a la vez (un **grupo mutex**: los ocho hechos `(robot-at r1 _)`) y funde cada grupo en una variable. `kairos -v` informa del resultado en su última línea. Para el modelo con predicados del Capítulo 4:

```
$ kairos courier-domain.pddl courier-p01.pddl -v
...
kairos: 18 atoms, 30 actions (0 durative), 3 variables (2 mutex
     groups), 50 states, 37 expanded, 47 evaluated, 1 plans, 4252.1
     bytes per state, 8 MB peak
```

Dieciocho hechos, en los que el planificador encontró dos grupos mutex (la posición del robot; el paradero del artículo) para acabar con 3 variables. Para el modelo con fluentes de objeto:

```
$ kairos courier-of-domain.pddl courier-of-p01.pddl -v
...
kairos: 1 atoms, 30 actions (0 durative), 3 variables (0 mutex
     groups), 50 states, 37 expanded, 47 evaluated, 1 plans, 4252.1
     bytes per state, 8 MB peak
```

Las mismas 3 variables, las mismas 30 acciones, la misma búsqueda, estado por estado. Pero no hubo que encontrar ningún grupo mutex: los dos fluentes *eran* las variables, y el único hecho que queda es `(free r1)`. ¿Y el dominio con el borrado olvidado? La misma línea de `kairos -v` para `courier-twin-p02` dice `10 variables (1 mutex groups)`. El análisis ya no pudo demostrar que el robot está en una sola habitación, porque no es verdad, y cada hecho `robot-at` se convirtió en una variable propia. Un número inesperado en esa línea es un síntoma que conviene conocer: un modelo cuyos invariantes el planificador no encuentra suele ser un modelo cuyos invariantes no se cumplen.

Los planificadores convierten conjuntos de hechos en variables de estado demostrando qué hechos se excluyen entre sí. Un fluente de objeto les entrega la variable, y con ella el invariante.

Representaciones de dominio finito

La representación con variables de varios valores se conoce en la literatura de planificación como SAS$^+$ o *representación de dominio finito*. El planificador Fast Downward (Helmert, 2006) la convirtió en la forma interna estándar: un traductor convierte el PDDL en variables de dominio finito antes de cualquier búsqueda, y el análisis de invariantes en el que se apoya esa traducción lo describe en detalle Helmert (2009). Muchas heurísticas se definen sobre estas variables y son más precisas cuanto menos numerosas y más grandes son: un planificador que sabe que un robot tiene una posición no malgasta esfuerzo en estados en los que tiene dos.

PDDL 3.1

Los fluentes de objeto llegaron con PDDL 3.1, el lenguaje de la sexta Competición Internacional de Planificación, en 2008. La

misma revisión dividió en dos el antiguo requisito `:fluents`, `:numeric-fluents` y `:object-fluents`, y añadió el `:action-costs` del Capítulo 9. La gramática completa de PDDL 3.1 la puso por escrito Daniel Kovacs (Kovacs, 2011), y es la referencia para la sintaxis que usa este capítulo.

16.7. El método MODEL: un valor o muchos

Nuevas preguntas para el paso O (Objetos y estado)

Para cada propiedad dinámica de la tabla de estado, pregunta cuántos valores tiene para un objeto:

- **¿Exactamente uno, siempre?** La habitación de un robot, el poseedor de un artículo, el estado de una puerta. Es una función. Escríbela como fluente de objeto, o como predicado con un invariante que declares en la tabla y pongas a prueba.
- **¿Como mucho uno?** El color de una región, el artículo de una pinza. Una función que puede estar indefinida. Decide quién necesita preguntar si tiene valor.
- **¿Cualquier número?** Las habitaciones que ha visitado un robot, los artículos de un carro, las habitaciones conectadas con una habitación. Es una relación: un predicado, y no vale otra cosa.
- **¿Un número?** Un fluente numérico (Capítulo 8).

La tabla de estado gana una columna, *valores por objeto*: uno, como mucho uno, muchos. Cada «uno» escrito como predicado es un invariante, y el paso E le debe una prueba.

Alternativas comparadas

Tres codificaciones de dónde están las cosas en el mundo del mensajero, todas en los ejemplos del capítulo y todas ejecutadas sobre la misma entrega: los **predicados** del Capítulo 4 (`courier-domain.pddl`), los **fluentes de objeto** con todos los parámetros (`courier-of`) y los fluentes usados **como términos** (`courier-terms`).

	Predicados	Fluentes de objeto	Fluentes como términos
Fidelidad	Correcta solo mientras cada acción mantenga el invariante; un borrado olvidado da planes imposibles	El invariante no puede romperse	Igual
Tamaño	18 hechos, 30 acciones instanciadas	1 hecho, 2 fluentes, 30 acciones instanciadas	23 acciones instanciadas; menos parámetros
Guía	3 variables, después de que el planificador encuentre 2 grupos mutex	3 variables, dadas	Igual
Claridad	Familiar; el invariante es implícito	La declaración documenta el invariante	La más breve; los planes dicen menos (`(move r1 lobby)`)
Portabilidad	Todos los planificadores	`:object-fluents`: de los planificadores comparados con Kairos, solo Kairos	Igual

Tabla 16.1.: Predicados frente a fluentes de objeto para la posición del robot y de los artículos. Las cifras son las de `kairos -v` para la entrega de `p01`.

Veredicto. En todos los criterios menos el último, el fluente es igual o mejor, y en Fidelidad es mejor de la forma que más importa: desaparece toda una clase de errores. El último criterio es el que decide en la práctica. La comparación de planificadores de la documentación de Kairos (Martín Rico et al., 2026) no encuentra, entre los probados, ningún otro planificador que acepte fluentes de objeto, y un modelo que tenga que funcionar en otro sitio tiene que usar predicados. Así que piensa siempre en fluentes, porque la pregunta «un valor o muchos» pertenece al paso O sea cual sea la sintaxis. Escribe fluentes cuando el planificador sea Kairos u otro que admita PDDL 3.1 por completo. En otro caso, escribe predicados, y entonces apunta cada propiedad de un solo valor como invariante y ponla a prueba; el recuento de variables y grupos mutex de `kairos -v` es una prueba barata.

16.8. PDDL 3 de un vistazo

Este capítulo cierra la Parte V. PDDL 3.0 añadió al lenguaje dos formas de hablar de la calidad y de la forma de un plan, y PDDL 3.1 añadió el fluente de este capítulo y puso orden en los requisitos. La Tabla 16.2 los reúne.

Característica	Requisito	Versión	Dónde
Objetivos blandos, preferencias en precondiciones, `is-violated`	`:preferences`	3.0	Capítulo 14
Restricciones de trayectoria, en el dominio y en el problema	`:constraints`	3.0	Capítulo 15
Restricciones blandas (una preferencia sobre una restricción)	las dos anteriores	3.0	Capítulo 15
Funciones con objetos como valores, `undefined`	`:object-fluents`	3.1	Capítulo 16
Funciones con números como valores (el antiguo `:fluents`)	`:numeric-fluents`	3.1	Capítulo 8
Fluentes numéricos y de objeto juntos	`:fluents`	3.1	
`total-cost` incrementado por las acciones	`:action-costs`	3.1	Capítulo 9

Tabla 16.2.: Las características de PDDL 3.0 y 3.1, sus requisitos y dónde las trata el libro.

Los tres capítulos de esta parte comparten una idea. Las preferencias, las restricciones y los fluentes de objeto no añaden ninguna clase nueva de acción ni de mundo: todo lo que describen podría codificarse, y durante años se codificó, con las herramientas de las Partes II y III, como costes, como predicados de contabilidad, como invariantes mantenidos a mano. Lo que añaden es un sitio donde decir directamente lo que el modelador quiere decir: esto es un deseo, esto es una regla, esto tiene un solo valor. Un modelo que dice lo que quiere decir es más corto, y sus errores se ven mejor. El coste es que menos planificadores lo leen, y el criterio de Portabilidad ha tenido la última palabra en cada comparativa.

Con la Parte VI, el lenguaje vuelve a crecer en la otra dirección. PDDL+ trae una clase nueva de mundo: uno que cambia por sí solo.

16.9. Errores comunes

Un valor del tipo equivocado

Un fluente tiene un tipo, y `kairos check` lo hace cumplir en las asignaciones, en los valores iniciales y en las comparaciones. `courier-of-badtype-domain.pddl` asigna un artículo donde se espera un lugar:

```
$ kairos check courier-of-badtype-domain.pddl
courier-of-badtype-domain.pddl:28:18: error: in action 'pick':
     object function 'location-of' returns location [semantic.
     type_mismatch]
 28 |     :effect (and (assign (location-of ?r) ?i) (not (free ?
     r))))
    |                  ^
invalid
```

Es el fluente trabajando para el modelador: el mismo despiste con predicados, `(robot-at ?r ?i)`, solo se detecta si el predicado también tiene tipos. *Solución*: lee el mensaje; cuando un fluente tenga que tomar valores de dos tipos, como hace `holder`, dales un supertipo común.

Dos valores iniciales

Un fluente tiene un valor, así que `:init` solo puede darle uno. `courier-of-twice.pddl` pone el robot en la base de carga y en el vestíbulo:

```
$ kairos check courier-of-domain.pddl courier-of-twice.pddl
courier-of-twice.pddl:16:31: warning: '(location-of r1)'
     initialized with two values; the last one is used [semantic.
     conflicting_init]
 16 |     (= (location-of r1) dock) (= (location-of r1) lobby)
    |                               ^
valid, plannable
...
```

Un aviso, y un plan que parte del vestíbulo. Con predicados, el mismo fichero habría puesto el robot en dos habitaciones sin decir nada. *Solución*: lee los avisos; los ficheros de problema que genera un programa son donde aparece este.

Un fluente que nadie inicializó

Dejar `(= (location-of r1) dock)` fuera de `:init` no es un error: el fluente está indefinido, toda precondición que lo comprueba es falsa y el planificador responde `unsolvable` tras explorar un solo estado. Nada señala la línea que falta. *Solución*: cuando un problema con fluentes es irresoluble de inmediato, comprueba primero que cada fluente de cada objeto tiene su valor inicial, salvo que deba empezar indefinido.

Un fluente para algo que es una relación

Después de este capítulo, todo parece una función. «La habitación conectada con el vestíbulo» no lo es: el vestíbulo tiene tres vecinas. «El artículo que lleva un robot» solo lo es mientras el robot no tenga cesta. Una propiedad metida a la fuerza en un fluente pierde los valores que no caben, y el modelo pierde soluciones. *Solución*: la

pregunta del paso O, hecha con honestidad: exactamente uno, como mucho uno o muchos. Solo los dos primeros son fluentes.

16.10. Resumen

Un fluente de objeto es una función de objetos en un objeto, con como mucho un valor en cada estado. Se declara en `:functions` con el tipo de su valor, se inicializa y se comprueba con `=` y se cambia con `assign`; el requisito es `:object-fluents`.

- Un fluente sustituye a un predicado y al invariante «un valor por objeto» que el predicado necesitaba. No hay ningún borrado que olvidar.
- Un fluente es un término: puede ser un argumento o un lado de una igualdad, y los parámetros que solo llevaban su valor pueden eliminarse.
- Un fluente puede estar indefinido, desde el principio o por asignación. `undefined` no puede comprobarse; prefiere fluentes que siempre tengan valor.
- Los planificadores construyen variables de estado a partir de predicados encontrando grupos mutex. Un fluente les da la variable.
- El paso O pregunta de cada propiedad: un valor, como mucho uno o muchos. Pocos planificadores aceptan fluentes de objeto, así que la respuesta suele escribirse como un predicado con un invariante puesto a prueba.

16.11. Lecturas complementarias

La gramática de los fluentes de objeto, con `assign` y `undefined`, está en la BNF de PDDL 3.1 de Kovacs (2011). El libro de Haslum et al. (2019) trata los fluentes de objeto dentro de su exposición de las variables de estado de PDDL. Para lo que hacen los planificadores por dentro, la descripción de Fast Downward (Helmert, 2006) explica la representación de dominio finito y por qué las heurísticas se benefician de ella, y Helmert (2009) da el algoritmo que encuentra los invariantes en un modelo escrito con predicados. La visión de la planificación basada en variables de estado es también la que adoptan Ghallab et al. (2016) desde su primer capítulo.

16.12. Ejercicios

Ejercicios de la Sección 16.1 — Un robot en dos habitaciones

Ejercicio 16.1.1. El dominio del mensajero con tipos (`courier-domain.pddl`) mantiene tres invariantes con sus efectos: un robot está en una habitación; un artículo está en una habitación o lo lleva un robot; un robot está libre exactamente cuando no lleva nada. Para cada uno, nombra los efectos de cada acción que lo mantienen.

Ejercicio 16.1.2. En `courier-twin-domain.pddl` el robot se multiplica. Rompe en su lugar el segundo invariante: quita `(not (item-at ?i ?l))` de `pick` en una copia del dominio con tipos y describe, antes

de ejecutar nada, un objetivo para el que el planificador devolvería un plan imposible.

Ejercicios de la Sección 16.2 — Una función cuyo valor es un objeto

Ejercicio 16.2.1. En `courier-of`, el poseedor de un artículo puede ser un robot, así que un objetivo puede pedirlo. Escribe un problema con dos robots, `r1` en la base de carga y `r2` en el laboratorio, y una muestra en la sala A, cuyo objetivo sea que `r2` tenga la muestra y esté en el laboratorio. ¿Qué robot va a buscarla?

Ejercicio 16.2.2. `(free ?r)` repite lo que `holder` ya sabe: un robot está libre cuando no es el poseedor de ningún artículo. Escribe un dominio `courier-nofree` sin ese predicado. ¿Es segura la nueva condición respecto a los valores indefinidos?

Ejercicios de la Sección 16.3 — Fluentes como términos

Ejercicio 16.3.1. Para un robot, ocho habitaciones con sus 14 conexiones y un artículo, cuenta a mano las acciones instanciadas de `courier-of` que sobreviven a la precondición estática `connected`, acción por acción, y compáralas con las 30 de las que informa Kairos. Después explica de dónde salen las 23 de `courier-terms`.

Ejercicio 16.3.2. Dos robots nunca deben estar en la misma habitación. Amplía `courier-of` para que `move` se niegue a entrar en una habitación donde haya otro robot, comparando los fluentes de los dos robots. Pruébalo con `r2` parado en el pasillo, en medio de una entrega que haría `r1`, y lee el plan.

Ejercicios de la Sección 16.4 — Sin valor: indefinido

Ejercicio 16.4.1. En `courier-cargo`, ¿podría terminar el plan con el objetivo `(= (room-of sample1) lab)` cumplido mientras el robot todavía lleva `sample1`? ¿Por qué no? ¿Qué haría falta para pedir, en un objetivo, que el robot termine sin llevar nada?

Ejercicio 16.4.2. Compara `holder` (total) y `room-of` con `carrying` (parcial) con el paso L: ¿qué invariantes deja cada modelo en manos de las acciones?

Ejercicios de la Sección 16.5 — Dos mundos más

Ejercicio 16.5.1. Escribe un problema para `blocks-of` con dos torres, `b` sobre `a` y `d` sobre `c`, cuyo objetivo sea las dos torres invertidas. ¿Cuántas acciones tiene el mejor plan? Si Kairos imprime primero un plan más largo, explica por qué.

Ejercicio 16.5.2. Colorea con tres colores el mapa de Australia: Australia Occidental, Territorio del Norte, Australia Meridional, Queensland, Nueva Gales del Sur, Victoria y Tasmania. (Australia Meridional limita con todas las regiones del continente.) ¿Qué hace el planificador con Tasmania?

Ejercicios de la Sección 16.6 — Variables de estado: lo que ve el planificador

Ejercicio 16.6.1. Ejecuta `kairos -v` con los problemas `courier-p02` y `courier-twin-p02`, y explica los dos recuentos de variables y grupos mutex. ¿Qué grupo mutex sobrevive en el dominio con el error?

Ejercicios de la Sección 16.7 — El método MODEL: un valor o muchos

Ejercicio 16.7.1. Clasifica cada propiedad como predicado, fluente de objeto, fluente de objeto que puede estar indefinido o fluente numérico: la planta en la que está un ascensor; las personas que hay dentro de un ascensor; el estado de una puerta (abierta, cerrada, con llave); qué habitaciones une una puerta; el nivel de batería; el carro que empuja un robot; los artículos de un carro; el robot que se está cargando en el único cargador.

Ejercicio 16.7.2. Las puertas del Capítulo 5 tenían los predicados `(open ?d)` y `(locked ?d)`. Modela una puerta con un único fluente `(state-of ?d)` cuyos valores sean tres constantes, con acciones para abrir la cerradura, abrir y cerrar, y un problema. ¿Qué estado imposible del modelo con predicados ya no puede escribirse?

Ejercicios de la Sección 16.8 — PDDL 3 de un vistazo

Ejercicio 16.8.1. Para cada necesidad, nombra la característica de la Parte V y su requisito: (a) el robot debería evitar el vestíbulo, pero puede cruzarlo; (b) el robot nunca debe estar en el vestíbulo con una muestra; (c) cada artículo tiene exactamente un poseedor; (d) se piden tres entregas y la batería da para dos; (e) la entrega debería hacerse en los diez primeros minutos, y es malo, pero no fatal, que no sea así.

Parte VI. Sistemas híbridos: PDDL+

17. Procesos y eventos

En todos los modelos hasta ahora, el mundo solo cambiaba porque el agente hacía algo. Un hecho se volvía cierto como efecto de una acción; un número subía o bajaba mientras se ejecutaba una acción durativa. Incluso la batería del Capítulo 12, que se descargaba de forma continua, solo se descargaba durante un `move`, y dejaba de hacerlo en el momento en que el desplazamiento terminaba. Dejemos ese robot en el pasillo, encendido, sin nada que hacer durante una hora, y su batería estará exactamente como estaba: no se ejecuta ninguna acción, así que nada cambia. Un robot real ya estaría sin batería. Una muestra que se deja en una mesa se calienta tanto si alguien planea llevarla como si no, un depósito sigue llenándose después de que el agente se haya puesto con otra cosa, y una pelota que se suelta cae sin pedir permiso. Los literales iniciales temporizados del Capítulo 13 dejan que el mundo cambie por sí solo, pero solo en instantes escritos en el problema. Aquí los cambios dependen del estado: *mientras* el robot está encendido, *cuando* el depósito está lleno.

Este capítulo da el paso de «el agente cambia el mundo» a «el mundo cambia por sí solo». PDDL+ añade dos construcciones al lenguaje. Un *proceso* es un cambio continuo que dura mientras se cumple su condición, sin que nadie lo ponga en marcha y sin que nadie pueda rechazarlo. Un *evento* es un cambio instantáneo que ocurre en el momento en que su condición se vuelve cierta, de nuevo sin que nadie elija. Las acciones siguen en el lenguaje, pero su papel se estrecha: son las únicas decisiones del agente, y todo lo demás se sigue de las leyes del mundo. Con las tres juntas modelamos un depósito que se llena y se desborda, una pelota que cae y rebota, un coche que el conductor solo gobierna a través de su aceleración y un robot mensajero cuya batería se descarga siempre que está encendido y cuyas muestras se estropean si se entretiene.

El capítulo empieza por el proceso y el evento más sencillos, en un depósito de agua, y lee sus planes, que tienen un aspecto distinto de los que conocemos. Después pone las tres clases de cambio una junto a otra y las sigue en dos dominios clásicos, la pelota que rebota y el coche. Viene luego el mundo del mensajero: desplazarse en PDDL+, la batería y una muestra que se calienta. Una sección explica cómo planifica y valida Kairos estos modelos, porque lo que un planificador puede y no puede encontrar importa aquí más que en ningún otro lugar del libro. El capítulo termina con lo que PDDL+ añade al método MODEL, los errores habituales y ejercicios.

17.1 Un mundo que no espera 383
17.2 Procesos: cambio mientras se cumple una condición 384
El primer plan 386
El plan no es único, y el instante tampoco 387
17.3 Eventos: cambio en un instante 387
Un evento debe desactivarse a sí mismo 388
Un problema sin plan . 388
17.4 Tres clases de cambio . 389
La pelota que rebota . . 390
El coche 392
17.5 El mundo del mensajero en PDDL+ 394
Un desplazamiento es una acción, un proceso y un evento 394
La batería: un proceso y un evento 395
Ensayar el modelo . . . 395
La prueba de la trampa: un robot que avanza por inercia 396
Una muestra que se calienta 397
17.6 Cómo planifica Kairos con PDDL+ 399
Búsqueda en pasos de tiempo 399
Validación en tiempo continuo 400
Lo que se le puede escapar 400
17.7 El método MODEL: quién causa cada cambio 401
Alternativas comparadas 402
17.8 Errores comunes 403
17.9 Resumen 404
17.10 Lecturas complementarias 405
17.11 Ejercicios 405

17.1. Un mundo que no espera

Tomemos un depósito de agua con una válvula. El agente abre la válvula, el agua entra a dos litros por segundo y el agente quiere ocho litros. Con las herramientas de la Parte IV escribiríamos una acción durativa `fill` cuya duración elige el planificador, con un efecto continuo sobre el nivel. Ese modelo es correcto mientras la historia trate de llenar. Deja de serlo en cuanto preguntamos qué ocurre si el agente se olvida de la válvula: en el modelo, nada, porque una acción durativa siempre termina, y cuando termina el agua se detiene. El modelo no tiene forma de decir «el agua

sigue entrando hasta que alguien cierre la válvula», y por eso tampoco puede decir «y entonces el depósito se desborda».

El problema no es el efecto continuo. Es *de quién es el cambio*. En una acción durativa, el cambio pertenece al agente: empieza cuando el agente empieza la acción y no puede sobrevivir a ella. En el mundo, el cambio pertenece al depósito: el agua fluye mientras la válvula está abierta, y la válvula es un hecho del estado, no una actividad del agente. El agente decide dos cosas, cuándo abrir y cuándo cerrar. Entre esas dos decisiones, el mundo funciona por su cuenta.

En PDDL+ el agente solo decide. El cambio que avanza por sí solo es un proceso; el cambio que ocurre por sí solo en un instante es un evento.

Un sistema con estado discreto (la válvula está abierta o cerrada) y estado continuo (el nivel) que evoluciona en el tiempo se llama **sistema híbrido**. **PDDL+** es la extensión de PDDL para sistemas híbridos, definida por Maria Fox y Derek Long Fox y Long, 2006. Conserva todo PDDL 2.1 y añade, junto a las acciones, dos clases de cosas que pueden ocurrir:

- Un **proceso** cambia números de forma continua, mientras se cumpla su condición. No se elige: siempre que la condición es cierta, el proceso está en marcha.
- Un **evento** cambia el estado en un instante, el primer instante en que su condición es cierta. Tampoco se elige: cuando la condición se vuelve cierta, el evento se dispara.

Los procesos y los eventos son las leyes del mundo. Las acciones son lo que le queda al agente, y en un modelo PDDL+ puro son instantáneas: abrir una válvula, soltar una pelota, pisar un pedal. Un plan es la lista de esas decisiones con sus instantes. Lo que hace el mundo entre medias no está en el plan, porque no es una decisión; se sigue del modelo.

De PDDL 2.1 a PDDL+

PDDL 2.1 Fox y Long, 2003 se diseñó por niveles, y sus autores, Maria Fox y Derek Long, ya tenían en mente un último nivel en el que el cambio pudiera ponerlo en marcha el mundo y no solo el agente. Ese nivel no se usó en la competición de 2002. Fox y Long lo publicaron completo en 2006 en el *Journal of Artificial Intelligence Research*, con el título *Modelling Mixed Discrete-Continuous Domains for Planning* Fox y Long, 2006. El artículo da a PDDL+ un significado formal traduciendo cada dominio a un *autómata híbrido*, el modelo que los ingenieros de control y los informáticos usan para describir y verificar sistemas como termostatos, motores y controladores de vuelo. Un plan PDDL+ es, en esos términos, una manera de llevar el autómata hasta un objetivo.

17.2. Procesos: cambio mientras se cumple una condición

El Listado 17.1 es el depósito. Léelo en tres bloques: lo que decide el agente (dos acciones), lo que hace el mundo mientras se cumple una condición (un proceso) y lo que hace el mundo en un instante (un evento, que explica la sección siguiente).

```
;; A water tank: the first PDDL+ model of the book.
;; The agent only opens and closes a valve; the water does the
;; rest.
(define (domain tank)
  (:requirements :negative-preconditions :numeric-fluents :time)

```

```
  (:predicates
    (valve-open)              ; water is coming in
    (flooded))                ; water went over the edge

  (:functions
    (level)                   ; liters in the tank
    (capacity)                ; liters it can hold
    (inflow))                 ; liters per second coming in

  ;; What the agent decides
  (:action open-valve
    :parameters ()
    :precondition (not (valve-open))
    :effect (valve-open))

  (:action close-valve
    :parameters ()
    :precondition (valve-open)
    :effect (not (valve-open)))

  ;; What the world does while the valve is open
  (:process fill
    :parameters ()
    :precondition (valve-open)
    :effect (increase (level) (* #t (inflow))))

  ;; What the world does at the instant the tank is full
  (:event overflow
    :parameters ()
    :precondition (and (>= (level) (capacity)) (not (flooded)))
    :effect (flooded))
```

Listado 17.1: El depósito: dos acciones, un proceso y un evento (`examples/ch17/tank-domain.pddl`).

Un proceso se escribe como una acción, con la palabra clave `:process`:

```
  (:process fill
    :parameters ()
    :precondition (valve-open)
    :effect (increase (level) (* #t (inflow))))
```

Las diferencias con una acción están en el significado, no en la sintaxis.

- La `:precondition` no es algo que el planificador deba conseguir para usar el proceso. Es la condición bajo la cual el proceso *está en marcha*. Mientras `(valve-open)` es cierto, `fill` funciona; cuando es falso, `fill` no funciona.
- El `:effect` solo puede ser continuo: `increase` o `decrease` de un fluente numérico en una expresión multiplicada por `#t`, como en los efectos continuos del Capítulo 12. Aquí `#t` es el tiempo que lleva en marcha el proceso, y `(inflow)` es el **ritmo**: litros por segundo. Un proceso no puede añadir ni borrar un hecho, ni hacer `assign`.
- Un proceso nunca aparece en un plan. Nadie lo aplica.

El requisito para procesos y eventos es `:time`[1], junto con `:numeric-fluents`, porque un proceso tiene que cambiar algún número. El dominio usa también `:negative-preconditions`, para `(not (valve-open))`.

1: Algunos ficheros PDDL+ que circulan por ahí declaran `:processes` y `:events`, o nada en absoluto. Kairos acepta un dominio con procesos y sin `:time`, con un aviso (Sección 17.8).

Fijémonos en cómo controla el agente el proceso: no lo controla. Controla el hecho `(valve-open)`, con dos acciones ordinarias, y el hecho enciende y apaga el proceso. Este control indirecto es el patrón de todo modelo PDDL+: *las acciones cambian hechos, los hechos habilitan procesos, los procesos cambian números*.

Un proceso es una acción que nadie elige: su precondición dice cuándo funciona, y su efecto, siempre con `#t`, a qué ritmo cambia un número.

El primer plan

El primer problema pide al menos ocho litros, con la válvula cerrada al final y sin inundación.

```
;; Fill the tank to at least 8 liters and leave the valve closed.
(define (problem tank-p01)
  (:domain tank)
  (:init
    (= (level) 0) (= (capacity) 10) (= (inflow) 2))
  (:goal (and (>= (level) 8) (not (valve-open))
              (not (flooded)))))
```

Listado 17.2: Ocho litros, válvula cerrada, sin inundación (`examples/ch17/tank-p01.pddl`).

```
$ kairos tank-domain.pddl tank-p01.pddl
; cost = 4
; makespan = 4.000
0.000: (open-valve) [0.000]
4.000: (close-valve) [0.000]
4.000: @PlanEND
note: the search minimizes the number of actions; the metric of the
     problem is reported for each plan but does not guide the search
      yet [orchestrator.metric]
```

El plan tiene el formato de los planes temporales del Capítulo 10: un instante, una acción y una duración entre corchetes, que es cero porque las dos acciones son instantáneas. Hay tres cosas nuevas.

Primero, el hueco. No hay nada escrito entre 0 y 4 segundos, y sin embargo es ahí donde el plan hace su trabajo: la válvula está abierta, así que `fill` funciona, y el nivel pasa de 0 a $2 \times 4 = 8$ litros. El plan dice *cuándo* actúa el agente; el modelo dice qué ocurre mientras tanto. Para leer un plan PDDL+ hay que reproducir los procesos de cabeza, o en papel: la Figura 17.1 lo hace para este plan.

Segundo, la línea `@PlanEND`. Un plan PDDL+ puede necesitar dejar pasar el tiempo después de su última acción, hasta que el mundo alcance el objetivo por sí solo, así que el plan indica dónde termina. Aquí el final coincide con la última acción.

Tercero, la nota. Kairos busca planes con pocas acciones e informa como *coste* de la duración del plan. No minimiza esa duración: veremos planes válidos y más largos de lo necesario. La nota se imprime con cada plan PDDL+ y a partir de aquí la omitimos en los listados.[2]

2: Las salidas de este capítulo y del siguiente se obtuvieron con las opciones del guion de pruebas del libro, en particular `-threads 1`. Con más hilos, Kairos ejecuta varias búsquedas a la vez y puede imprimir otros planes, tan válidos como estos.

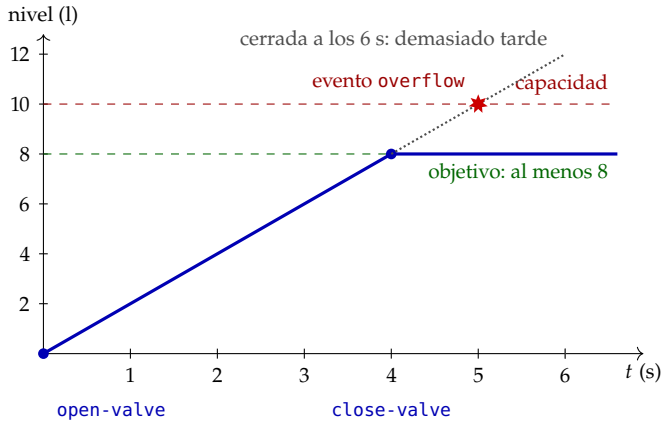


Figura 17.1.: El nivel del depósito a lo largo del tiempo. Línea gruesa: el plan de Kairos. La pendiente entre las dos acciones es el ritmo del proceso `fill`. De puntos: el mismo plan con la válvula cerrada dos segundos más tarde; por el camino, el nivel cruza la capacidad y el evento se dispara.

El plan no es único, y el instante tampoco

Cerrar la válvula a los 4 segundos es el primer momento que alcanza el objetivo, pero no el único. Vale cualquier instante desde 4 segundos hasta justo antes de 5. Podemos comprobarlo con un plan escrito a mano: el validador de Kairos reproduce cualquier plan en tiempo continuo.

```
$ cat tank-p01-wait.plan
; A plan written by hand: closing at 4.5 s is as good as at 4 s.
0.000: (open-valve)
4.500: (close-valve)
$ kairos validate tank-domain.pddl tank-p01.pddl tank-p01-wait.plan
plan valid, makespan 4.5
```

En planificación clásica, un plan era una elección de *qué* acciones. Con las acciones durativas pasó a ser también una elección de *orden y solapamiento*. En PDDL+, los propios instantes son el plan: las mismas dos acciones resuelven el problema a los 4,5 segundos y fallan a los 3 o a los 6. El planificador busca entre instantes, y por eso planificar en PDDL+ es caro y este capítulo mantiene pequeños sus problemas.

17.3. Eventos: cambio en un instante

La segunda ley del depósito es la inundación.

```
(:event overflow
  :parameters ()
  :precondition (and (>= (level) (capacity)) (not (flooded)))
  :effect (flooded))
```

Un evento tiene la sintaxis de una acción y los efectos de una acción: puede añadir y borrar hechos, y aplicar `assign`, `increase` o `decrease` a números en una cantidad fija. Lo que cambia es, de nuevo, quién decide. Un evento **se dispara** en el primer instante en que su precondición es cierta. El planificador no puede aplicarlo antes, ni retrasarlo, ni saltárselo.

Un evento es una acción que nadie elige y nadie puede evitar: se dispara en el instante en que su condición se vuelve cierta. El agente solo puede impedir que esa condición llegue a ser cierta.

Los eventos son la forma en que un modelo PDDL+ dice que algo sale mal, o simplemente que algo ocurre. El planificador evita un evento malo de la única manera que puede: actuando de modo que su condición no se cumpla nunca. En el plan de la sección anterior, la válvula se cierra a los 4 segundos, el nivel se queda en 8 y `overflow` no se dispara nunca. Si se cierra demasiado tarde, se dispara:

```
$ cat tank-p01-late.plan
; A plan written by hand: the valve is closed one second too late.
0.000: (open-valve)
6.000: (close-valve)
$ kairos validate tank-domain.pddl tank-p01.pddl tank-p01-late.plan
tank-p01-late.plan: error: the goal does not hold at the end of the
     plan: (flooded) is true [validate.goal_not_reached]
plan invalid
```

Las dos acciones de ese plan eran aplicables. Nada en él es «ilegal». Falla porque, a los 5 segundos, el nivel alcanzó la capacidad y el mundo añadió `(flooded)`, que el objetivo prohíbe (la línea de puntos de la Figura 17.1). Esta es la forma habitual de excluir un resultado malo con un evento: el evento deja una marca, y el objetivo (o la precondición de las acciones

que importan) rechaza la marca. El Capítulo 18 muestra otra forma, la restricción global.

Un evento debe desactivarse a sí mismo

Fijémonos en el segundo término de la conjunción de la precondición, `(not (flooded))`. Sin él, la condición del evento sería cierta a los 5 segundos, el evento se dispararía y la condición seguiría siendo cierta: el nivel sigue en la capacidad. Tendría que dispararse otra vez, y otra, infinitas veces en el mismo instante. PDDL+ lo prohíbe: **un evento debe hacer falsa su propia precondición**. La forma más sencilla es la que se usa aquí, un hecho que el evento añade y que su precondición niega. Así, el evento se dispara una sola vez.

Un evento también puede ser el objetivo del agente. El problema `tank-p02` pide la inundación:

```
$ kairos tank-domain.pddl tank-p02.pddl
; cost = 5
; makespan = 5.000
0.000: (open-valve) [0.000]
5.000: @PlanEND
```

Una acción y, después, cinco segundos en los que el agente no hace nada. El plan termina en `@PlanEND`, cuando el evento se ha disparado. Aquí el final del plan importa: un plan con `open-valve` sola, que terminara en el instante cero, no alcanzaría el objetivo.

Un problema sin plan

El problema `tank-p03` pide 12 litros en un depósito de 10 sin inundación. Ningún momento de apertura y cierre de la válvula lo consigue.

```
$ kairos tank-domain.pddl tank-p03.pddl
note: no plan in steps of time; a plan that needs finer timing than
     the steps would be missed, so this is evidence but not a proof
     [orchestrator.steps_exhausted]
note: no plan: the engines that apply ran out of states (408 states)
      before the time limit, so more time does not help; none of
     them is complete for this task, so a plan may still exist (see
     the note before). [orchestrator.why]
kairos: no plan found (exhausted)
```

La respuesta no es el `unsolvable` de los capítulos clásicos, y el código de salida es 14, no 10. Kairos dice *no plan found*: ha probado todas las formas de temporizar la válvula que contempla y ninguna ha funcionado, pero no contempla todos los instantes, así que no afirma tener una demostración. La Sección 17.6 explica qué prueba exactamente. Para la batería de pruebas del paso E, esto significa que la prueba «sin plan» se comprueba, en PDDL+, como «Kairos no encuentra ninguno», y que conviene convencerse también razonando, como acabamos de hacer.

17.4. Tres clases de cambio

La Tabla 17.1 pone las tres construcciones una junto a otra. La sintaxis es casi la misma; las filas que difieren son las que tratan de la elección.

	Acción	**Proceso**	**Evento**
Quién lo causa	el agente	el mundo	el mundo
Cuándo	cuando el planificador decide, si se cumple la precondición	todo el tiempo que se cumple la precondición	en el primer instante en que se cumple la precondición
Cuánto dura	un instante	lo que dure la condición	un instante
Efectos	hechos y números, a saltos	solo números, de forma continua (`#t`)	hechos y números, a saltos
En el plan	sí, con su instante	no	no

Tabla 17.1.: Acciones, procesos y eventos.

Las tres se alimentan unas a otras en un ciclo, dibujado en la Figura 17.2. Una acción cambia un hecho. El hecho hace cierta la condición de un proceso, y el proceso empieza a mover un número. El número alcanza un umbral, lo que hace cierta la condición de un evento, y el evento cambia hechos. Esos hechos pueden detener el proceso, poner en marcha otro o habilitar una acción que el agente estaba esperando.

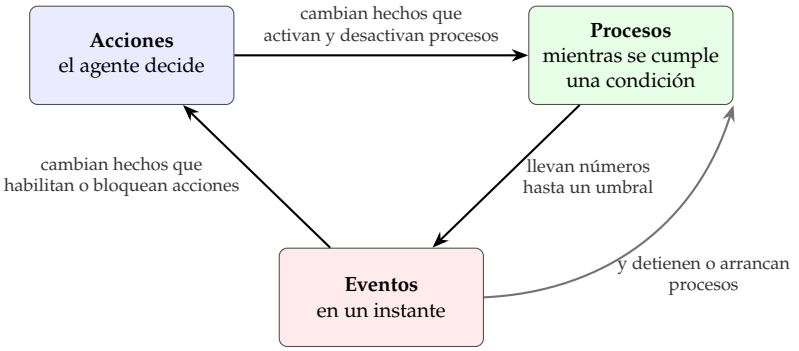


Figura 17.2.: Cómo se alimentan unas a otras las tres clases de cambio. Solo la caja de la izquierda está bajo el control del planificador; las otras dos son las leyes del mundo.

Qué ocurre entre dos acciones

Entre una decisión del agente y la siguiente, un estado PDDL+ evoluciona por sí solo, y las reglas son pocas. (1) Mientras no sea cierta la condición de ningún evento, el tiempo fluye: cada proceso cuya condición se cumple cambia sus números a su ritmo, y los hechos se quedan como están. (2) En el momento en que la condición de algún evento se vuelve cierta, el tiempo se detiene, el evento se dispara y se aplican sus efectos. Esto puede hacer ciertas las condiciones de otros eventos; se disparan también, en el mismo instante, hasta que no pueda dispararse ningún evento. (3) Después el tiempo vuelve a fluir, con los procesos que habilita el nuevo estado. Una acción del plan se aplica en su instante al estado al que ha llegado esta evolución, y sus efectos pueden disparar de nuevo eventos antes de que el tiempo siga. Si dos procesos cambian el mismo número, sus ritmos se suman. Si un estado permitiera que los eventos se disparasen sin fin en un instante, el que está mal es el modelo, no el plan.

Una forma útil de recordar la diferencia con el Capítulo 12: una acción durativa con un efecto continuo es un proceso cuyo inicio elige el agente

y cuyo final llega en un instante fijado al empezar, o elegido. Un proceso PDDL+ empieza y termina cuando lo dice el *estado*. Para dar al agente el control que tenía una acción durativa, los modelos PDDL+ usan tres piezas: una acción que inicia la actividad (añade un hecho), un proceso que funciona mientras se cumple el hecho y una acción o un evento que la termina (borra el hecho). Este patrón **inicio-proceso-parada** está en el depósito (`open-valve`, `fill`, `close-valve`) y estará en todos los modelos del capítulo.

La pelota que rebota

La pelota que rebota es el ejemplo más antiguo de sistema híbrido: una ley continua (la gravedad) interrumpida por una instantánea (el rebote). En el Listado 17.3, el agente sostiene una pelota a cinco metros, la suelta y debe cogerla después con una mano que espera a dos metros.

```
;; The bouncing ball: a process makes it fall, an event makes it
;; bounce. The agent lets it go and catches it.
(define (domain ball)
  (:requirements :negative-preconditions :numeric-fluents :time)

  (:predicates
    (held)                    ; the ball is in the hand
    (falling))                ; the ball is in the air

  (:functions
    (height)                  ; meters above the floor
    (velocity)                ; meters per second, positive up
    (gravity)                 ; meters per second squared
    (restitution)             ; share of the speed a bounce keeps
    (bounces))                ; how many times it hit the floor

  (:action release
    :parameters ()
    :precondition (held)
    :effect (and (not (held)) (falling)))

  ;; The hand waits at 2 m: the ball must pass slowly enough
  (:action catch
    :parameters ()
    :precondition (and (falling)
                       (>= (height) 1.9) (<= (height) 2.1)
                       (>= (velocity) -2) (<= (velocity) 2))
    :effect (and (not (falling)) (held)
                 (assign (velocity) 0)))

  ;; Two quantities change at once: the height with the velocity,
  ;; the velocity with gravity
  (:process fall
    :parameters ()
    :precondition (falling)
    :effect (and (increase (height) (* #t (velocity)))
                 (decrease (velocity) (* #t (gravity)))))

  ;; On the floor and going down: the velocity turns around
  (:event bounce
    :parameters ()
    :precondition (and (falling) (<= (height) 0)
                       (< (velocity) 0))
    :effect (and (assign (velocity)
                         (* -1 (* (restitution) (velocity))))
                 (increase (bounces) 1))))
```

Listado 17.3: La pelota que rebota (`examples/ch17/ball-domain.pddl`).

Hay tres detalles que merecen atención.

Un proceso, dos efectos. El proceso `fall` cambia la altura al ritmo `(velocity)` y la velocidad al ritmo `(gravity)`, a la vez. El ritmo del primer efecto no es una constante: es un fluente que está cambiando el segundo efecto. Por eso la altura no es una recta en el tiempo, sino una parábola. Los efectos continuos del Capítulo 12 tenían ritmos constantes; un ritmo que a su vez cambia es algo que en PDDL+ obtenemos casi gratis, y el Capítulo 18 lo estudia.

Un evento que asigna. El evento `bounce` se dispara cuando la pelota está en el suelo y bajando. Su efecto invierte la velocidad y conserva el 80 % de ella: `(assign (velocity) (* -1 (* (restitution) (velocity))))`. Después, la velocidad es positiva, así que el término `(< (velocity) 0)` es falso: el evento se ha desactivado a sí mismo sin ningún hecho adicional.

Una acción con una ventana estrecha. El agente solo puede hacer `catch` cuando la pelota está a menos de diez centímetros de la mano y va a menos de 2 m/s. El planificador debe encontrar un instante en el que el mundo, funcionando por sí solo, cumpla eso.

El problema `ball-p01` deja caer la pelota desde 5 m con gravedad 10 y pide tenerla de nuevo en la mano después de al menos dos rebotes.

```
$ kairos ball-domain.pddl ball-p01.pddl
; cost = 3.1
; makespan = 3.100
0.000: (release) [0.000]
3.100: (catch) [0.000]
3.100: @PlanEND
```

Otra vez dos acciones, y todo lo interesante está entre medias. Reproduzcámoslo. Desde 5 m, la pelota tarda un segundo en llegar al suelo ($5 = \frac{1}{2} \cdot 10 \cdot t^2$), y llega a -10 m/s. El evento se dispara: la velocidad pasa a ser $+8$. La pelota sube durante 0,8 s hasta 3,2 m, y vuelve al suelo 1,6 s después del primer rebote, en $t = 2{,}6$, a -8 m/s. Segundo evento: +6,4 m/s. Medio segundo después, en $t = 3{,}1$, la pelota está a $6{,}4 \cdot 0{,}5 - 5 \cdot 0{,}5^2 = 1{,}95$ m y se mueve a $6{,}4 - 5 = 1{,}4$ m/s: dentro de la ventana de `catch`. La Figura 17.3 muestra el vuelo completo.

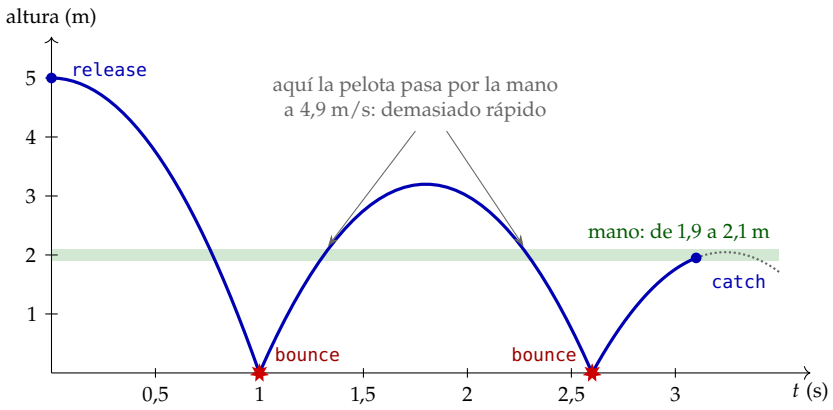


Figura 17.3.: El vuelo de la pelota en el plan de Kairos. Cada arco es el proceso `fall`; las estrellas son el evento `bounce`. La pelota cruza la franja de la mano cinco veces, y solo cerca de lo alto del tercer arco va lo bastante despacio para cogerla.

La figura muestra también por qué el plan no coge la pelota antes. Después del primer rebote, la pelota cruza los dos metros dos veces, pero a 4,9 m/s. Solo el tercer arco, cuya cima está a 2,05 m, pasa despacio por la mano. El planificador no lo «sabía»: encontró el único tramo de tiempo en el que la precondición de `catch` es cierta. Nosotros no habríamos sabido dónde está ese tramo sin hacer la física, y esa es la gracia de un

modelo PDDL+: escribimos las leyes, y el planificador encuentra cuándo actuar.

El coche

El segundo clásico es el coche de Fox y Long Fox y Long, 2006: un coche en una carretera recta cuyo conductor solo puede cambiar la aceleración. El Listado 17.4 es nuestra versión, con un radar.

```
;; A car on a straight road. The driver only changes the
;; acceleration; speed and distance follow by themselves.
(define (domain car)
  (:requirements :negative-preconditions :numeric-fluents :time)

  (:predicates
    (running)                ; the engine is on
    (parked)                 ; stopped, engine off
    (fined))                 ; the radar caught the car

  (:functions
    (distance-run)           ; meters from the start
    (velocity)               ; meters per second
    (acceleration)           ; meters per second squared
    (speed-limit))           ; meters per second

  (:action accelerate
    :parameters ()
    :precondition (and (running) (< (acceleration) 1))
    :effect (increase (acceleration) 1))

  (:action brake
    :parameters ()
    :precondition (and (running) (> (acceleration) -1))
    :effect (decrease (acceleration) 1))

  ;; Parking needs the car at rest and the pedals released
  (:action park
    :parameters ()
    :precondition (and (running) (= (acceleration) 0)
                       (>= (velocity) -0.05) (<= (velocity) 0.05))
    :effect (and (not (running)) (parked)
                 (assign (velocity) 0)))

  (:process moving
    :parameters ()
    :precondition (running)
    :effect (and
      (increase (distance-run) (* #t (velocity)))
      (increase (velocity) (* #t (acceleration)))))

  (:event radar
    :parameters ()
    :precondition (and (> (velocity) (speed-limit))
                       (not (fined)))
    :effect (fined)))
```

Listado 17.4: El coche: el conductor cambia la aceleración y un proceso hace el resto (`examples/ch17/car-domain.pddl`).

Las acciones `accelerate` y `brake` no mueven el coche. Suman o restan una unidad de aceleración, que puede valer −1, 0 o 1. El proceso `moving` convierte la aceleración en velocidad y la velocidad en distancia, mientras el motor esté en marcha. El evento `radar` marca el coche como `fined` si alguna vez supera el límite, y `park` necesita el coche en reposo, con un margen de 5 cm/s, y sin aceleración.

El problema `car-p01` pide recorrer al menos diez metros y aparcar, con un límite de 2 m/s y sin multa. Kairos es un planificador *anytime*: imprime un primer plan y luego otros mejores.

```
$ kairos car-domain.pddl car-p01.pddl
; cost = 33.8
; makespan = 33.800
0.000: (accelerate) [0.000]
0.300: (brake) [0.000]
33.500: (brake) [0.000]
33.800: (accelerate) [0.000]
33.800: (park) [0.000]
33.800: @PlanEND
; better plan 2
; cost = 7.5
; makespan = 7.500
0.000: (accelerate) [0.000]
1.800: (brake) [0.000]
4.800: (brake) [0.000]
5.000: (accelerate) [0.000]
5.600: (brake) [0.000]
5.700: (accelerate) [0.000]
5.800: (brake) [0.000]
6.000: (accelerate) [0.000]
6.200: (brake) [0.000]
7.500: (accelerate) [0.000]
7.500: (park) [0.000]
7.500: @PlanEND
```

El primer plan es correcto y absurdo. El conductor acelera durante tres décimas de segundo, suelta el pedal (el `brake` en 0,3 lleva la aceleración de 1 de vuelta a 0) y después avanza a paso de tortuga, a 0,3 m/s, durante más de medio minuto antes de frenar. Tiene cinco acciones, el mínimo posible, y eso es lo que perseguía la búsqueda. El segundo plan tarda 7,5 segundos: hasta 1,8 m/s, tres segundos de crucero y una frenada interrumpida tres veces. Todas las acciones son aplicables y el radar no se dispara nunca, así que es válido; un profesor de autoescuela no lo firmaría.

¿Podemos hacerlo mejor a mano? El límite sugiere el plan: dos segundos de aceleración para llegar exactamente a 2 m/s (2 m recorridos), tres segundos al límite (6 m más) y dos segundos de frenada (los últimos 2 m). Diez metros en siete segundos:

```
$ kairos validate car-domain.pddl car-p01.pddl car-p01-clean.plan
plan valid, makespan 7
```

La Figura 17.4 dibuja los dos. ¿Y un conductor con prisa, que mantiene el pedal pisado 3,2 segundos y luego frena? El coche recorre 10,24 m en 6,4 s, y el validador responde `(fined) is true`: la velocidad superó el límite en $t = 2$ y el evento se disparó.

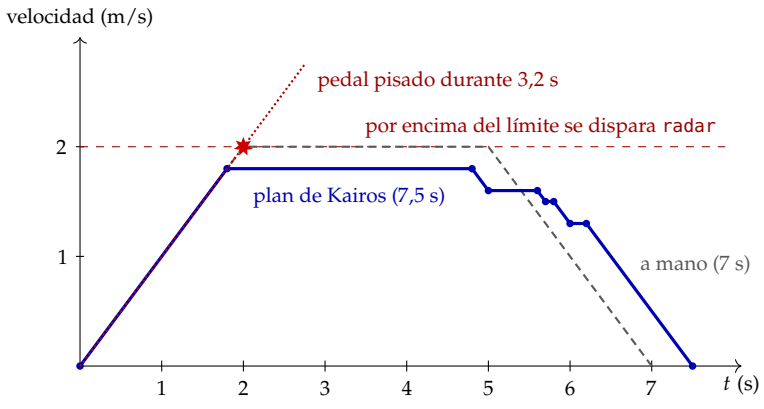


Figura 17.4.: La velocidad del coche en tres planes. Cada punto es una acción del conductor; entre puntos, la pendiente es la aceleración. El área bajo cada línea es la distancia: 10 m en los dos planes válidos.

Paso D para el coche: quién causa cada cambio

La historia dice «el conductor recorre diez metros». El paso D pregunta qué se puede hacer o qué puede ocurrir, y en un modelo híbrido primero hay que partir la historia según la causa. *Lo decide el agente:* pisar y soltar los pedales, aparcar. *Lo hace el mundo mientras se cumple una condición:* el coche gana velocidad mientras hay aceleración y recorre distancia mientras tiene velocidad, siempre que el motor esté en marcha. *Lo hace el mundo en un instante:* el radar hace la foto en el momento en que se supera el límite. Cada línea se convierte en una construcción: tres acciones, un proceso, un evento. Si hubiéramos escrito `drive-ten-meters` como una acción, habríamos dado al agente un poder que no tiene, el de decidir dónde está el coche.

17.5. El mundo del mensajero en PDDL+

El robot mensajero tiene batería desde el Capítulo 8 y, desde el Capítulo 12, una batería que se descarga de forma continua durante un desplazamiento. Ahora la batería se descarga *siempre que el robot está encendido*: moviéndose, esperando, recogiendo o sin hacer nada en absoluto. Esa sola frase cambia el modelo más de lo que parece, porque hace que el tiempo importe para todas las acciones, y por tanto desplazarse debe llevar tiempo de una forma que los procesos puedan ver.

Solo cambian dos líneas de la ficha del modelo. En *Alcance*: la batería se gasta por estar encendido, no por moverse; un robot con la batería vacía se apaga donde esté. En *Supuestos*: la descarga es constante mientras el robot está encendido, y la velocidad es constante mientras se mueve.

Un desplazamiento es una acción, un proceso y un evento

En un modelo PDDL+ puro no hay acciones durativas, así que un desplazamiento que lleva tiempo se construye con el patrón inicio-proceso-parada. El Listado 17.5 muestra las tres piezas.

```
;; --- The decisions of the agent: instantaneous actions ---
(:action start-move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (powered ?r) (robot-at ?r ?from)
                     (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from)) (moving ?r ?from ?to)
               (assign (traveled ?r) 0)))
;; --- What the world does: processes and events ---
;; The robot advances while it is on its way and on
(:process travel
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (moving ?r ?from ?to) (powered ?r))
  :effect (increase (traveled ?r) (* #t (speed ?r))))

;; It arrives at the instant it has covered the distance
(:event arrive
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (moving ?r ?from ?to)
                     (>= (traveled ?r) (distance ?from ?to)))
  :effect (and (not (moving ?r ?from ?to)) (robot-at ?r ?to)))
```

- La acción `start-move` es la decisión. Saca al robot de `?from`, registra que está `(moving ?r ?from ?to)` y pone a cero el cuentakilómetros `(traveled ?r)`. Mientras se mueve, el robot no está en ninguna

Listado 17.5: Desplazarse en PDDL+: el agente inicia el desplazamiento, un proceso hace avanzar al robot y un evento termina el desplazamiento (parte del dominio `courier-battery`).

habitación: no se cumple ningún hecho `robot-at`, así que no puede recoger, dejar ni iniciar otro desplazamiento.
- El proceso `travel` hace avanzar el cuentakilómetros a la velocidad del robot, mientras el robot está de camino *y encendido*.
- El evento `arrive` se dispara cuando el cuentakilómetros alcanza la distancia. Pone al robot en `?to` y borra `moving`, lo que desactiva a la vez el proceso y el propio evento.

El final del desplazamiento es un evento, y no una acción, a propósito: un robot que ha recorrido la distancia *está* en la habitación siguiente, le guste o no al agente. La sección de alternativas lo compara con una acción `stop-move`.

La batería: un proceso y un evento

La batería necesita dos leyes (Listado 17.6) y dos acciones, `power-off` y `power-on`, que dan al agente el único control que tiene sobre la descarga. `(powered ?r)` es precondición de todas las acciones del robot, y `power-on` necesita que quede algo de carga.

```
71  ;; The battery drains whenever the robot is on
72  (:process drain
73    :parameters (?r - robot)
74    :precondition (powered ?r)
75    :effect (decrease (battery ?r) (* #t (drain-rate ?r))))
76
77  ;; An empty battery switches the robot off, wherever it is
78  (:event shutdown
79    :parameters (?r - robot)
80    :precondition (and (powered ?r) (<= (battery ?r) 0))
81    :effect (not (powered ?r))))
```

Listado 17.6: La batería se descarga mientras el robot está encendido, y una batería vacía apaga el robot (parte del dominio `courier-battery`).

El proceso `drain` no tiene nada que ver con moverse: su condición es solo `(powered ?r)`. El evento `shutdown` borra `(powered ?r)`, que es su propia condición, la condición de `drain` y la mitad de la condición de `travel`. Un robot que se queda sin batería entre dos habitaciones se queda allí: `moving` se cumple, `travel` no funciona, `arrive` no se disparará nunca y `power-on` no es aplicable con la batería vacía. El planificador no puede dejar que esto le ocurra a un robot que necesita, y en ningún sitio hemos escrito una precondición que diga «ten batería suficiente para el trayecto». Ese trabajo lo hacen las leyes del mundo.

Paso O: el estado de un modelo híbrido

La tabla de estado crece de dos maneras. Primero, hechos que solo existen para encender y apagar procesos: `(powered ?r)` y `(moving ?r ?from ?to)` no son cosas que la historia mencione como objetivos; son las condiciones de `drain` y `travel`. Segundo, números que miden un proceso: `(traveled ?r)` está ahí para que un evento pueda saber cuándo ha terminado el desplazamiento. Para cada proceso, pregunta: ¿qué hecho lo pone en marcha y lo detiene, y qué número dice cuánto ha avanzado? Si falta cualquiera de los dos en la tabla, el proceso no se puede escribir.

Ensayar el modelo

El problema típico es la entrega de los primeros capítulos: `r1` está en la base de carga, los medicamentos están en la farmacia y deben ir a

la sala A. La batería está a 30 y se descarga a 0,5 por segundo, así que el robot puede estar encendido 60 segundos. El trayecto es de 38 m a 1 m/s.

```
$ kairos courier-battery-domain.pddl courier-battery-p01.pddl
; cost = 38
; makespan = 38.000
0.000: (start-move r1 dock lobby) [0.000]
4.000: (start-move r1 lobby pharmacy) [0.000]
10.000: (pick r1 meds1 pharmacy) [0.000]
10.000: (start-move r1 pharmacy lobby) [0.000]
16.000: (start-move r1 lobby corridor) [0.000]
26.000: (start-move r1 corridor ward-a) [0.000]
38.000: (drop r1 meds1 ward-a) [0.000]
38.000: @PlanEND
```

El plan solo enumera los inicios de los desplazamientos. Las llegadas son eventos: las deducimos de las distancias. El robot sale de la base de carga en 0 y del vestíbulo en 4, así que llegó al vestíbulo en 4 (4 m a 1 m/s); vuelve a salir del vestíbulo en 16, seis segundos después de salir de la farmacia. Cada `start-move` ocurre justo en el instante de la llegada anterior. Al final, la batería está a $30 - 0{,}5 \times 38 = 11$. La Figura 17.5 muestra los diez primeros segundos en tres carriles: lo que hace el agente, lo que hacen los procesos y cuándo se disparan los eventos.

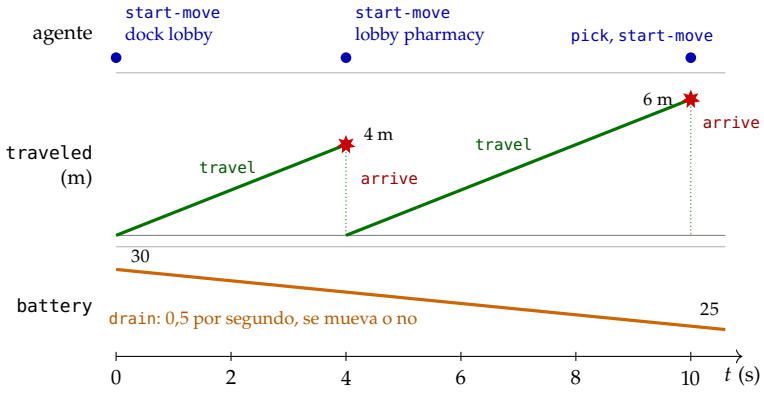


Figura 17.5.: Los diez primeros segundos del plan. Arriba: las decisiones del agente. En medio: el proceso `travel` hace subir el cuentakilómetros hasta que se dispara el evento `arrive`, y el siguiente `start-move` lo pone a cero. Abajo: al proceso `drain` no le importa nada de esto.

El problema `courier-battery-p02` es la misma entrega con la batería a 8: suficiente para 16 segundos. Kairos responde `no plan found`, después de probar 221 664 estados. Ese es el problema «sin plan» de la batería de pruebas. El cuarto problema de la batería, el que tienta al planificador a hacer trampa, merece su propia subsección.

La prueba de la trampa: un robot que avanza por inercia

Supongamos que hubiéramos escrito el proceso `travel` solo con la condición `(moving ?r ?from ?to)`, olvidando `(powered ?r)`. El modelo parece correcto y resuelve `p01` con el mismo plan. Ese error está en el dominio `courier-battery-coast`, y esto es lo que hace Kairos con él en el problema que no debería tener plan:

```
$ kairos courier-battery-coast-domain.pddl courier-battery-coast-p02.
    pddl
; cost = 38
; makespan = 38.000
```

```
0.000: (start-move r1 dock lobby) [0.000]
4.000: (start-move r1 lobby pharmacy) [0.000]
10.000: (pick r1 meds1 pharmacy) [0.000]
10.000: (start-move r1 pharmacy lobby) [0.000]
15.400: (power-off r1) [0.000]
16.000: (power-on r1) [0.000]
16.000: (start-move r1 lobby corridor) [0.000]
16.500: (power-off r1) [0.000]
26.000: (power-on r1) [0.000]
26.000: (start-move r1 corridor ward-a) [0.000]
26.000: (power-off r1) [0.000]
38.000: (power-on r1) [0.000]
38.000: (drop r1 meds1 ward-a) [0.000]
38.000: @PlanEND
```

El robot inicia un desplazamiento y se apaga. Apagado, su batería no se descarga, y el proceso equivocado lo sigue empujando por el pasillo. Se enciende un instante en cada habitación para iniciar el siguiente desplazamiento. La batería, a 8 al principio, está a 0,3 cuando el robot se apaga por primera vez, y eso le basta para el resto del día. El planificador ha encontrado una máquina de movimiento perpetuo, porque el modelo contenía una.

Con el dominio correcto, el mismo plan se rechaza en el primer paso que depende del viaje gratis: el robot se quedó a 60 cm del vestíbulo cuando se apagó.

```
$ kairos validate courier-battery-domain.pddl courier-battery-p02.
    pddl courier-battery-coast-p02.plan
courier-battery-coast-p02.plan:8: error: step 7 (start-move r1 lobby
     corridor) at 16.000: precondition false: (robot-at r1 lobby)
    is false [validate.precondition_failed]
 8 | 16.000: (start-move r1 lobby corridor) [0.000]
plan invalid
```

En un modelo híbrido, el planificador busca viajes gratis: un proceso que funciona cuando no debería, o uno que no funciona cuando debería. Prueba con un problema cuyos únicos planes necesitarían uno.

La lección es general. En los modelos clásicos, la trampa era una precondición que faltaba. En los modelos híbridos es una *condición de un proceso* que falta: cualquier tramo de tiempo en el que un proceso bueno funciona gratis, o un proceso malo está apagado, se encontrará y se usará. La prueba de la trampa para un modelo PDDL+ es un problema que es imposible en el mundo y que solo sería posible a través de un tramo así.

Una muestra que se calienta

La segunda ley de la naturaleza en el hospital trata de las muestras. Una muestra de sangre debe mantenerse fría. En una nevera está a salvo; fuera de la nevera se calienta, y por encima de un límite se estropea. El dominio `courier-cold` deja fuera la batería, para que sea pequeño, y conserva el mismo desplazamiento de tres piezas. El Listado 17.7 muestra lo nuevo.

```
35  ;; Picking an item takes it out of the fridge, if it was in one
36  (:action pick
37    :parameters (?r - robot ?i - item ?l - location)
38    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
39    :effect (and (not (item-at ?i ?l)) (not (free ?r))
40                 (carrying ?r ?i) (not (chilled ?i))))
41  ;; Dropping a sample into the fridge of the room
42  (:action store
43    :parameters (?r - robot ?i - sample ?l - location)
```

```
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i)
                       (has-fridge ?l))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l) (free ?r) (chilled ?i)))
  ;; Nobody does this: it happens to every sample out of a fridge
  (:process warm-up
    :parameters (?i - sample)
    :precondition (and (not (chilled ?i)) (not (spoiled ?i)))
    :effect (increase (temperature ?i) (* #t (warming-rate))))

  (:event spoil
    :parameters (?i - sample)
    :precondition (and (>= (temperature ?i) (max-temperature ?i))
                       (not (spoiled ?i)))
    :effect (spoiled ?i)))
```

Listado 17.7: La cadena de frío: recoger saca una muestra de la nevera, guardar la mete en una, y entre medias se calienta (parte del dominio `courier-cold`).

El proceso `warm-up` no tiene ningún hecho que una acción «ponga en marcha». Su condición trata de dónde está la muestra: funciona para toda muestra que no esté `(chilled ?i)`. Ninguna acción del robot menciona siquiera la temperatura. El agente controla el proceso solo a través de los hechos de su condición: `pick` saca una muestra del frío, y `store` (un `drop` en la nevera de una habitación que la tenga) la vuelve a meter.

En el problema `courier-cold-p01`, el robot está en el pasillo, con unos medicamentos a mano que deben ir a la sala A. Una muestra espera en una mesa del laboratorio a 4 grados, ya calentándose a 0,1 grados por segundo, y debe acabar en la nevera de la UCI. Se estropea a 8 grados: tiene 40 segundos fuera de una nevera. Tanto el laboratorio como la UCI tienen nevera.

Un planificador que cuenta acciones tiene la tentación de entregar primero los medicamentos, ya que están ahí mismo: diez acciones en total. Escribimos ese plan a mano y lo validamos.

```
$ kairos validate courier-cold-domain.pddl courier-cold-p01.pddl
     courier-cold-p01-meds-first.plan
courier-cold-p01-meds-first.plan: error: the goal does not hold at
     the end of the plan: (spoiled sample1) is true [validate.
     goal_not_reached]
plan invalid
```

Todas sus acciones son aplicables; falla por algo que no hizo ninguna acción. El robot llega al laboratorio a los 32 segundos y a la UCI a los 57, y la muestra se estropeó a los 40. Ahora, el plan de Kairos:

```
$ kairos courier-cold-domain.pddl courier-cold-p01.pddl
; cost = 73
; makespan = 73.000
0.000: (start-move r1 corridor lab) [0.000]
8.000: (pick r1 sample1 lab) [0.000]
8.000: (store r1 sample1 lab) [0.000]
8.000: (start-move r1 lab corridor) [0.000]
16.000: (pick r1 meds1 corridor) [0.000]
16.000: (start-move r1 corridor ward-a) [0.000]
28.000: (drop r1 meds1 ward-a) [0.000]
28.000: (start-move r1 ward-a corridor) [0.000]
40.000: (start-move r1 corridor lab) [0.000]
48.000: (pick r1 sample1 lab) [0.000]
48.000: (start-move r1 lab corridor) [0.000]
56.000: (start-move r1 corridor ward-b) [0.000]
68.000: (start-move r1 ward-b icu) [0.000]
73.000: (store r1 sample1 icu) [0.000]
73.000: @PlanEND
```

El planificador ha encontrado algo en lo que no habíamos pensado al escribir el problema. El robot va al laboratorio y, a los 8 segundos, recoge la muestra de la mesa y la guarda en la nevera *del laboratorio*. Eso detiene el proceso. Con la muestra a salvo a 4,8 grados, el robot entrega los medicamentos sin prisa, vuelve a los 48 segundos y solo entonces saca la muestra para los 25 segundos del trayecto a la UCI, adonde llega a 7,3 grados. Es lo que haría una enfermera sensata. El plan no es el más corto en tiempo: llevar la muestra directamente a la UCI y después entregar los medicamentos requiere 62 segundos y once acciones, y el validador también lo acepta (`courier-cold-p01-sample-first.plan`). La Figura 17.6 dibuja la temperatura en los tres planes.

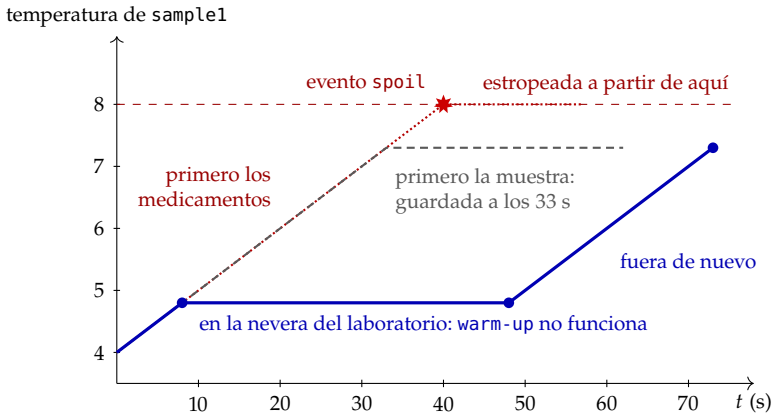


Figura 17.6.: La temperatura de la muestra en tres planes. De puntos: primero los medicamentos; la muestra se estropea en la mesa a los 40 s. Discontinua: primero la muestra, a mano. Gruesa: el plan de Kairos, que aparca la muestra en la nevera del laboratorio. La pendiente es siempre la misma, 0,1 grados por segundo: los planes solo se diferencian en *cuándo* funciona el proceso.

El problema `courier-cold-p02` es el que no tiene plan: solo la muestra, en un día de calor en el que aguanta 10 segundos en la mesa, con el robot a 8 segundos del laboratorio y el laboratorio a 33 segundos de la UCI. Kairos responde `no plan found`.

17.6. Cómo planifica Kairos con PDDL+

Un planificador clásico elige entre un conjunto finito de acciones. Un planificador PDDL+ debe elegir también *cuándo*, y el tiempo es un continuo. Todo planificador PDDL+ doma el continuo de alguna manera, y la manera determina qué planes puede encontrar. Esta sección describe lo que hace Kairos; los datos salen de su documentación Martín Rico et al., 2026 y de las ejecuciones de este capítulo.

Búsqueda en pasos de tiempo

Kairos planifica PDDL+ **en pasos de tiempo**. A las acciones del dominio les añade acciones propias que no hacen otra cosa que dejar pasar el tiempo:

- una *espera corta*, que deja pasar 0,1 segundos;
- una *espera larga*, que deja pasar el tiempo, en pasos de 0,1 segundos, hasta que alguna condición numérica está a punto de cambiar: se detiene en el último paso antes del cambio.

Cuando pasa el tiempo, los procesos activos se integran numéricamente con un método de Runge-Kutta, de modo que una cantidad cuyo ritmo está cambiando (la altura de la pelota) se calcula con precisión y no como

una escalera. Los eventos se disparan entre pasos: después de cada paso y de cada acción, si se cumple la condición de un evento, el evento se aplica, y no puede ocurrir nada más mientras algún evento pueda dispararse. Con estos añadidos, el problema se convierte en una búsqueda como las del Capítulo 3, sobre estados que ahora llevan números y a los que se llega mediante acciones y esperas. Las esperas no aparecen en el plan: son los huecos entre sus líneas.

Esto explica el aspecto de los planes que hemos visto. Todos sus instantes son múltiplos de 0,1. La válvula del depósito se cerró a los 4,0 segundos, el primer paso en el que se cumplía el objetivo, y no a los 4,5. El robot que avanzaba por inercia se apagó a los 15,4 segundos, el último paso antes de que se le agotara la batería. Y el primer plan del coche, con sus 33 segundos a 0,3 m/s, son dos esperas cortas, una espera larga y unas pocas acciones: pocos pasos de búsqueda, mucho tiempo.

Validación en tiempo continuo

Un plan encontrado por pasos es solo un candidato, porque el mundo del modelo no se mueve por pasos. Supongamos que el nivel de un depósito cruza su capacidad entre dos pasos y vuelve a bajar: la búsqueda por pasos nunca ve la inundación. Por eso Kairos valida cada candidato **en tiempo continuo** antes de imprimirlo. El validador integra los procesos de forma exacta y dispara cada evento en el instante en que se cruza su condición, caiga donde caiga ese instante. Un candidato al que llegan los pasos pero que el tiempo continuo rechaza se descarta, y la búsqueda sigue. Todo plan PDDL+ que imprime Kairos ha pasado esta comprobación, y `kairos validate` aplica la misma comprobación a cualquier plan, incluidos los que escribimos a mano, cuyos instantes no tienen por qué ser múltiplos de nada.

Kairos busca en pasos de 0,1 s y valida en tiempo continuo. Lo que imprime es correcto; lo que no puede imprimir es un plan cuya temporización sea más fina que los pasos.

Lo que se le puede escapar

El precio se paga por el otro lado. Un plan que necesita una decisión en un instante que no está en la rejilla de pasos no se encuentra. El problema `tank-p04` es `p01` con un caudal de entrada de 3 litros por segundo. El nivel alcanza 8 litros a los 2,667 s, entre los pasos de 2,6 y 2,7:

```
$ kairos tank-domain.pddl tank-p04.pddl
; cost = 2.7
; makespan = 2.700
0.000: (open-valve) [0.000]
2.700: (close-valve) [0.000]
2.700: @PlanEND
```

Aquí no hay daño: el objetivo pide *al menos* 8 litros, y a los 2,7 s el depósito tiene 8,1. El objetivo es un blanco ancho y algún paso cae en él (Figura 17.7). Si el objetivo hubiera sido un nivel entre 7,95 y 8,05 litros, ningún paso caería en él, y Kairos respondería `no plan found` aunque cerrar a los 2,667 s resuelve el problema. Por eso esa respuesta no se llama `unsolvable`, y por eso las notas que la acompañan dicen «evidence but not a proof». El Capítulo 18 vuelve sobre este límite y sobre cómo modelar para sortearlo.

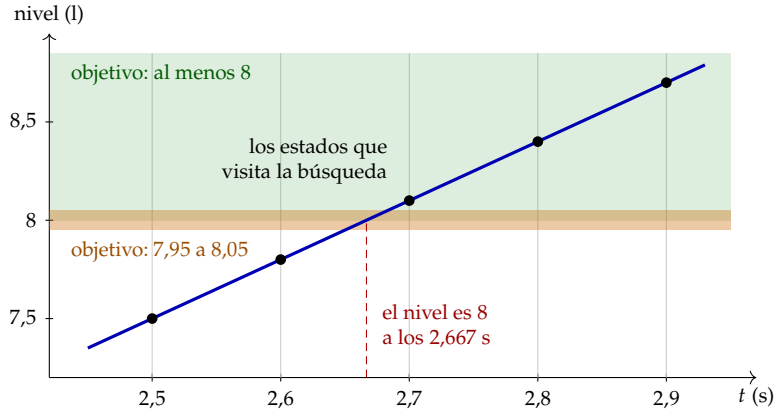


Figura 17.7.: El depósito con un caudal de entrada de 3 l/s, cerca del instante en que contiene 8 litros. La línea es el nivel en tiempo continuo; los puntos son los estados que visita la búsqueda por pasos. Un objetivo de «al menos 8» contiene puntos. La estrecha franja naranja contiene la línea durante un momento y ningún punto.

Otras formas de planificar con PDDL+

Los planificadores de PDDL+ se diferencian en cómo tratan el tiempo. Unos fijan un paso, como hace Kairos, y buscan sobre el modelo discretizado; ENHSP Scala et al., 2016 es el más conocido de ellos. Otros traducen el problema a ecuaciones sobre números reales y se las dan a un resolutor de restricciones, que encuentra instantes exactos pero restringe la clase de cambio que se puede escribir. Todos comparten la idea de comprobar el resultado frente a la semántica continua, que es también lo que hace el validador VAL con los planes de cualquier planificador Howey et al., 2004. Pocos planificadores aceptan PDDL+ en absoluto: de los ocho planificadores de la tabla de soporte del Apéndice B, solo Kairos planificó el dominio de prueba con procesos. Un modelo que necesita procesos y eventos es, por tanto, la clase de modelo menos portable de este libro.

17.7. El método MODEL: quién causa cada cambio

Hasta ahora, el paso D preguntaba qué se puede hacer. A partir de ahora pregunta qué se puede hacer *o puede ocurrir*, y la primera tarea del modelador es clasificar cada cambio de la historia según su causa. La Tabla 17.2 enumera las preguntas que añade este capítulo.

Paso	Preguntas nuevas
M	¿Cambia el mundo cuando el agente no hace nada? ¿Esperar cuesta algo, o consigue algo?
O	Para cada cantidad que cambia por sí sola: ¿qué hecho enciende y apaga el cambio, y qué número mide cuánto ha avanzado?
D	Para cada cambio: ¿quién lo causa? El agente, por una decisión (acción); el mundo, mientras se cumple una condición (proceso); el mundo, en el instante en que una condición se vuelve cierta (evento).
D	Para cada evento: ¿qué vuelve a hacer falsa su condición una vez que se ha disparado?
E	¿Hay algún tramo de tiempo en el que un proceso bueno funcione gratis, o uno malo esté apagado? Escribe el problema que lo necesita.

Tabla 17.2.: Lo que los procesos y los eventos añaden al método MODEL.

Una prueba rápida para la tercera pregunta: imagina que el agente se queda dormido. Lo que siga ocurriendo es un proceso o un evento. Lo que deje de ocurrir era una acción. El agua sigue entrando (proceso), el depósito se desborda (evento), la válvula se queda como estaba (así que

abrirla y cerrarla son acciones). Y si algo sigue ocurriendo solo durante un tiempo fijo después de que el agente lo iniciara, como un desplazamiento que termina por sí solo, las dos visiones son posibles, y ese es el tema de la comparativa que sigue.

Alternativas comparadas

¿Cómo debe modelarse el final de un desplazamiento? Lo hemos escrito como un evento. Hay otras dos codificaciones razonables, y las tres están en los ejemplos de este capítulo, cada una con el problema `p01`:

- **Evento** (`courier-battery`): `start-move`, proceso `travel`, evento `arrive`. El modelo de la Sección 17.5.
- **Acción** (`courier-battery-stop`): el evento se sustituye por una acción `stop-move` con la misma precondición y los mismos efectos. El agente decide cuándo el robot «está» en la habitación siguiente, en cualquier momento después de recorrer la distancia.
- **Acción durativa** (`courier-battery-durative`): las tres piezas vuelven a plegarse en el `move` durativo del Capítulo 10, con `(over all (powered ?r))`. La batería sigue siendo un proceso y `shutdown` sigue siendo un evento: las acciones durativas y los procesos pueden compartir un modelo, como explica el Capítulo 18.

Con la acción, Kairos devuelve un plan de 44 segundos en el que el robot llega a la farmacia a los 10 s y «se detiene» a los 16: durante seis segundos no está en ningún sitio, con la batería descargándose. Con las otras dos, el plan tarda 38 segundos. Los números de las filas de Tamaño y Guía son los que informa `kairos -v` para `p01`.

Tabla 17.3.: Tres codificaciones del final de un desplazamiento, sobre `courier-battery-p01`.

	Llegada como evento	**Llegada como acción**	**Desplazamiento durativo**
Fidelidad	Buena: el robot llega cuando llega. Un robot que se apaga por el camino se queda allí.	Permite planes en los que el robot se entretiene entre habitaciones; ningún robot real hace eso. No se pierde ninguna solución.	Buena. Un apagado durante un desplazamiento rompe la condición `over all`, así que esos planes son inválidos en lugar de quedarse atascados.
Tamaño	32 acciones instanciadas, 15 procesos, 15 eventos	46 acciones instanciadas, 15 procesos, 1 evento	18 acciones y 14 acciones durativas, 1 proceso, 1 evento
Guía	76 139 estados, 0,4 s	290 833 estados, 1,4 s: cada llegada es una decisión más, con muchos instantes posibles	279 estados, 0,01 s: un desplazamiento es una decisión y su final es un instante conocido
Claridad	Tres piezas por actividad; el plan solo muestra los inicios	El plan muestra inicios y paradas, como un registro	Una pieza por actividad; el plan muestra duraciones
Portabilidad	Solo PDDL+	Solo PDDL+	Planificadores PDDL+ que acepten también acciones durativas

Veredicto. Cuando una actividad termina por sí sola, no hagas de su final una decisión: la acción `stop-move` pierde en Fidelidad y en Guía y no gana nada. Usa una acción para terminar una actividad solo cuando parar sea de verdad una elección (cerrar una válvula, soltar un pedal). Entre el evento y la acción durativa, prefiere la acción durativa siempre que la actividad tenga una duración conocida al empezar y nada

pueda interrumpirla a medias y dejar que se reanude: la búsqueda es dos órdenes de magnitud más pequeña. Quédate con el evento cuando el final dependa de cómo evoluciona el mundo: un robot que va más despacio cuando va cargado, un depósito que se llena a un ritmo que cambian otros procesos, un desplazamiento que un apagado debe congelar y no cancelar.

17.8. Errores comunes

Un evento que no se desactiva a sí mismo

El evento `overflow` escrito solo con la precondición `(>= (level) (capacity))` parece inofensivo (`tank-wrong-loop-domain.pddl`). `kairos check` acepta el dominio: el error no está en la sintaxis. El síntoma aparece cuando un plan llega al evento. El planificador no encuentra plan para el problema cuyo objetivo es la inundación, y el validador dice por qué con un plan escrito a mano:

```
tank-wrong-loop-p02.plan: error: events keep firing at
5.000000 (overflow): an event must make its own condition
false [validate.event_loop]
```

Solución: todo evento necesita, en su precondición, algo que su propio efecto haga falso. Un hecho que añade y que niega (`(not (flooded))`), un hecho que borra (`(powered ?r)` en `shutdown`) o un número que devuelve al otro lado del umbral (la velocidad en `bounce`).

Un proceso sin #t, o #t fuera de un proceso

Un proceso cambia números a un ritmo, así que su efecto debe multiplicar por `#t`. Escribir `(increase (level) (inflow))` en el proceso `fill` se rechaza:

```
tank-wrong-jump-domain.pddl:29:13: error: in process 'fill':
processes can only have continuous effects (with #t)
[semantic.discrete_effect_in_process]
```

El error contrario es dar el efecto continuo a la acción que inicia el cambio, «porque abrir la válvula llena el depósito»:

```
tank-wrong-action-domain.pddl:19:39: error: in action
'open-valve': #t can only appear in increase/decrease effects
of durative actions and processes
[semantic.misplaced_continuous]
```

Solución: la acción añade un hecho; un proceso con ese hecho como condición se encarga de llenar. Un salto en un número en un instante corresponde a una acción o a un evento.

Un proceso que funciona cuando no debe

El robot que avanza por inercia mientras está apagado (Sección 17.5) es el error típico de un modelo híbrido: la condición de un proceso es demasiado débil. Nada rechaza el dominio, y los problemas corrientes dan planes corrientes. El síntoma es un plan para un problema que no debería tenerlo, o un plan que enciende y apaga cosas en momentos extraños. Solución: para cada proceso, enumera todo lo que debe ser cierto para que el cambio continúe, y escríbelo todo en la precondición. Después, haz la prueba de la trampa.

Leer «no plan found» como «unsolvable»

En un problema PDDL+ sin plan, Kairos termina con `kairos: no plan found (exhausted)` y código de salida 14, nunca con `unsolvable` y 10. Ha probado todas las temporizaciones de su rejilla de pasos, no todas las temporizaciones. Antes de concluir que un modelo es demasiado estricto, comprueba si el plan que esperabas necesita un instante entre dos pasos: escríbelo a mano y dáselo a `kairos validate`, que no tiene rejilla.

Olvidar `:time`, o usar las palabras clave de una acción durativa

Un dominio con procesos o eventos y sin `:time` en sus requisitos se acepta con un aviso por cada característica:

```
tank-no-time-domain.pddl:3:18: warning: uses processes but
:requirements does not include :time
[semantic.undeclared_requirement]
```

Otros planificadores pueden rechazar el fichero. Y los procesos y los eventos llevan `:precondition` y `:effect`, como las acciones; escribir `:condition`, como en una acción durativa, es un error (`unknown key :condition in process or event`).

17.9. Resumen

PDDL+ separa lo que decide el agente de lo que hace el mundo. El agente tiene acciones instantáneas. El mundo tiene procesos, que cambian números de forma continua mientras se cumple su condición, y eventos, que cambian el estado en el instante en que su condición se vuelve cierta. Un plan enumera las decisiones con sus instantes y dónde termina; el resto se sigue del modelo y hay que reproducirlo para leerlo.

- `(:process name :parameters (...) :precondition C :effect E)`: funciona mientras se cumple `C`; `E` solo tiene `increase` y `decrease` con `#t`. Requisito `:time`.
- `(:event name :parameters (...) :precondition C :effect E)`: se dispara cuando `C` se vuelve cierta; `E` es como el efecto de una acción y debe hacer falsa `C`.
- Las acciones controlan los procesos de forma indirecta, a través de los hechos de sus condiciones. El patrón inicio-proceso-parada da al agente una actividad que puede iniciar y terminar.
- Los resultados malos son eventos que dejan una marca que el objetivo rechaza; el planificador los evita impidiendo que su condición llegue a ser cierta.
- En el mundo del mensajero, un desplazamiento es una acción, un proceso y un evento; la batería se descarga mientras el robot está encendido y una batería vacía lo apaga; una muestra se calienta fuera de la nevera y se estropea.
- Kairos busca en pasos de 0,1 s, integra los procesos con Runge-Kutta, dispara los eventos entre pasos y valida cada plan en tiempo continuo. Sus planes son válidos; un plan que necesita una temporización más fina no se encuentra, y la respuesta es entonces `no plan found` (código de salida 14), no `unsolvable`.
- El método MODEL pregunta de cada cambio quién lo causa, y pone a prueba los modelos híbridos en busca de procesos que funcionan gratis.

17.10. Lecturas complementarias

La definición de PDDL+ es el artículo de Fox y Long Fox y Long, 2006: sus primeras secciones motivan los procesos y los eventos con los ejemplos que se usan aquí (el coche es suyo) y se pueden leer sin la parte formal; la semántica, por traducción a autómatas híbridos, está en la segunda mitad. Los efectos continuos de las acciones durativas, de los que los procesos son la generalización, se definen en el artículo de PDDL 2.1 Fox y Long, 2003. El libro de Haslum y otros Haslum et al., 2019 tiene un capítulo breve sobre PDDL+ con la misma notación. Para cómo busca un planificador en un problema PDDL+ discretizado, véase el artículo sobre ENHSP Scala et al., 2016; para cómo se validan los planes con cambio continuo, el artículo sobre VAL Howey et al., 2004. Lo que hace Kairos se describe en su documentación y en Martín Rico et al., 2026.

17.11. Ejercicios

Ejercicios de la Sección 17.2 — Procesos: cambio mientras se cumple una condición

Ejercicio 17.2.1. El depósito tiene un caudal de entrada de 4 litros por segundo y una capacidad de 10, y el objetivo pide al menos 9,5 litros, la válvula cerrada y sin inundación. Antes de ejecutar nada: ¿entre qué dos instantes hay que cerrar la válvula? ¿Qué instante mostrará el plan de Kairos? Escribe el problema y compruébalo.

Ejercicio 17.2.2. El depósito tiene una fuga: pierde medio litro por segundo siempre que hay agua dentro. Añade la fuga al dominio como un segundo proceso. ¿Cuánto se tarda ahora en llegar a 8 litros con un caudal de entrada de 2? Ejecútalo y explica el instante del plan. ¿Qué le ocurre al nivel después de cerrar la válvula?

Ejercicios de la Sección 17.3 — Eventos: cambio en un instante

Ejercicio 17.3.1. En `tank-domain.pddl`, un estudiante quita del evento el término `(not (flooded))` y quita el objetivo `(not (flooded))` del problema `p01`, «porque el evento no hace falta para ese problema». ¿Sigue encontrando Kairos el plan de la Sección 17.2? ¿Qué ocurre con el plan que cierra la válvula a los 6 segundos? Explica las dos respuestas.

Ejercicio 17.3.2. Añade al depósito un interruptor de boya: un evento que cierra la válvula por sí solo cuando el nivel alcanza `(switch-level)`. Con el interruptor a 9 litros, escribe un problema cuyo plan tenga una sola acción, y un problema que muestre que el depósito ya no puede inundarse.

Ejercicios de la Sección 17.4 — Tres clases de cambio

Ejercicio 17.4.1. Toma el plan de `ball-p01` y mueve el `catch` a los 3,0 segundos. Calcula a mano la altura y la velocidad de la pelota en ese instante, predice el mensaje del validador y compruébalo.

Ejercicio 17.4.2. Cambia el objetivo de `ball-p01` por coger la pelota después de exactamente un rebote: `(= (bounces) 1)`. Ejecútalo. ¿La

respuesta es un límite del planificador o un hecho sobre la pelota? Justifícalo con números.

Ejercicio 17.4.3. Clasifica cada una de estas cosas como acción, proceso o evento, para un modelo de un ascensor que usa una persona: la cabina sube; la puerta se abre cuando la cabina llega a una planta; la persona pulsa un botón; la persona entra; la puerta se cierra cinco segundos después de abrirse; la cabina está sobrecargada.

Ejercicio 17.4.4. El problema `car-p01` con un límite de velocidad de 1 m/s. Diseña a mano el plan más rápido, calcula su duración y valídalo. Después ejecuta Kairos y compara. ¿Qué dice la comparación sobre la nota que Kairos imprime con cada plan PDDL+?

Ejercicio 17.4.5. El coche consume combustible: 1 ml por segundo mientras el motor está en marcha y 2 ml por segundo más mientras la aceleración es positiva. Cuando se acaba el combustible el motor se cala, y un coche con el motor calado no se puede aparcar. Amplía el dominio. Encuentra un plan con 15 ml, y una cantidad de combustible para la que no haya plan.

Ejercicios de la Sección 17.5 — El mundo del mensajero en PDDL+

Ejercicio 17.5.1. En `courier-battery-p01`, ¿cuál es el mayor ritmo de descarga con el que la entrega sigue siendo posible? ¿Qué marca la batería cuando el robot recoge los medicamentos? Responde a partir del plan, sin ejecutar nada.

Ejercicio 17.5.2. Escribe un problema para `courier-battery` en el que el robot empiece apagado en la farmacia, junto a los medicamentos, con la batería a 15. Predice el plan y su primera acción.

Ejercicio 17.5.3. Supón que `pick` no tuviera `(powered ?r)` en su precondición. Describe un plan que el modelo aceptaría entonces y el mundo no. ¿Es una precondición que falta o un proceso que funciona gratis?

Ejercicio 17.5.4. Para `courier-cold-p01` hay tres planes en el texto. Ordénalos por número de acciones y por duración. ¿Cuál preferiría un planificador que minimiza acciones, y por qué no lo devolvió Kairos? ¿Cuál querrías que ejecutara el robot, y qué tendría que haber en el modelo para que el planificador lo prefiriera?

Ejercicios de la Sección 17.6 — Cómo planifica Kairos con PDDL+

Ejercicio 17.6.1. Kairos responde `unsolvable` (código de salida 10) para un problema clásico sin plan y `no plan found` (código de salida 14) para `tank-p03`. Explica la diferencia a un compañero que cree que la segunda respuesta es un fallo.

Ejercicio 17.6.2. Escribe a mano un plan para `tank-p01` que cierre la válvula a los 4,05 segundos. ¿Es válido? ¿Podría haberlo devuelto Kairos? ¿Importa?

Ejercicios de la Sección 17.7 — El método MODEL: quién causa cada cambio

Ejercicio 17.7.1. Una cafetera: cuando se enciende, una resistencia calienta el agua a 5 grados por segundo; un termostato corta la resistencia a 95 grados; el café se puede preparar a partir de 90 grados. Escribe

la tabla de acciones, marcando la causa de cada cambio, y después el dominio. ¿Cuánto tarda un café partiendo de agua a 20 grados?

Ejercicio 17.7.2. Aplica los cinco criterios a esta elección en el modelo de la cadena de frío: el calentamiento de la muestra como un proceso (como en el texto) o como un efecto de los desplazamientos (cada `start-move` con una muestra en la mano suma los grados del trayecto). Da una historia en la que la segunda codificación acepte un plan imposible.

18. Modelar sistemas híbridos

El Capítulo 17 nos dio las dos construcciones de PDDL+ y tres mundos pequeños en los que cada ley podía leerse por separado: un proceso llena el depósito, un evento lo desborda. Los sistemas reales no son tan ordenados. En un invernadero, el calefactor calienta el aire mientras el frío de fuera lo enfría, y las plantas solo crecen cuando el resultado de esa pugna se mantiene en una franja estrecha. Un termostato reacciona a la temperatura, y su reacción cambia la temperatura a la que reacciona. Un robot que espera a que abra la farmacia está gastando batería, y que aguante la espera depende de un reloj que ninguna acción controla. Y algunas reglas no hablan de ningún momento concreto: el coche no debe superar *nunca* el límite, las plantas no deben helarse *nunca*. Con las construcciones del capítulo anterior se puede escribir cada una de estas cosas, pero escribirlas bien exige algo más que conocer la sintaxis.

Este capítulo trata de modelar con procesos y eventos. Añade una construcción, la restricción global, para las reglas que deben cumplirse en todo momento. Estudia el cambio que no es lineal, en el que el ritmo de una cantidad depende de otras cantidades o de ella misma. Estudia qué ocurre cuando varios procesos tocan el mismo número y cuando unos eventos disparan otros, incluidas las maneras en que un conjunto de eventos puede dispararse para siempre, y cómo evitarlo. Junta los procesos con las acciones durativas y los literales temporizados de la Parte IV. Después toma un sistema, el mensajero que debe recargar, y lo modela en tres niveles: solo números, acciones durativas con efectos continuos y PDDL+. La comparativa es el resultado principal del capítulo: la mayoría de los problemas no necesitan PDDL+, y saber cuándo lo necesitan es una habilidad de modelado.

El capítulo sigue ese orden. Tras los tres modelos vienen los límites de lo que puede encontrar un planificador que avanza en pasos de tiempo, y lo que el modelador puede hacer al respecto. La sección del método MODEL reúne un procedimiento para probar modelos híbridos y una guía para decidir entre los tres niveles. Como siempre, los errores y los ejercicios cierran el capítulo.

18.1 Reglas que se cumplen en todo momento: restricciones globales . 408
18.2 Cambio que no es lineal 410
Un coche con resistencia del aire 411
18.3 Procesos que interactúan 412
Un dron que aterriza . . 414
18.4 Eventos en cascada y eventos que no paran . 416
Eventos como dispositivos: el termostato 417
Eventos que no paran . 418
18.5 Procesos con acciones durativas y literales temporizados 420
Esperar no sale gratis . 421
18.6 Un sistema, tres modelos 423
Lo que cada nivel no puede decir 424
18.7 Límites: pasos y tolerancias 425
Un evento no rescata a la búsqueda 427
Valores exactos 428
18.8 El método MODEL: probar y elegir modelos híbridos 428
Alternativas comparadas 429
18.9 Errores comunes 430
18.10 Resumen 431
18.11 Lecturas complementarias 431
18.12 Ejercicios 432

18.1. Reglas que se cumplen en todo momento: restricciones globales

En el coche del Capítulo 17, el límite de velocidad se imponía en dos pasos. Un evento, `radar`, añadía el hecho `(fined)` cuando la velocidad pasaba del límite, y el objetivo del problema pedía `(not (fined))`. Funciona, y tiene dos debilidades. La regla vive a medias en el dominio y a medias en cada problema: un problema cuyo autor olvide el objetivo no tiene límite de velocidad. Y la búsqueda puede vagar por estados en los que el coche ya ha sido multado, desde los que no se puede completar ningún plan, sin saber que son callejones sin salida.

Una **restricción global** enuncia la regla directamente: una condición que debe ser cierta en cada instante de cada plan. El Listado 18.1 es de nuevo el coche, con una restricción en lugar del evento.

```
;; The car of chapter 17 with the speed limit as a global
;; constraint: a condition that must hold at every instant.
(define (domain car-limit)
  (:requirements :negative-preconditions :numeric-fluents :time)

  (:predicates
    (running)                ; the engine is on
    (parked))                ; stopped, engine off

  (:functions
    (distance-run)           ; meters from the start
    (velocity)               ; meters per second
    (acceleration)           ; meters per second squared
    (speed-limit))           ; meters per second

  ;; No plan may ever break this
  (:constraint keep-the-limit
    :parameters ()
    :condition (<= (velocity) (speed-limit)))

  (:action accelerate
    :parameters ()
    :precondition (and (running) (< (acceleration) 1))
    :effect (increase (acceleration) 1))

  (:action brake
    :parameters ()
    :precondition (and (running) (> (acceleration) -1))
    :effect (decrease (acceleration) 1))

  (:action park
    :parameters ()
    :precondition (and (running) (= (acceleration) 0)
                       (>= (velocity) -0.05) (<= (velocity) 0.05))
    :effect (and (not (running)) (parked)
                 (assign (velocity) 0)))

  (:process moving
    :parameters ()
    :precondition (running)
    :effect (and
      (increase (distance-run) (* #t (velocity)))
      (increase (velocity) (* #t (acceleration))))))
```

```
  (:constraint keep-the-limit
    :parameters ()
    :condition (<= (velocity) (speed-limit)))
```

Listado 18.1: El coche con el límite de velocidad como restricción global (`examples/ch18/car-limit-domain.pddl`).

Una restricción tiene un nombre, parámetros y una `:condition`. No tiene efecto: no hace nada, solo prohíbe. Con parámetros, hay una restricción por cada manera de instanciarlos (una por robot, una por depósito). Un plan solo es válido si todas las restricciones se cumplen en el estado inicial, después de cada acción y cada evento, y durante todo el tiempo en que los procesos actúan entre medias.

Una restricción global es una condición que debe cumplir cada instante de cada plan. Sustituye a la pareja «evento que marca el fallo, objetivo que rechaza la marca».

La construcción `:constraint` no está en PDDL+ tal como lo definieron Fox y Long. Viene del planificador ENHSP Scala et al., 2016 y es habitual en los *benchmarks* de PDDL+ escritos desde entonces. Kairos la lee como ENHSP y no pide ningún requisito propio. [1] `kairos check` la incluye entre las características del fichero:

```
$ kairos check car-limit-domain.pddl car-limit-p01.pddl
valid, plannable
  numeric-fluents: full
  continuous-effects: full (linear ones as discrete changes in
```

1: No la confundas con la sección `(:constraints ...)` de PDDL 3 (Capítulo 15). Aquella contiene fórmulas como `(always ...)` sobre los estados que recorre un plan; esta es una declaración con nombre, en singular, que se comprueba en tiempo continuo.

```
    engines M2 and M3, any in steps of time (integrated with Runge-
    Kutta))
  processes: full (planned in steps of time (to where a condition
    changes), integrated with Runge-Kutta; every plan validated in
    continuous time)
  global-constraints: full
```

El problema `car-limit-p01` es `car-p01` sin la conjunción sobre la multa: el objetivo es solo estar aparcado a diez metros o más. Kairos devuelve los mismos dos planes que en el Capítulo 17, el de 33,8 segundos y después el de 7,5. La diferencia aparece con un plan que incumple la regla. El conductor con prisa mantiene el pedal pisado durante 3,2 segundos:

```
$ kairos validate car-limit-domain.pddl car-limit-p01.pddl car-limit-
    p01-fast.plan
car-limit-p01-fast.plan: error: constraint 'keep-the-limit' broken
    at 2.000001: (<= (velocity) (speed-limit)): 2 <= 2 is false [
    validate.global_constraint]
plan invalid
```

Con el evento, el validador decía que el objetivo no se cumplía al final. Ahora dice qué regla se ha incumplido y cuándo: justo después de los 2 segundos, el instante en que la velocidad pasa de 2 m/s (los valores se imprimen redondeados, de ahí el extraño `2 <= 2 is false`). El diagnóstico es mejor, y no depende del objetivo.

¿Cuándo sigue siendo el evento la herramienta adecuada? Cuando lo malo no está prohibido, sino que tiene consecuencias. Una multa que cuesta dinero y se puede pagar, una muestra que se estropea y hay que reponer, una batería que se agota y deja al robot a la espera de que otro lo rescate: son estados del mundo por los que los planes pueden pasar, y necesitan un evento que cambie el estado. Una restricción es para lo que no debe ocurrir en absoluto. La tabla comparativa de las dos codificaciones de esta sección es el Ejercicio 18.1.1.

18.2. Cambio que no es lineal

El efecto de un proceso tiene la forma `(* #t` *ritmo*`)`. De lo que sea el ritmo depende la forma del cambio.

- El ritmo es una **constante**, o un fluente que nada cambia mientras el proceso actúa: `(inflow)`, `(drain-rate ?r)`. La cantidad es una recta en el tiempo. Es el **cambio lineal**, el único del Capítulo 12.
- El ritmo es **otra cantidad que también está cambiando**: la distancia del coche crece al ritmo `(velocity)` mientras la velocidad crece al ritmo `(acceleration)`. La distancia es una parábola. Cada eslabón de una cadena así sube el grado en uno.
- El ritmo depende de **la propia cantidad**: un cuerpo caliente se enfría más deprisa cuanto más caliente está; la resistencia del aire crece con la velocidad que reduce. La cantidad sigue una curva que no es un polinomio en absoluto: una exponencial que se acerca a un límite sin alcanzarlo.

PDDL+ escribe las tres de la misma manera. La segunda y la tercera son **cambio no lineal**, y importan por dos razones: muchos planificadores que aceptan efectos continuos solo aceptan la primera clase, y el instante en

que una cantidad no lineal cruza un umbral no se encuentra dividiendo una distancia entre una velocidad.

Un coche con resistencia del aire

El Listado 18.2 añade al coche la resistencia del aire. Un segundo proceso frena el coche en proporción al cuadrado de su velocidad, el modelo habitual de la resistencia aerodinámica.

```
(:process moving
  :parameters ()
  :precondition (running)
  :effect (and
    (increase (distance-run) (* #t (velocity)))
    (increase (velocity) (* #t (acceleration)))))

;; A second process on the same quantity: the two rates add up
(:process air-drag
  :parameters ()
  :precondition (and (running) (> (velocity) 0))
  :effect (decrease (velocity)
            (* #t (* (drag) (* (velocity) (velocity))))))

;; A braking car stops at zero: it does not start to go back
(:event halt
  :parameters ()
  :precondition (and (running) (<= (velocity) 0)
                     (< (acceleration) 0))
  :effect (and (assign (velocity) 0)
               (assign (acceleration) 0))))
```

Listado 18.2: Los procesos y el evento del coche con resistencia del aire (parte del dominio `car-drag`).

Dos procesos cambian la velocidad, y sus ritmos se suman: con el pedal pisado, la velocidad crece a $a - kv^2$, donde a es la aceleración y k el coeficiente de resistencia. A medida que el coche gana velocidad, el segundo término crece, hasta anular el primero. En esa *velocidad terminal*, $\sqrt{a/k}$, el coche deja de acelerar. Con $a = 1$ y $k = 0{,}25$ es de 2 m/s: este coche no necesita límite de velocidad, porque la física le pone uno.

El evento `halt` corrige una debilidad del primer coche. Allí, `park` necesitaba la velocidad a menos de 5 cm/s de cero, una ventana en la que el plan tenía que acertar mientras frenaba. Aquí, cuando el coche que frena llega a cero, el evento lo detiene: velocidad y aceleración pasan a ser exactamente cero, y `park` puede pedir exactamente cero y aplicarse en cualquier momento posterior. Un evento que se dispara en un cruce convierte una ventana estrecha en un estado estable. Volveremos a necesitar esta idea en la Sección 18.7.

```
$ kairos car-drag-domain.pddl car-drag-p01.pddl
; cost = 8
; makespan = 8.000
0.000: (accelerate) [0.000]
6.400: (brake) [0.000]
6.400: (brake) [0.000]
8.000: (park) [0.000]
8.000: @PlanEND
```

El conductor acelera durante 6,4 segundos, después frena dos veces en el mismo instante (el primer `brake` suelta el pedal, el segundo frena) y aparca a los 8 segundos. ¿Por qué 6,4? La Figura 18.1 reproduce el plan. La velocidad sube deprisa al principio y luego se curva hacia los 2 m/s sin llegar a tocarlos. A los 6,4 segundos el coche ha recorrido 10,03 m:

es el primer paso en que se alcanza la distancia objetivo. Frenar desde 1,99 m/s lleva 1,57 s, con ayuda de la resistencia, y el evento `halt` se dispara a los 7,97 s, que no es múltiplo de nada. El plan aparca en el paso siguiente.

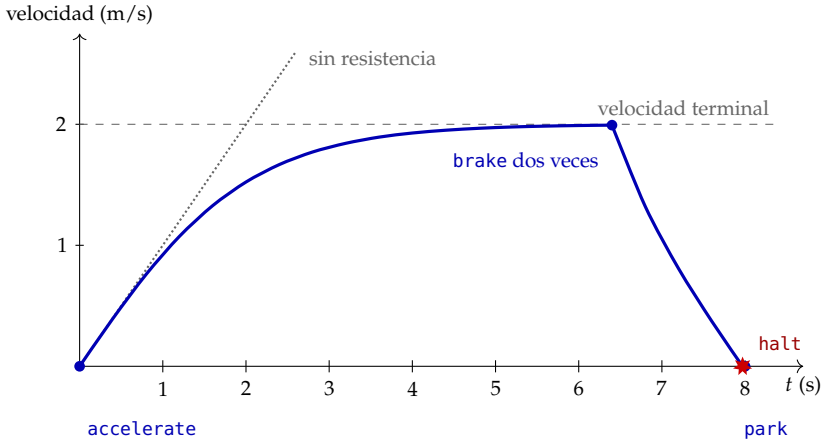


Figura 18.1.: La velocidad del coche con resistencia del aire en el plan de Kairos. Con un ritmo constante, la velocidad seguiría la línea de puntos; con resistencia, se curva hacia la velocidad terminal. La estrella es el evento `halt`, a los 7,97 s.

Por qué no basta una escalera

La manera más sencilla de calcular por pasos una cantidad que cambia es suponer que su ritmo se mantiene fijo durante cada paso. Para el cambio lineal, esto es exacto. Para cualquier otro, se equivoca en una cantidad que se va acumulando. Tomemos la pelota del Capítulo 17, soltada desde 5 m: calcular la altura en pasos de 0,1 s, cada uno con la velocidad del inicio del paso, da 0,5 m al cabo de un segundo, cuando en realidad la pelota está en el suelo. Un planificador que trabajara así situaría el bote un paso tarde, y todos los instantes posteriores heredarían el error. Kairos integra los procesos activos con un método de Runge-Kutta, que evalúa los ritmos en varios puntos dentro de cada paso. Su tabla de características lo resume: el cambio numérico no lineal se «planifica y valida de forma exacta; las heurísticas linealizan las condiciones no lineales en cada estado». La última parte significa que las estimaciones que *guían* la búsqueda son más toscas que el cálculo que *comprueba* cada estado, lo que cuesta esfuerzo de búsqueda, pero nunca validez.

En cuanto a la Portabilidad, el cambio no lineal vuelve a estrechar el campo. Los planificadores que aceptan los efectos continuos del Capítulo 12 suelen esperar ritmos constantes. Un modelo cuyos ritmos dependen de cantidades que cambian es, en la práctica, un modelo para un planificador de PDDL+.

18.3. Procesos que interactúan

Dos procesos interactúan cuando uno cambia un número que el otro lee o cambia. La resistencia y el motor del coche eran un primer caso. El invernadero del Listado 18.3 tiene las tres maneras en que se encuentran los procesos.

```
;; A greenhouse in winter. Three processes share the temperature:
;; the heater raises it, the cold outside lowers it, and the
;; plants grow only while it is in their comfort band.
(define (domain greenhouse)
```

```
(:requirements :negative-preconditions :numeric-fluents :time)

(:predicates
  (heater-on)
  (harvested))

(:functions
  (temperature)             ; degrees inside
  (outside)                 ; degrees outside
  (heater-power)            ; degrees per second it adds
  (loss)                    ; share of the difference lost per s
  (growth))                 ; seconds of good growing so far

;; The plants die of frost below 10 degrees and burn above 30
(:constraint plants-alive
  :parameters ()
  :condition (and (>= (temperature) 10) (<= (temperature) 30)))

(:action switch-on
  :parameters ()
  :precondition (not (heater-on))
  :effect (heater-on))

(:action switch-off
  :parameters ()
  :precondition (heater-on)
  :effect (not (heater-on)))

(:action harvest
  :parameters ()
  :precondition (>= (growth) 20)
  :effect (harvested))

(:process heating
  :parameters ()
  :precondition (heater-on)
  :effect (increase (temperature) (* #t (heater-power))))

;; Always active: the warmer inside, the faster the heat leaves
(:process losing-heat
  :parameters ()
  :precondition (> (temperature) (outside))
  :effect (decrease (temperature)
            (* #t (* (loss) (- (temperature) (outside))))))

(:process growing
  :parameters ()
  :precondition (and (>= (temperature) 22)
                     (<= (temperature) 28))
  :effect (increase (growth) (* #t 1))))
```

Listado 18.3: Un invernadero en invierno (dominio `greenhouse`).

- **Dos procesos cambian la misma cantidad.** `heating` sube la temperatura a un ritmo constante de 3 grados por segundo mientras el calefactor está encendido. `losing-heat` la baja todo el tiempo, una décima parte por segundo de la diferencia con el exterior. Sus ritmos se suman. Ninguno de los dos procesos sabe del otro.
- **Un proceso cambia aquello de lo que depende su propio ritmo.** La pérdida es proporcional a la temperatura que reduce: la tercera clase de cambio de la sección anterior.
- **Un proceso se activa por una cantidad que cambian otros.** `growing` no tiene ningún hecho en su condición, solo números: actúa mientras la temperatura está entre 22 y 28 grados. El agente no tiene ninguna acción que ponga en marcha el crecimiento. Solo puede llevar la temperatura a la franja y mantenerla allí.

Una restricción global mantiene vivas las plantas: nunca por debajo de 10 grados, nunca por encima de 30. Antes de ejecutar el planificador,

el paso E de un modelo híbrido empieza con un lápiz. ¿Adónde va la temperatura si el calefactor se queda encendido? Sube hasta que la pérdida iguala al calentamiento: $0{,}1\,(T - 5) = 3$, es decir, $T = 35$ grados, por encima de lo que aguantan las plantas. ¿Y si se queda apagado? Baja hacia los 5 grados del exterior, por debajo de lo que aguantan. Ninguna de las dos posiciones del interruptor puede dejarse tal cual. El plan tiene que encender y apagar, y el planificador tiene que encontrar cuándo.

```
$ kairos greenhouse-domain.pddl greenhouse-p01.pddl
; cost = 24.9
; makespan = 24.900
0.100: (switch-on) [0.000]
10.700: (switch-off) [0.000]
13.800: (switch-on) [0.000]
20.100: (switch-off) [0.000]
23.200: (switch-on) [0.000]
24.900: (harvest) [0.000]
24.900: @PlanEND
```

La Figura 18.2 lo reproduce. El calefactor se enciende al principio (un paso tarde, lo que cuesta una décima de grado y nada más). La temperatura entra en la franja de crecimiento 4,5 segundos después y sigue subiendo. A los 10,7 segundos está en lo alto de la franja, y el calefactor se apaga; la temperatura baja, más deprisa de lo que subió, y a los 13,8 segundos, en lo bajo de la franja, el calefactor vuelve a encenderse. El ciclo se repite hasta que las plantas han tenido sus 20 segundos de crecimiento. Nadie le dijo al planificador que mantuviera la temperatura entre 22 y 28: el objetivo pide crecimiento, el crecimiento solo ocurre en la franja, y la búsqueda encontró los instantes de conmutación que la mantienen allí. Los ingenieros de control llaman a este tipo de control, todo o nada, *bang-bang*.

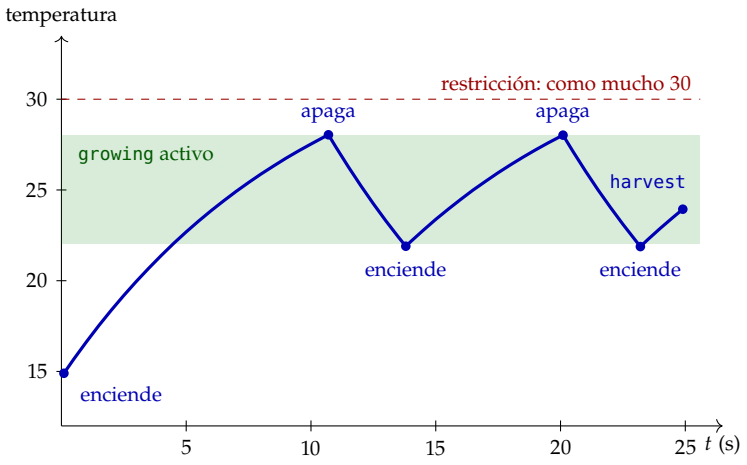


Figura 18.2.: La temperatura del invernadero en el plan de Kairos. Cada punto es una acción. Con el calefactor encendido, la curva se dobla hacia los 35 grados; con el calefactor apagado, hacia los 5: las subidas son más lentas que las bajadas. Las plantas crecen mientras la curva está en la franja.

Un dron que aterriza

El segundo ejemplo de procesos que interactúan no tiene una franja en la que mantenerse, sino un blanco en el que acertar: el suelo, despacio. Un dron se mantiene en el aire a 20 metros y debe aterrizar. El Listado 18.4 muestra sus leyes.

```
22  ;; Flight rules, at every instant
23  (:constraint below-ceiling
24    :parameters ()
```

```
    :condition (<= (altitude) (ceiling)))
  (:process flying
    :parameters ()
    :precondition (airborne)
    :effect (and (increase (altitude) (* #t (velocity)))
                 (decrease (velocity) (* #t (gravity)))))

  (:process pushing
    :parameters ()
    :precondition (and (airborne) (rotors-on))
    :effect (and (increase (velocity) (* #t (thrust)))
                 (decrease (battery) (* #t (rotor-rate)))))

  (:event battery-empty
    :parameters ()
    :precondition (and (rotors-on) (<= (battery) 0))
    :effect (not (rotors-on)))

  ;; Reaching the ground ends the flight, well or badly. The two
  ;; conditions exclude each other: only one event can fire
  (:event touch-down
    :parameters ()
    :precondition (and (airborne) (<= (altitude) 0)
                       (>= (velocity) (- 0 (max-impact))))
    :effect (and (not (airborne)) (not (rotors-on)) (landed)
                 (assign (velocity) 0)))

  (:event crash
    :parameters ()
    :precondition (and (airborne) (<= (altitude) 0)
                       (< (velocity) (- 0 (max-impact))))
    :effect (and (not (airborne)) (not (rotors-on)) (crashed)
                 (assign (velocity) 0))))
```

Listado 18.4: Las leyes del dron (parte del dominio `drone`).

La gravedad actúa todo el tiempo sobre la velocidad; los rotores, mientras están encendidos, empujan en sentido contrario y gastan batería. El mismo proceso, `pushing`, cambia dos cantidades de naturaleza distinta: así dice el modelo que el empuje cuesta energía. Una restricción mantiene al dron por debajo de un techo, y tres eventos vigilan umbrales.

Los dos últimos eventos merecen atención. Llegar al suelo termina el vuelo de una de dos maneras, y el modelo tiene un evento para cada una. Sus condiciones comparten `(<= (altitude) 0)` y se diferencian en la última conjunción: como mucho `(max-impact)` para `touch-down`, más para `crash`. Las dos condiciones *se excluyen mutuamente*. Es una regla que conviene conservar: cuando dos eventos pueden hacerse ciertos en el mismo instante y ambos cambian los mismos hechos, haz que sus condiciones sean excluyentes, para que el modelo nunca tenga que decir cuál va primero.

```
$ kairos drone-domain.pddl drone-p01.pddl
; cost = 3.3
; makespan = 3.300
1.100: (rotors-up) [0.000]
2.000: (rotors-down) [0.000]
2.100: (rotors-up) [0.000]
3.300: @PlanEND
```

El dron cae libremente durante 1,1 segundos y alcanza 11 m/s. Los rotores, que dan 15 m/s^2 frente a los 10 de la gravedad, lo frenan entonces 5 m/s cada segundo. Tras una pausa de una décima de segundo, siguen encendidos hasta el suelo, que el dron toca a los 3,28 segundos a 1,58 m/s, por debajo de los 2 m/s permitidos (Figura 18.3). Ahí se dispara el evento `touch-down`, y el plan termina en el paso siguiente. Los rotores

funcionaron durante 2,08 segundos y gastaron 10,4 de las 30 unidades de batería. Fíjate en la pausa a los 2,0 s: la búsqueda avanza en pasos de 0,1 s, y un único encendido que empezara en un paso tocaría el suelo demasiado deprisa o detendría el dron en el aire. Una décima de segundo de caída libre es la manera en que la búsqueda ajustó la velocidad de aterrizaje con la única resolución que tiene.

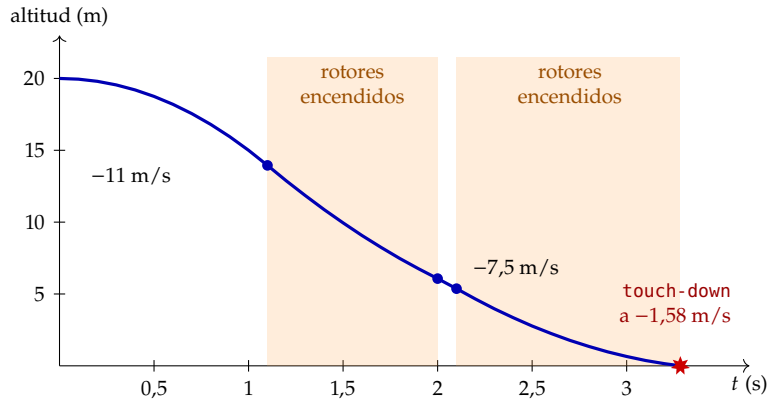


Figura 18.3.: El descenso del dron en el plan de Kairos. En los tramos blancos solo actúa la gravedad y la curva se dobla hacia abajo; en los sombreados, los rotores están encendidos y se dobla hacia arriba. Los puntos son las acciones.

18.4. Eventos en cascada y eventos que no paran

Cuando un evento se dispara, sus efectos pueden hacer cierta la condición de otro evento. Ese otro se dispara enseguida, en el mismo instante, y puede disparar un tercero. Una cadena de eventos en un mismo instante es una **cascada**. El ejemplo más limpio no tiene números (Listado 18.5).

```
;; A row of dominoes: the agent pushes one, events do the rest.
(define (domain dominoes)
  (:requirements :typing :negative-preconditions :time)

  (:types domino)

  (:predicates
    (next ?a ?b - domino)     ; ?b stands right after ?a
    (within-reach ?d - domino) ; the hand gets to it
    (fallen ?d - domino))

  (:action push
    :parameters (?d - domino)
    :precondition (and (within-reach ?d) (not (fallen ?d)))
    :effect (fallen ?d))

  ;; A fallen domino knocks down the next one. The condition
  ;; (not (fallen ?b)) is what stops the event: it fires once
  (:event topple
    :parameters (?a ?b - domino)
    :precondition (and (fallen ?a) (next ?a ?b)
                       (not (fallen ?b)))
    :effect (fallen ?b)))
```

```
$ kairos dominoes-domain.pddl dominoes-p01.pddl
; cost = 0
; makespan = 0.000
0.000: (push d1) [0.000]
0.000: @PlanEND
```

Listado 18.5: Fichas de dominó: un evento por cada par de vecinas (`examples/ch18/dominoes-domain.pddl`).

El agente empuja la primera ficha, la única a su alcance. El objetivo es la quinta, y el plan termina en el instante cero: cuatro eventos se dispararon uno tras otro sin que pasara el tiempo. La cascada se detiene porque cada evento se desactiva a sí mismo (su ficha ya ha caído) y la fila tiene un final.

Una cascada es también algo que el agente no puede interrumpir. El problema `dominoes-p02` pide la segunda ficha caída y la tercera en pie, y no tiene plan: entre el evento que derriba `d2` y el que derriba `d3` no hay ningún instante en el que colocar una acción. Si la historia necesita que el agente pueda intervenir, tiene que pasar tiempo entre los eventos: un proceso y un umbral, como en el Ejercicio 18.4.2.

Los eventos de un mismo instante se disparan en una cascada que ninguna acción puede interrumpir. Una cascada debe terminar: cada evento debe desactivarse a sí mismo, y ningún evento puede volver a activar otro.

Eventos como dispositivos: el termostato

En el Capítulo 17, los eventos eran accidentes: una inundación, un apagado, una muestra estropeada. Los eventos son también la manera en que PDDL+ modela los dispositivos automáticos de un sistema. El plan *bang-bang* del invernadero tenía cinco acciones de conmutación que un invernadero real no dejaría en manos de un planificador: eso lo hace un termostato. El dominio `greenhouse-auto` quita las dos acciones de conmutación y añade una acción, `enable-thermostat`, y dos eventos (Listado 18.6).

```
;; The thermostat: what the agent decided by hand, now as events
(:event too-cold
  :parameters ()
  :precondition (and (thermostat-on) (not (heater-on))
                     (<= (temperature) (low)))
  :effect (heater-on))

(:event too-warm
  :parameters ()
  :precondition (and (thermostat-on) (heater-on)
                     (>= (temperature) (high)))
  :effect (not (heater-on))))
```

```
$ kairos greenhouse-auto-domain.pddl greenhouse-auto-p01.pddl
; cost = 24.5
; makespan = 24.500
0.100: (enable-thermostat) [0.000]
24.500: (harvest) [0.000]
24.500: @PlanEND
; better plan 2
; cost = 24.4
; makespan = 24.400
0.000: (enable-thermostat) [0.000]
24.400: (harvest) [0.000]
24.400: @PlanEND
```

Listado 18.6: Un termostato son dos eventos (parte del dominio `greenhouse-auto`).

Dos acciones. Todo lo demás es el mundo: en el instante en que se activa el termostato, a 15 grados, se dispara `too-cold` y el calefactor se enciende (una acción que dispara un evento en el mismo instante). Si se reproduce el modelo en tiempo continuo, `too-warm` se dispara a los 9,16 s, `too-cold` a los 11,17 s, y así sucesivamente, siete eventos en total, en instantes que no caen en ninguna rejilla (Figura 18.4). La temperatura se mantiene entre 23 y 27, y las plantas están listas medio segundo antes que con las conmutaciones del planificador.

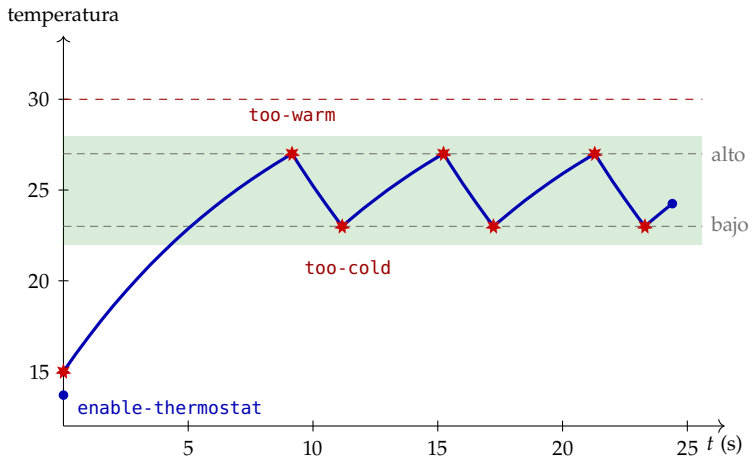


Figura 18.4.: El invernadero con el termostato. Las estrellas son eventos: entre el primer instante y la cosecha no aparece ninguna acción del agente. Compárala con la Figura 18.2: el mismo diente de sierra, dibujado por el mundo y no por el plan.

Para el modelador, pasar las conmutaciones del plan a los eventos es decidir *dónde está la inteligencia*. Si el sistema tiene un termostato, modélalo: los planes son más cortos, la búsqueda es más pequeña (8 030 estados en lugar de 432 899 en estas dos ejecuciones) y el plan ya no depende de que el planificador acierte con las décimas de segundo justas. Si las conmutaciones son lo que queremos que el planificador averigüe, déjalas como acciones.

Eventos que no paran

Cada evento del termostato se desactiva a sí mismo: `too-warm` necesita el calefactor encendido y lo apaga. Pero la regla del Capítulo 17, que un evento debe hacer falsa su propia condición, no basta cuando hay varios eventos. Dos eventos pueden activar cada uno *al otro*. El intermitente del Listado 18.7 es el caso más pequeño.

```
22
23  (:event light-up
24    :parameters ()
25    :precondition (and (blinking) (not (lit)))
26    :effect (lit))
27
28  (:event go-dark
29    :parameters ()
30    :precondition (and (blinking) (lit))
31    :effect (not (lit))))
```

Listado 18.7: Mal a propósito: dos eventos que se deshacen el uno al otro (parte del dominio `blinker-wrong`).

Cada evento, por separado, es correcto. Juntos, en cuanto el intermitente está activado, la lámpara se enciende, lo que habilita `go-dark`, que habilita `light-up`, y así sucesivamente, infinitas veces en el instante cero. `kairos check` acepta el dominio, ya que su sintaxis no tiene nada de malo. El planificador no encuentra plan para el problema «dejar pasar tres segundos», y el validador explica por qué cuando se le da el plan obvio:

```
$ kairos validate blinker-wrong-domain.pddl blinker-wrong-p01.pddl
    blinker-wrong-p01.plan
blinker-wrong-p01.plan: error: events keep firing at 0.000000 (go-
    dark): an event must make its own condition false [validate.
    event_loop]
plan invalid
```

La solución es poner tiempo entre los eventos. En el dominio `blinker`, un temporizador, movido por un proceso, debe llegar a medio segundo antes de que pueda dispararse cualquiera de los dos eventos, y cada evento lo reinicia:

```
  ;; Each event resets the timer: its own condition becomes false,
  ;; and the condition of the other needs half a period to hold
  (:event light-up
    :parameters ()
    :precondition (and (blinking) (not (lit))
                       (>= (timer) (half-period)))
    :effect (and (lit) (assign (timer) 0)
                 (increase (flashes) 1)))

  (:event go-dark
    :parameters ()
    :precondition (and (blinking) (lit)
                       (>= (timer) (half-period)))
    :effect (and (not (lit)) (assign (timer) 0))))
```

Listado 18.8: El intermitente arreglado: cada evento reinicia un temporizador al que el otro tiene que esperar (parte del dominio `blinker`).

```
$ kairos blinker-domain.pddl blinker-p01.pddl
; cost = 2.5
; makespan = 2.500
0.000: (switch-on) [0.000]
2.500: @PlanEND
```

La lámpara se enciende a los 0,5, 1,5 y 2,5 segundos; el tercer destello es el objetivo. La pareja «proceso que hace avanzar un temporizador, evento en un umbral que lo reinicia» es la expresión idiomática de PDDL+ para todo lo periódico y para todo lo que llega con retraso.

El termostato esconde una versión más sutil del mismo problema. Tiene dos umbrales, 23 y 27, y tras cada evento la temperatura necesita segundos para llegar al otro. Esa separación entre los umbrales se llama **histéresis**. El problema `greenhouse-auto-p02` pone los dos umbrales en 25. Ahora, en el momento en que la temperatura llega a 25, `too-warm` apaga el calefactor, y a esa misma temperatura `too-cold` vuelve a encenderlo. Entre uno y otro no pasa el tiempo: el modelo pide infinitos eventos alrededor de un instante, lo que se conoce como *comportamiento de Zenón*, por la paradoja. En la práctica:

```
$ kairos greenhouse-auto-domain.pddl greenhouse-auto-p02.pddl --
    timeout 3
note: no plan within the time limit of 3 s (34803 states searched).
    More time may find one: --timeout (SolveOptions::timeout). More
     threads run the phases of the search at once: --threads (
    SolveOptions::threads). [orchestrator.why]
kairos: timeout (timeout)
```

El planificador no devuelve nada, ni con 3 segundos ni con 20. El validador, al que se le da el plan «activar el termostato y cosechar a los 40 segundos», lo acepta: como calcula con números de precisión finita, nunca ve la temperatura exactamente en 25 con las dos condiciones ciertas, y sigue al calefactor mientras se enciende y se apaga muchas veces por segundo. Ninguno de los dos comportamientos es un veredicto sobre el modelo, y esa es la lección: un modelo con comportamiento de Zenón no falla con un mensaje claro. Falla volviendo lentas las herramientas, o haciendo que no se pongan de acuerdo. Los termostatos reales tienen histéresis por una razón parecida, para proteger al calefactor de conmutar sin descanso. Los modelos también la necesitan.

Paso D: las preguntas para un conjunto de eventos

Para cada evento por separado: ¿qué hace falsa su condición una vez que se ha disparado? Para los eventos en conjunto: dibuja una flecha del evento A al evento B si un efecto de A puede hacer cierta la condición de B. Sigue las flechas. Una cadena que termina es una cascada, y está bien. Un ciclo solo está bien si en algún punto de él *tiene que pasar tiempo*: una cantidad tiene que viajar de un umbral a otro (histéresis), o un temporizador tiene que correr. Un ciclo que puede cerrarse en tiempo cero es un fallo. Para dos eventos que pueden hacerse ciertos en el mismo instante y tocan los mismos hechos: haz que sus condiciones sean excluyentes, como en las dos maneras de llegar al suelo.

18.5. Procesos con acciones durativas y literales temporizados

Nada obliga a que un modelo híbrido sea puro. Los procesos y los eventos conviven en el mismo dominio con las acciones durativas del Capítulo 10 y los literales iniciales temporizados del Capítulo 13, y la comparativa del final del Capítulo 17 mostró por qué nos interesa: un desplazamiento cuya duración se conoce al empezar se escribe mejor como una acción durativa que como una acción, un proceso y un evento. La regla práctica es usar cada construcción para la clase de cambio que le corresponde:

- una **acción durativa** para una actividad del agente que, una vez empezada, sigue su curso;
- un **proceso** para un cambio que depende del estado y no de lo que esté haciendo el agente;
- un **evento** para una consecuencia que el mundo saca por sí mismo;
- un **literal inicial temporizado** para un cambio que llega a una hora del reloj, sea cual sea el estado.

El dominio `courier-plus` es el mundo del mensajero escrito de esa manera. Los desplazamientos son acciones durativas. La batería es cosa de tres procesos y dos eventos (Listado 18.9), y cargar ya no es una acción: el agente enchufa y desenchufa el robot, y el cargador hace el resto.

```
(:durative-action move
  :parameters (?r - robot ?from ?to - location)
  :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
  :condition (and (at start (robot-at ?r ?from))
                  (at start (connected ?from ?to))
                  (at start (not (plugged ?r)))
                  (over all (powered ?r)))
  :effect (and (at start (not (robot-at ?r ?from)))
               (at start (driving ?r))
               (at end (not (driving ?r)))
               (at end (robot-at ?r ?to))))
;; Three processes change the same battery; their rates add up
(:process drain
  :parameters (?r - robot)
  :precondition (powered ?r)
  :effect (decrease (battery ?r) (* #t (drain-rate ?r))))

(:process motor-drain
  :parameters (?r - robot)
  :precondition (and (powered ?r) (driving ?r))
  :effect (decrease (battery ?r) (* #t (motor-rate ?r))))

(:process charging
```

```
    :parameters (?r - robot)
    :precondition (plugged ?r)
    :effect (increase (battery ?r) (* #t (charge-rate ?r))))

  ;; The charger cuts off by itself when the battery is full
  (:event battery-full
    :parameters (?r - robot)
    :precondition (and (plugged ?r)
                       (>= (battery ?r) (battery-capacity ?r)))
    :effect (not (plugged ?r)))

  (:event shutdown
    :parameters (?r - robot)
    :precondition (and (powered ?r) (<= (battery ?r) 0))
    :effect (not (powered ?r))))
```

Listado 18.9: La batería de `courier-plus`: tres procesos y dos eventos (parte del dominio).

El `move` durativo marca el robot como `(driving ?r)` desde su inicio hasta su final. Ese hecho es el enlace entre la acción y el proceso `motor-drain`: la acción no toca la batería, solo le dice al mundo que los motores están en marcha. Los ritmos se suman como antes. Un robot encendido pierde 0,2 por segundo; en marcha, $0{,}2 + 0{,}8 = 1$; enchufado y encendido, gana $2 - 0{,}2 = 1{,}8$. La condición `(over all (powered ?r))` ata en el otro sentido: si el evento `shutdown` se dispara durante un desplazamiento, el desplazamiento se rompe y el plan no es válido. El evento `battery-full` es un dispositivo, como el termostato: el cargador se desconecta solo.

Tres problemas ponen a prueba el modelo. En `p01`, la batería está a 40 y la entrega, a 34 metros del vestíbulo, necesita 34: el plan es el clásico, cuatro desplazamientos, una recogida y una descarga. En `p02`, la batería está a 20:

```
$ kairos courier-plus-domain.pddl courier-plus-p02.pddl
; cost = 95.3
; makespan = 95.300
0.000: (move r1 lobby pharmacy) [6.000]
6.000: (pick r1 meds1 pharmacy) [0.000]
6.000: (move r1 pharmacy lobby) [6.000]
12.000: (move r1 lobby dock) [4.000]
16.000: (plug r1 dock) [0.000]
69.300: (unplug r1) [0.000]
69.300: (move r1 dock lobby) [4.000]
73.300: (move r1 lobby corridor) [10.000]
83.300: (move r1 corridor ward-a) [12.000]
95.300: (drop r1 meds1 ward-a) [0.000]
95.300: @PlanEND
```

El robot recoge los medicamentos, va a la base de carga con 4 unidades y se enchufa. Se queda enchufado 53,3 segundos y se desenchufa con la batería a 99,9, un paso antes de que lo hubiera hecho el evento `battery-full`. Para los 26 metros que quedan necesitaba 26. Es la espera larga del Capítulo 17 en acción: una vez enchufado, la búsqueda deja correr el tiempo hasta el último paso antes de que cambie alguna condición numérica, y el cambio más cercano era que la batería se llenara. El plan es válido y derrochador, y es el mismo derroche que comete un planificador cuando prueba las duraciones de un `charge` durativo solo en sus extremos.

Esperar no sale gratis

El problema `courier-plus-p03` añade un reloj. La batería está a 10, y la farmacia está cerrada: un literal inicial temporizado la abre a los

60 segundos.

```
    ; the pharmacy opens one minute from now
    (at 60 (is-open pharmacy))
```

En los modelos temporales de la Parte IV, el robot simplemente esperaría en el vestíbulo. Aquí, un minuto de espera cuesta 12 unidades de batería, y el robot tiene 10.

```
$ kairos courier-plus-domain.pddl courier-plus-p03.pddl
; cost = 98
; makespan = 98.000
0.000: (move r1 lobby dock) [4.000]
33.900: (plug r1 dock) [0.000]
60.000: (unplug r1) [0.000]
60.000: (move r1 dock lobby) [4.000]
64.000: (move r1 lobby pharmacy) [6.000]
70.000: (pick r1 meds1 pharmacy) [0.000]
70.000: (move r1 pharmacy lobby) [6.000]
76.000: (move r1 lobby corridor) [10.000]
86.000: (move r1 corridor ward-a) [12.000]
98.000: (drop r1 meds1 ward-a) [0.000]
98.000: @PlanEND
```

La Figura 18.5 sigue la batería. El robot va enseguida a la base de carga y llega con 6 unidades. Después hace algo que ninguna persona haría: se queda medio minuto junto al cargador y se enchufa a los 33,9 segundos, con 0,02 unidades, un paso antes del apagado. Carga hasta que abre la farmacia, a los 60 segundos exactos (los literales temporizados conservan sus instantes exactos), y hace la entrega con 47 unidades, de las que necesita 38.

El plan es correcto, y nos dice cómo razona la búsqueda. Desde la base de carga, una espera larga corre hasta el último paso antes de que algo cambie; lo primero que cambiaría es que la batería llegara a cero. Ahí la búsqueda tiene que actuar, y enchufarse es la acción que ayuda. Un plan que se enchufa nada más llegar y sale de la base de carga a tiempo para estar en la farmacia cuando abre es diez segundos más corto (Ejercicio 18.5.1), y el planificador no tenía ningún motivo para buscarlo. Lo que importa para el modelo es que los dos planes respetan el mismo hecho: la espera tenía que pasarse junto al cargador.

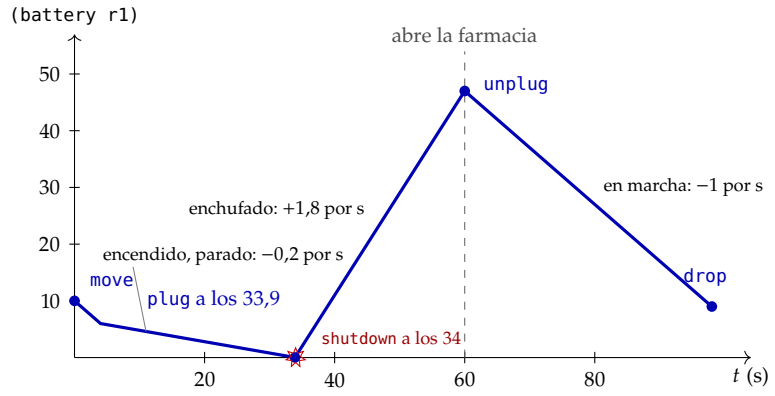


Figura 18.5.: La batería en el plan de `courier-plus-p03`. Cada pendiente es una suma de los ritmos de los procesos activos en ese tramo. La línea discontinua es el literal inicial temporizado. La estrella hueca es el evento que el plan evita por un paso.

¿Cómo mezcla Kairos las construcciones? Su documentación dice que, en un modelo con procesos, una acción durativa se ejecuta como un inicio, un reloj y un final, con su final y cada literal temporizado en sus instantes exactos, y que hay una tercera clase de espera, de longitud épsilon, para

separar acontecimientos que no deben coincidir. Por eso las duraciones del plan son exactas, y la búsqueda es pequeña: 736 estados para `p02`, cuando el mensajero de PDDL+ puro del Capítulo 17 necesitaba decenas de miles para un problema más sencillo.

18.6. Un sistema, tres modelos

El mensajero con batería se ha modelado ya en tres capítulos con tres juegos de herramientas. Esta sección pone los tres uno junto a otro sobre la misma tarea: el robot está en el vestíbulo con la batería a 20, los medicamentos están en la farmacia, la sala A está a 34 metros de recorrido y hay un cargador en la base de carga.

Nivel 1: números. El dominio `courier-discrete` usa solo los fluentes numéricos del Capítulo 8. Un desplazamiento resta la distancia por el consumo; `charge` pone la batería a su capacidad.

```
  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                       (>= (battery ?r) (* (distance ?from ?to)
                                           (consumption ?r))))
    :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
                 (decrease (battery ?r) (* (distance ?from ?to)
                                           (consumption ?r)))))
  ;; Charging fills the battery in one step
  (:action charge
    :parameters (?r - robot ?l - location)
    :precondition (and (robot-at ?r ?l) (has-charger ?l))
    :effect (assign (battery ?r) (battery-capacity ?r))))
```

```
$ kairos courier-discrete-domain.pddl courier-discrete-p01.pddl
; cost = 9
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby dock)
(charge r1 dock)
(move r1 dock lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Listado 18.10: Nivel 1: la batería cambia a saltos (parte del dominio `courier-discrete`).

Nueve acciones, encontradas en un milisegundo y con optimalidad demostrada. El plan dice qué hacer y en qué orden. No dice cuánto tarda nada, y menos que nada la carga.

Nivel 2: acciones durativas con efectos continuos. El segundo dominio, `courier-durative`, usa las herramientas del Capítulo 12. La batería baja de forma continua durante un desplazamiento y sube de forma continua durante un `charge` cuya duración elige el planificador, hasta la que llena la batería.

```
  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
    :condition (and (at start (robot-at ?r ?from))
                    (at start (connected ?from ?to))
                    (at start (idle ?r))
                    (over all (>= (battery ?r) 0)))
    :effect (and (at start (not (robot-at ?r ?from)))
                 (at start (not (idle ?r)))
```

```
                (decrease (battery ?r) (* #t (drain-rate ?r)))
                (at end (robot-at ?r ?to))
                (at end (idle ?r))))

  ;; The planner chooses for how long to charge, up to full
  (:durative-action charge
    :parameters (?r - robot ?l - location)
    :duration (<= ?duration (/ (- (battery-capacity ?r)
                                  (battery ?r))
                               (charge-rate ?r)))
    :condition (and (at start (idle ?r))
                    (over all (robot-at ?r ?l))
                    (over all (has-charger ?l)))
    :effect (and (at start (not (idle ?r)))
                 (increase (battery ?r) (* #t (charge-rate ?r)))
                 (at end (idle ?r))))
```

```
$ kairos courier-durative-domain.pddl courier-durative-p01.pddl
; cost = 92.00800000000002
; makespan = 92.008
0.000: (move r1 lobby pharmacy) [6.000]
6.001: (pick r1 meds1 pharmacy) [1.000]
7.002: (move r1 pharmacy lobby) [6.000]
13.003: (move r1 lobby dock) [4.000]
17.004: (charge r1 dock) [48.000]
...
; better plan 2
; cost = 86.008
; makespan = 86.008
0.000: (move r1 lobby dock) [4.000]
4.001: (charge r1 dock) [42.000]
46.002: (move r1 dock lobby) [4.000]
50.003: (move r1 lobby pharmacy) [6.000]
56.004: (pick r1 meds1 pharmacy) [1.000]
57.005: (move r1 pharmacy lobby) [6.000]
63.006: (move r1 lobby corridor) [10.000]
73.007: (move r1 corridor ward-a) [12.000]
85.008: (drop r1 meds1 ward-a) [1.000]
```

Listado 18.11: Nivel 2: la batería cambia mientras se ejecuta una acción (parte del dominio `courier-durative`).

Ahora el plan tiene instantes. Los dos planes cargan durante la máxima duración permitida, 48 y después 42 segundos, con la batería llena cada vez: el extremo de la desigualdad es donde la búsqueda mira primero. El modelo sabe que cargar lleva tiempo y que un desplazamiento necesita batería durante todo el recorrido. No sabe que un robot parado gasta algo.

Nivel 3: PDDL+. El dominio `courier-plus` de la sección anterior, con su problema `p02`: el plan de 95,3 segundos que ya hemos leído. La batería se descarga siempre que el robot está encendido, cargar es un estado del mundo y una batería vacía es un evento.

Lo que cada nivel no puede decir

Los tres planes son la misma ruta con la misma parada en la base de carga. Los niveles se diferencian en los problemas que resuelven *mal*. Dale a cada uno el escenario de `p03`, una farmacia que abre dentro de un minuto y una batería a 10. El nivel 1 no tiene tiempo, así que ni siquiera puede enunciar la hora de apertura. El nivel 2 puede enunciarla, con un literal temporizado, y responderá con un robot que espera un minuto en el vestíbulo y luego hace la entrega: en ese modelo, a un robot que no está ejecutando una acción no le pasa nada. Solo el nivel 3 sabe que la espera vacía la batería. En el otro extremo, dale a cada uno un día con

cuarenta entregas: el nivel 1 lo planificará y los otros dos, con la mayoría de los planificadores, no.

	1. Números	**2. Durativas, continuas**	**3. PDDL+**
Fidelidad	Orden y energía de los desplazamientos. Sin tiempo: cargar es instantáneo, esperar no existe.	Tiempo, solapamiento, energía durante las acciones. El cambio se para cuando se paran las acciones: esperar es gratis.	Cambio siempre que el estado lo dice: descarga en reposo, apagado, el cargador que se corta.
Tamaño	31 acciones instanciadas	31 acciones durativas	11 acciones, 14 acciones durativas, 3 procesos, 2 eventos
Guía	220 estados, óptimo demostrado	9 629 estados; duraciones probadas en sus extremos	736 estados aquí; en general, el más difícil, porque cada instante es una decisión
Claridad	El más sencillo de leer y de explicar	Una acción por actividad; los planes son cronogramas	Las leyes están separadas de las decisiones; para leer los planes hay que reproducirlos
Portabilidad	Cualquier planificador numérico: cinco de los ocho del Apéndice B	Efectos continuos: dos de los ocho	Procesos: uno de los ocho

Tabla 18.1.: Alternativas comparadas: el mensajero que recarga, en tres niveles. Las cifras de Tamaño y Guía son las de `kairos -v` sobre los tres problemas de esta sección.

Veredicto. Elige el nivel más bajo que no mienta sobre aquello de lo que depende el problema. El nivel 1, cuando la pregunta es qué acciones y en qué orden, y las cantidades son presupuestos que las acciones gastan: casi toda la logística y casi todas las misiones de robots vistas desde lo bastante lejos. El nivel 2, cuando el tiempo forma parte de la pregunta (plazos, horarios de apertura, robots que trabajan en paralelo) y todo cambio puede atribuirse a una actividad del agente. El nivel 3, cuando algo cambia mientras el agente no hace nada, o cuando el mundo interrumpe: descargarse, calentarse, llenarse, caer, y los eventos que vienen después. Dentro del nivel 3, conserva como acciones durativas todo lo que habría sido una en el nivel 2. Los niveles son también un camino: el Capítulo 19 muestra cómo hacer crecer un modelo de uno al siguiente, conservando las pruebas del nivel inferior como pruebas del superior.

18.7. Límites: pasos y tolerancias

El Capítulo 17 terminaba su explicación de cómo planifica Kairos con una advertencia: un plan que necesita una precisión temporal más fina que los pasos de la búsqueda no se encuentra. Un modelador debe saber qué aspecto tiene eso, porque el síntoma, `no plan found`, es el mismo que el de un modelo que está mal.

El dominio `doser` es un depósito que hay que llenar hasta una dosis de 8 litros, con un margen de 0,1 arriba o abajo (Listado 18.12). A 2 litros por segundo (`p01`), el plan cierra la válvula a los 4 segundos. El problema `p02` tiene una bomba más rápida, de 7 litros por segundo.

```
;; Filling a dose. The valve is opened by the agent and closed
;; either by the agent or, if armed, by a float switch.
(define (domain doser)
```

```
  (:requirements :negative-preconditions :numeric-fluents :time)

  (:predicates
    (valve-open)
    (switch-armed))           ; the float switch is in use

  (:functions
    (level)                   ; liters in the vessel
    (inflow)                  ; liters per second coming in
    (target))                 ; liters at which the switch closes

  (:action open-valve
    :parameters ()
    :precondition (not (valve-open))
    :effect (valve-open))

  (:action close-valve
    :parameters ()
    :precondition (valve-open)
    :effect (not (valve-open)))

  (:process fill
    :parameters ()
    :precondition (valve-open)
    :effect (increase (level) (* #t (inflow))))

  ;; An event fires at the exact instant, whatever the steps
  (:event float-switch
    :parameters ()
    :precondition (and (switch-armed) (valve-open)
                       (>= (level) (target)))
    :effect (not (valve-open))))
```

Listado 18.12: El dosificador, con un interruptor de flotador que el problema puede armar (`examples/ch18/doser-domain.pddl`).

```
$ kairos doser-domain.pddl doser-p02.pddl
note: no plan in steps of time; a plan that needs finer timing than
     the steps would be missed, so this is evidence but not a proof
     [orchestrator.steps_exhausted]
note: no plan: the engines that apply ran out of states (100 states)
      before the time limit, so more time does not help; none of
     them is complete for this task, so a plan may still exist (see
     the note before). [orchestrator.why]
kairos: no plan found (exhausted)
$ kairos validate doser-domain.pddl doser-p02.pddl doser-p02-hand.
     plan
plan valid, makespan 1.143
```

Un plan existe: el escrito a mano, que cierra la válvula a los 1,143 segundos, es válido. El nivel está dentro de la dosis desde 1,129 hasta 1,157 segundos, una ventana de 29 milisegundos de ancho. La búsqueda mira en 1,1 s, donde el nivel es 7,7, y en 1,2 s, donde es 8,4 (Figura 18.6), y en nada de lo que hay entre medias.

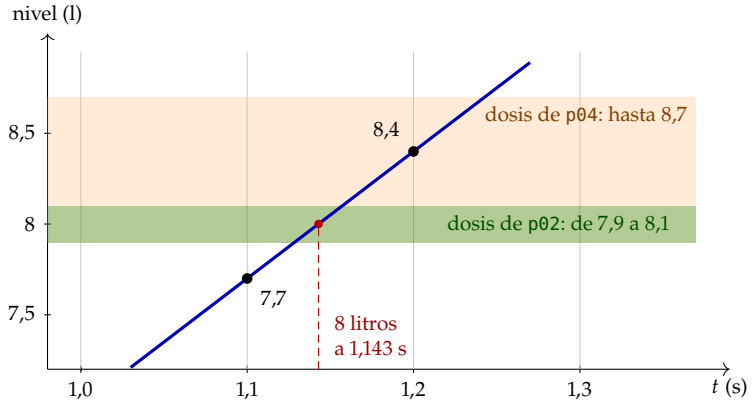


Figura 18.6.: El dosificador a 7 litros por segundo. El nivel (línea) está dentro de la dosis estrecha (franja oscura) durante 29 ms, entre dos estados de la búsqueda (puntos), que están a 0,7 litros el uno del otro. La dosis más ancha (franja clara) contiene un punto.

Un evento no rescata a la búsqueda

Las secciones anteriores sugieren un remedio: dejar que un evento haga la parte precisa. El problema `p03` arma el interruptor de flotador, que cierra la válvula cuando el nivel llega a 8 litros. En tiempo continuo el plan es trivial, abrir la válvula y esperar, y el validador está de acuerdo (`doser-p03-hand.plan` es válido). Sin embargo:

```
$ kairos doser-domain.pddl doser-p03.pddl
...
kairos: no plan found (exhausted)
```

La razón está en cómo trata la búsqueda los eventos: los dispara *entre pasos*. En el mundo por pasos, el interruptor se comprueba a 1,1 s (7,7 litros, todavía no) y a 1,2 s (8,4 litros, ahora), y cierra la válvula con 8,4 litros en el recipiente. La búsqueda nunca visita un estado con la dosis correcta, así que no tiene ningún candidato que pasar al validador, que habría disparado el evento a los 1,143 s. El evento hace preciso el *mundo*, no la búsqueda.

Lo que ayuda es un modelo en el que los estados de la búsqueda también sean aceptables. El problema `p04` conserva el interruptor y acepta dosis de hasta 8,7 litros, los 0,7 litros que entran en un paso:

```
$ kairos doser-domain.pddl doser-p04.pddl
; cost = 1.2
; makespan = 1.200
0.000: (open-valve) [0.000]
1.200: @PlanEND
```

La búsqueda llega a 1,2 s con la válvula cerrada por el interruptor y, según su propia aritmética, 8,4 litros. El validador reproduce después el plan en tiempo continuo: el interruptor cierra a los 1,143 s con 8 litros, el objetivo se cumple, el plan se imprime. El plan es mejor en el mundo que en la búsqueda que lo encontró.

Una ventana en la que un plan debe acertar debería ser al menos tan ancha como lo que cambia en un paso: el ritmo por 0,1 s. Las ventanas más estrechas dan «no plan found» en problemas que tienen plan.

De aquí se sigue una regla para el modelador. **Toda ventana en la que un plan deba acertar debería ser al menos tan ancha como el cambio de un paso**: el ritmo de la cantidad por 0,1 s. Una velocidad que cambia a 10 m/s^2 se mueve 1 m/s por paso, así que una ventana sobre la velocidad debería tener 1 m/s de ancho o más. Si la historia necesita una ventana más estrecha, pregúntate si la historia de verdad la necesita (pocas válvulas

cierran al milisegundo), frena el proceso modelando un actuador más fino, o acepta que este problema queda fuera de lo que encuentra una búsqueda por pasos y valida planes obtenidos de otro modo.

Valores exactos

El caso extremo de una ventana estrecha es la igualdad. El problema `doser-p05` pide `(= (level) 8)`. Ningún paso cae en ese valor, y tampoco un plan escrito a mano con precisión de milisegundos:

```
$ kairos validate doser-domain.pddl doser-p05.pddl doser-p05-hand.
    plan
doser-p05-hand.plan: error: the goal does not hold at the end of the
     plan: (= (level) 8): 8.001 = 8 is false [validate.
    goal_not_reached]
plan invalid
```

Cerrar a los 1,143 s deja 8,001 litros, y 8,001 no es 8. El validador compara de forma exacta los números que producen los procesos, como debe: una tolerancia lo bastante generosa para aceptar este plan ocultaría también violaciones reales de umbrales. Ningún instante decimal da exactamente 8 litros a 7 litros por segundo. Una cantidad que cambia un proceso debe compararse con `>=` y `<=`, nunca con `=`. Las únicas igualdades seguras son las que se hacen sobre valores que una acción o un evento ha *asignado*: la velocidad que `halt` puso a cero en el coche con resistencia del aire es exactamente cero, y `park` podía pedirla.

18.8. El método MODEL: probar y elegir modelos híbridos

El capítulo amplía dos pasos del método. Al paso E, un procedimiento para probar un modelo en el que la mayor parte de lo que ocurre no está en el plan. Al paso L, una decisión que va antes de cualquier comparativa de codificaciones: en cuál de los tres niveles modelar.

Paso E para un modelo híbrido

1. **Primero, el lápiz.** Para cada cantidad, calcula adónde va si no se hace nada y si cada interruptor se deja en cada posición (los 35 y los 5 grados del invernadero). Esas cifras dicen qué problemas deberían tener plan.
2. **Comprobar cada evento.** Para cada evento, escribe un problema cuyo único plan pase por él, o cuyo objetivo sea su efecto (la inundación como objetivo). Un evento que no se ha disparado nunca en una prueba no se ha probado. Comprueba que se dispara una vez, y dibuja las flechas entre eventos para buscar ciclos en tiempo cero.
3. **Acotar el horizonte.** Un problema sin plan se explora hasta agotarlo, y la búsqueda crece con el tiempo durante el que las cosas pueden seguir cambiando. Da a los problemas «sin plan» márgenes pequeños (una batería para 16 segundos, no para 300) para que terminen en segundos.
4. **Validar en tiempo continuo.** Escribe a mano el plan que esperas, con instantes que *no* sean múltiplos de 0,1, y el plan

que temes, y dáselos los dos a `kairos validate`. Es el juez del modelo; el planificador es solo una fuente de planes.

5. **Reproducir y dibujar.** Calcula las cantidades a lo largo del plan del planificador y dibújalas, como hacen las figuras de estos dos capítulos. Un plan que es válido por una razón equivocada (un proceso que actuó gratis) se ve enseguida en una gráfica y casi nunca en la lista de acciones.
6. **Medir las ventanas.** Para cada condición que el plan deba cumplir mientras una cantidad se mueve, compara el ancho de la ventana con el ritmo por 0,1 s.

La Tabla 18.2 recoge las preguntas nuevas, y la Figura 18.7 es la guía de decisión del paso L.

Paso	Preguntas nuevas
M	¿Hay algo que no deba ocurrir nunca, en ningún plan? (Una restricción global.) ¿Un mal resultado está prohibido, o solo tiene consecuencias?
O	Para cada ritmo: ¿es una constante, otra cantidad que cambia o la propia cantidad?
D	¿Qué procesos tocan la misma cantidad, y cuál es la suma de sus ritmos en cada situación? ¿Qué eventos pueden disparar a cuáles, y dónde tiene que pasar tiempo en cada ciclo? ¿Qué dispositivos del sistema son eventos, y qué decisiones se dejan al agente?
E	Los seis puntos de la caja anterior.
L	¿Cuál es el nivel más bajo (números, acciones durativas, PDDL+) que no miente sobre aquello de lo que depende el problema?

Tabla 18.2.: Lo que el modelado de sistemas híbridos añade al método MODEL.

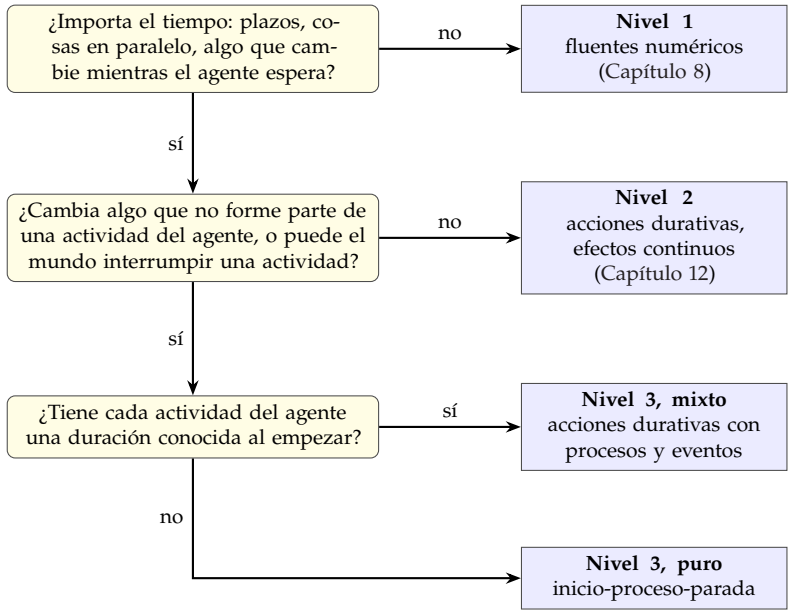


Figura 18.7.: Elegir el nivel de un modelo. Cada «sí» compra fidelidad y la paga en esfuerzo de búsqueda y en el número de planificadores que pueden leer el modelo.

Alternativas comparadas

Las alternativas de este capítulo son los tres niveles, escritos y ejecutados en la Sección 18.6 y juzgados según los cinco criterios en la Tabla 18.1, con su veredicto. La guía de la Figura 18.7 condensa esa tabla en tres preguntas. Dentro del nivel 3 queda una segunda elección, entre un evento que marca un fallo y una restricción global que lo prohíbe; se compara en el Ejercicio 18.1.1.

18.9. Errores comunes

Dos eventos que se activan el uno al otro

Tanto `light-up` como `go-dark` hacen falsa su propia condición, y juntos no paran nunca (`blinker-wrong-domain.pddl`). `kairos check` dice `valid, plannable`; el planificador responde `no plan found`; solo el validador, con un plan que llega al bucle, nombra el problema:

```
blinker-wrong-p01.plan: error: events keep firing at 0.000000
(go-dark): an event must make its own condition false
[validate.event_loop]
```

El mensaje habla de un evento porque es donde el validador se rindió; la causa es la pareja. Solución: haz que pase tiempo dentro de cada ciclo de eventos, con un temporizador o con dos umbrales distintos.

Un termostato con un solo umbral

«Calefactor apagado por encima de 25, encendido por debajo de 25» se lee con naturalidad y es un modelo de Zenón. Nada lo avisa. Con `greenhouse-auto-p02`, el planificador se quedó sin tiempo (`kairos: timeout (timeout)`) y el validador aceptó un plan escrito a mano, porque con precisión finita la temperatura nunca está exactamente en el umbral. Un planificador lento y unas herramientas que no se ponen de acuerdo son los síntomas. Solución: histéresis, dos umbrales lo bastante separados como para que la cantidad necesite varios pasos para ir de uno a otro.

Una ventana más estrecha que un paso

La dosis de 7,9 a 8,1 litros a 7 litros por segundo da `no plan found (exhausted)` tras 100 estados, aunque existe un plan. La señal delatora es el tamaño de la búsqueda: un modelo demasiado estricto suele tardar muchos estados en agotarse, y una ventana que se pasa por alto tarda pocos, porque la búsqueda pasa de largo. Confírmalo con un plan escrito a mano. Solución: ensancha la ventana hasta el ritmo por 0,1 s, o frena el proceso.

Igualdad sobre una cantidad que fluye

`(= (level) 8)` en un objetivo o en una precondición, cuando un proceso cambia el nivel, es una ventana de ancho cero. El validador es exacto:

```
doser-p05-hand.plan: error: the goal does not hold at the end
of the plan: (= (level) 8): 8.001 = 8 is false
[validate.goal_not_reached]
```

Solución: dos desigualdades con una tolerancia, o un evento que asigne el valor exacto cuando se cruza el umbral.

Esperar el plan más corto

Los planes de `courier-plus` cargan hasta el 99,9 % cuando bastaba con el 30 %, y esperan junto al cargador antes de enchufarse. Son válidos. En PDDL+, Kairos minimiza el número de acciones, y sus esperas largas se detienen donde algo está a punto de cambiar, no donde sería conveniente. Solución, cuando la duración importa: pon la presión en el modelo (un plazo como literal temporizado, una

batería más pequeña), o mejora el plan a mano y valídalo.

Una restricción donde hacía falta un evento, y al revés

Una restricción global «la batería nunca está vacía» prohíbe todos los planes en los que un robot se queda sin batería, incluidos aquellos en los que un segundo robot rescata al primero. Un evento que marca al robot como `stranded`, con un objetivo que diga qué robots no deben estarlo, los permite. El error contrario es el límite de velocidad impuesto con un evento y un objetivo que algún problema olvida. Pregunta de cada regla: ¿incumplirla es imposible, está prohibido o solo es costoso? Lo imposible es un proceso o un evento del mundo; lo prohibido, una restricción; lo costoso, un evento con consecuencias.

18.10. Resumen

Los modelos híbridos se construyen con pocas construcciones, y su dificultad está en cómo se combinan.

- Una **restricción global**, escrita con `:constraint`, un nombre, `:parameters` y una `:condition`, debe cumplirse en cada instante de cada plan. Viene de ENHSP, no del PDDL+ original. Úsala para lo que está prohibido; usa un evento para lo que tiene consecuencias.
- El cambio es **lineal** cuando el ritmo es constante, y **no lineal** cuando el ritmo es otra cantidad que cambia (polinómico) o la propia cantidad (exponencial). PDDL+ escribe los tres igual; Kairos los integra con Runge-Kutta y valida de forma exacta.
- Los procesos que cambian la misma cantidad suman sus ritmos. Un proceso puede activarse por un número que mueven otros procesos, y entonces el agente solo puede dirigir.
- Los eventos se disparan **en cascada** dentro de un instante. Un conjunto de eventos no debe contener un ciclo que se cierre en tiempo cero: pon un temporizador o **histéresis** en cada ciclo. Los eventos con un disparador común deberían tener condiciones excluyentes. Los eventos son también los dispositivos de un sistema.
- Los procesos y los eventos se combinan con acciones durativas y literales iniciales temporizados. Conserva como acción durativa toda actividad cuya duración se conozca al empezar.
- El mismo sistema puede modelarse con números, con acciones durativas y efectos continuos, o con PDDL+. Elige el nivel más bajo que no mienta sobre aquello de lo que depende el problema.
- Una búsqueda en pasos de tiempo pasa por alto las ventanas más estrechas que un paso, aunque un evento fuera a acertar en ellas. Haz las ventanas al menos tan anchas como el ritmo por 0,1 s, y no compruebes nunca la igualdad de una cantidad que fluye.
- El paso E para un modelo híbrido: primero el lápiz, comprobar cada evento, acotar el horizonte, validar en tiempo continuo, reproducir y dibujar, medir las ventanas.

18.11. Lecturas complementarias

La segunda mitad del artículo de Fox y Long Fox y Long, 2006 da la semántica de los procesos que interactúan y de los eventos en cascada, y discute qué modelos están bien formados; su módulo de aterrizaje

planetario es un ejemplo más grande en el espíritu de este capítulo. El tratamiento por intervalos de las condiciones numéricas no lineales está en el artículo sobre ENHSP Scala et al., 2016, el planificador del que vienen las restricciones globales de este capítulo. Para el cambio continuo lineal dentro de acciones durativas, el nivel 2 de este capítulo, consulta el artículo de PDDL 2.1 Fox y Long, 2003 y el planificador que lo hizo práctico Coles et al., 2010. La validación de planes con efectos continuos, incluida la manera de comprobar los invariantes entre acontecimientos, es el tema del artículo sobre VAL Howey et al., 2004. El libro de Haslum y otros Haslum et al., 2019 sitúa PDDL+ entre las demás extensiones del lenguaje, y Martín Rico et al., 2026 describe cómo elige Kairos entre sus motores para cada uno de los tres niveles.

18.12. Ejercicios

Ejercicios de la Sección 18.1 — Reglas que se cumplen en todo momento: restricciones globales

Ejercicio 18.1.1. Compara las dos codificaciones del límite de velocidad, el evento `radar` con el objetivo `(not (fined))` (Capítulo 17) y la restricción `keep-the-limit`, según los cinco criterios del paso L. Incluye en la Fidelidad un problema que una de las codificaciones resuelve mal cuando su autor se descuida.

Ejercicio 18.1.2. Reescribe el depósito del Capítulo 17 para que la inundación quede excluida por una restricción global y no por un evento. ¿Qué dice el validador del plan que cierra la válvula a los 6 segundos? ¿Qué puede expresar el modelo antiguo que el nuevo no puede?

Ejercicios de la Sección 18.2 — Cambio que no es lineal

Ejercicio 18.2.1. Para cada cantidad, di si su cambio es lineal, polinómico o exponencial, y por qué: (a) la batería de un robot que se descarga a 0,5 por segundo; (b) la altura de la pelota del Capítulo 17; (c) la temperatura del invernadero con el calefactor apagado; (d) el nivel de un depósito que se vacía por un agujero, más deprisa cuanto más lleno está; (e) el cuentakilómetros de un robot que se mueve a velocidad constante.

Ejercicio 18.2.2. Se sirve una taza de café a 90 grados en una habitación a 20, y se enfría a un ritmo de una décima parte de la diferencia por segundo. Bebérselo por encima de 65 grados quema la lengua; por debajo de 55 no merece la pena. Modélalo y encuentra cuándo beberlo. Calcula a mano el primer instante posible y compáralo con el plan.

Ejercicios de la Sección 18.3 — Procesos que interactúan

Ejercicio 18.3.1. En el invernadero, el calefactor tiene una potencia de 2 grados por segundo en lugar de 3. ¿Adónde va la temperatura con el calefactor siempre encendido? ¿Cuál es entonces el plan más corto, y cuántas acciones tiene? Responde con lápiz y luego cambia el problema y ejecútalo.

Ejercicio 18.3.2. En el plan de `drone-p01`, calcula la altitud y la velocidad en cada una de las tres acciones, y la velocidad a la que el dron chocaría con el suelo si los rotores no se encendieran nunca. ¿Qué

evento se dispararía en ese caso, y qué haría el planificador al respecto?

Ejercicios de la Sección 18.4 — Eventos en cascada y eventos que no paran

Ejercicio 18.4.1. Dibuja las flechas «un efecto de A puede hacer cierta la condición de B» entre los eventos de cuatro modelos: `blinker-wrong`, `blinker`, `greenhouse-auto` con umbrales 23 y 27, y con los dos umbrales en 25. Para cada ciclo, di qué tiene que ocurrir entre un evento y el siguiente, y cuánto tarda.

Ejercicio 18.4.2. Fichas de dominó que tardan: cada ficha necesita medio segundo para caer, y solo cuando está en el suelo golpea a la siguiente. Modélalo con un proceso y eventos, de modo que una fila de cinco tarde 2,5 segundos. ¿Podría ahora el agente salvar la tercera ficha, si tuviera una acción para ello?

Ejercicios de la Sección 18.5 — Procesos con acciones durativas y literales temporizados

Ejercicio 18.5.1. Para `courier-plus-p03`, escribe a mano un plan en el que el robot se enchufe en cuanto llegue a la base de carga y esté en la farmacia cuando abra. Valídalo. ¿Cuánto más corto es que el plan de Kairos? ¿Por qué el `pick` tiene que ir un poco después de los 60 segundos y no a los 60?

Ejercicio 18.5.2. La farmacia está abierta ahora y cierra dentro de 8 segundos; la batería está a 40. Escribe el problema con un literal inicial temporizado, predice si hay plan y ejecútalo. Lee los instantes del plan: ¿hay algo raro en él?

Ejercicios de la Sección 18.6 — Un sistema, tres modelos

Ejercicio 18.6.1. Elige un nivel para cada historia con la guía de la Figura 18.7, y di qué pregunta decidió: (a) un camión con un depósito de combustible reparte en diez tiendas; (b) dos robots pintan una habitación, y a una pared solo se le puede dar una segunda mano cuando la primera lleva una hora secándose; (c) un horno que sigue calentándose mientras el cocinero prepara la masa; (d) un satélite que solo puede transmitir mientras se ve una estación de tierra, en instantes conocidos de antemano.

Ejercicio 18.6.2. En el modelo de nivel 2 de la Sección 18.6, la carga duró 42 segundos, y en el de nivel 3, 53,3. Explica cada cifra. Después explica por qué ninguno de los dos planificadores cargó solo la docena de segundos que necesita la tarea.

Ejercicios de la Sección 18.7 — Límites: pasos y tolerancias

Ejercicio 18.7.1. El dosificador de `p02` con una bomba de 5 litros por segundo. ¿Es la ventana más ancha que un paso? ¿Encuentra Kairos un plan? Concilia las dos respuestas.

Ejercicio 18.7.2. Un compañero propone resolver `doser-p03` poniendo el objetivo del interruptor de flotador en 7,6 litros. Predice, con la visión por pasos de la búsqueda, el nivel en el que la búsqueda ve cerrarse la válvula, y el nivel en el que se cierra en tiempo continuo. ¿Es la propuesta una solución?

Ejercicios de la Sección 18.8 — El método MODEL: probar y elegir modelos híbridos

Ejercicio 18.8.1. Completa la batería de pruebas del dron. Además del problema típico (`p01`), escribe uno trivial, uno que no deba tener plan y uno que tiente al planificador a incumplir una regla. Para cada uno, di qué esperas antes de ejecutarlo.

Ejercicio 18.8.2. Aplica los seis puntos del paso E al dominio `courier-plus`: para cada punto, nombra el problema o el plan de este capítulo que lo cubre, o di que falta y escribe lo que habría que añadir.

Parte VII. El método en acción

El método MODEL, completo | 19.

Un estudiante que ha leído los dieciocho capítulos anteriores conoce todas las construcciones de PDDL y las ha visto funcionar. Démosle ahora a ese estudiante un fichero en blanco y un párrafo: *una lavandería, tres coladas, una lavadora, una secadora, un jersey de lana que no puede ir a la secadora y una mañana que no dura para siempre*. ¿De dónde sale la primera línea? Las respuestas están en el libro, pero repartidas por él. La pregunta sobre la ausencia está en el Capítulo 5; la de las magnitudes, en el Capítulo 8; la regla que distingue una acción de un proceso, en el Capítulo 17; el patrón para una máquina compartida, en el Capítulo 7 y otra vez, con reloj, en el Capítulo 10. Nadie modela con diecisiete capítulos abiertos a la vez.

Este capítulo reúne el método MODEL en un solo sitio. No enseña PDDL nuevo. Recoge lo que cada capítulo añadió a los cinco pasos, lo ordena y lo conecta: las preguntas, en cinco listas de comprobación; una guía que va de lo que necesita un problema a la característica que lo proporciona, con lo que cuesta esa característica; un índice de los patrones y otro de los antipatrones de todo el libro; y el procedimiento de depuración del Capítulo 7, ampliado a los números, el tiempo y los sistemas híbridos. Añade también dos cosas que ninguna característica podía enseñar por sí sola: cómo hacer crecer un modelo por versiones, de clásico a PDDL+, sin perder lo que ya estaba bien, y cómo documentar un modelo para que otra persona pueda usarlo.

El capítulo empieza con el método en una página y las cinco listas de comprobación. Vienen luego la guía de decisión, los dos catálogos y el procedimiento de depuración. La segunda mitad es un ejemplo resuelto en un dominio nuevo en el libro, una lavandería, llevado a través de cinco versiones, cada una de ellas un fichero probado. El capítulo termina con las plantillas para documentar un modelo, una última comparativa de alternativas y unos ejercicios que ya no son ejercicios, sino proyectos.

19.1 El método en una página 436
19.2 Las cinco listas de comprobación 438
19.3 Qué característica para qué necesidad 441
Lo que cuesta cada característica 443
19.4 El catálogo de patrones 444
19.5 El catálogo de antipatrones 446
19.6 Depurar, más allá de lo clásico 448
Primero, leer el veredicto 448
Sin plan, con demostración 449
Sin plan, sin demostración 449
Un plan erróneo, y uno pobre 450
19.7 Hacer crecer un modelo por versiones 451
Versión 1: clásica 452
Versión 2: números . . . 454
Versión 3: tiempo 456
Versión 4: deseos 458
Versión 5: procesos . . . 459
Cinco versiones, una al lado de otra 462
19.8 Documentar un modelo 462
19.9 El método MODEL: el método completo 464
Alternativas comparadas 464
19.10 Errores comunes 466
19.11 Resumen 467
19.12 Lecturas complementarias 468
19.13 Ejercicios 468

19.1. El método en una página

Los cinco pasos no han cambiado desde el Capítulo 1. Lo que ha crecido es la lista de cosas que hay que preguntarse en cada uno. La Tabla 19.1 enuncia los pasos una vez más, con el artefacto que deja cada uno, y la Figura 19.1 muestra qué capítulos añadieron preguntas a qué paso.

Paso	Pregunta	Artefacto
M Mapear la situación	¿Cuál es el problema, quién actúa, qué cuenta como resuelto, qué se deja fuera?	La *ficha del modelo*: Historia, Agentes, Éxito, Alcance, Supuestos; más adelante también la métrica, los deseos, las reglas y el calendario del mundo
O Objetos y estado	¿Qué cosas existen, y qué debe saber de ellas el planificador para decidir?	La *tabla de estado*: cada propiedad, de qué trata, si es estática o dinámica, cuántos valores tiene, cómo se codifica
D Dinámica	¿Qué se puede hacer o puede ocurrir, cuándo, y qué cambia?	La *tabla de acciones*: cada acción, proceso y evento; sus parámetros, cuándo se aplica, qué cambia, cuánto dura
E Ensayar el modelo	¿Hace el planificador con él lo que el mundo permitiría, y nada más?	La *batería de pruebas*: un problema trivial, uno típico, uno sin plan, uno que tienta al planificador a hacer trampa; cada plan, leído y validado
L Lectura crítica	¿Qué otra codificación serviría, y cuál es mejor aquí?	La *comparativa*: dos o tres alternativas según los cinco criterios, y una decisión con su razón

Tabla 19.1.: El método MODEL: cinco pasos, cinco preguntas, cinco artefactos.

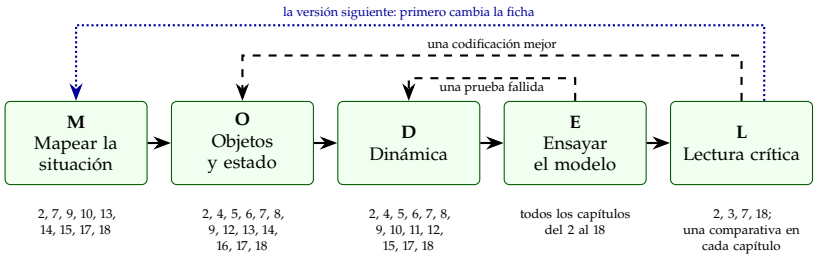


Figura 19.1.: Los cinco pasos, los dos bucles del Capítulo 1 (discontinuos) y los capítulos que añadieron preguntas a cada paso. La flecha punteada es el bucle que añade este capítulo: un modelo crece por versiones, y cada versión vuelve a empezar en el paso M.

En la figura merece la pena fijarse en tres cosas. El paso D y el paso O cargan con la mayor parte del lenguaje: casi toda característica de PDDL es una manera nueva de describir una propiedad o un cambio. El paso E creció en todos los capítulos, porque cada característica trajo su propia manera de equivocarse. Y el paso M, que no escribe PDDL, lo tocaron nueve capítulos: las métricas, el tiempo, los calendarios, los deseos, las reglas y un mundo que se mueve solo se deciden todos en la ficha del modelo, antes de ser características.

Los criterios del paso L son los mismos cinco, en el mismo orden (Capítulo 1). Lo que el libro añadió es cómo ponerles cifras, y la Tabla 19.2 lo recoge.

Criterio	Cómo se juzga
Fidelidad	Ningún plan imposible: las pruebas de la trampa, y planes erróneos escritos a mano que `kairos validate` rechaza (Capítulo 2). Ninguna solución perdida: planes razonables escritos a mano que acepta (Capítulo 11)
Tamaño	Los átomos y las acciones instanciadas de la línea `[ground]` de `kairos -v`, y cómo crecen con el problema (Capítulo 3)
Guía	Estados expandidos hasta el primer plan y hasta el último; el primer plan frente al mejor; las variables de estado y los grupos mutex de la línea `[sas]` (Capítulo 3, Capítulo 7); el motor que tuvo que usar el planificador (Capítulo 11)
Claridad	Si el fichero se lee como la historia, y si el plan también; cuántos sitios cambian cuando cambia una regla
Portabilidad	Los requisitos del fichero frente a la tabla de planificadores del Apéndice B

Tabla 19.2.: Los cinco criterios del paso L y cómo los mide el libro.

El método son cinco pasos y dos bucles. Este capítulo añade un tercer bucle: un modelo crece por versiones, y cada versión vuelve a empezar en la ficha del modelo.

19.2. Las cinco listas de comprobación

Una **lista de comprobación** es una lista de preguntas que se responden por escrito, en orden, cada vez. Las cinco que siguen contienen todas las preguntas que los Capítulos 2 a 18 añadieron al método, con el capítulo en el que se planteó cada pregunta por primera vez y donde se explica su respuesta. Están pensadas para tenerlas junto al teclado. Nadie necesita todas las preguntas en todos los modelos: un modelo clásico se salta la mayoría con un «no» rápido, y cada «no» es una decisión que la ficha del modelo debería anotar en el *Alcance*.

Lista M: Mapear la situación (la ficha del modelo)

- **Historia.** ¿Puedo escribir las cinco líneas de la ficha antes de nada de PDDL? ¿Qué reglas del mundo enuncia la ficha (Capítulo 2)?
- **Agentes.** ¿Quién actúa, y quién es solo parte del escenario (Capítulo 7)? ¿Quién puede actuar al mismo tiempo que quién (Capítulo 10)?
- **Éxito.** ¿Qué cuenta como resuelto? ¿Qué objetivos son obligatorios y cuáles son deseos, cuánto vale cada deseo, y en qué unidad (Capítulo 14)? ¿Cuál es el plan cuando no se concede nada (Capítulo 14)?
- **Calidad.** ¿Qué hace que un plan sea mejor que otro, y en qué unidad? ¿Es un solo número, y si no lo es, quién fija los pesos (Capítulo 9)? ¿Importa *cuándo* se alcanza el objetivo (Capítulo 10)?
- **Alcance.** ¿Qué se deja fuera, y quién se encarga de ello (Capítulo 7)?
- **Supuestos.** ¿Cuáles de los supuestos clásicos se cumplen (Capítulo 1, Capítulo 7)? ¿Cambia el mundo en momentos conocidos, haga lo que haga el agente (Capítulo 13)? ¿Cambia cuando el agente no hace nada; esperar cuesta o consigue algo (Capítulo 17)?
- **Reglas.** Se enumeran aparte de los objetivos. Para cada una: ¿trata de una acción o de todo el plan; mira un estado o la historia; de quién es la regla, y puede ceder (Capítulo 15)? ¿Hay algo que no deba ocurrir nunca, en ningún instante, y está prohibido o solo tiene consecuencias (Capítulo 18)?

Lista O: Objetos y estado (la tabla de estado)

- **Clases.** ¿Qué clases de cosas hay? ¿Hay una jerarquía? ¿Pertenece un objeto a todos los problemas, y es por tanto una constante (Capítulo 4)?
- Para cada propiedad:
  - ¿Es estática o dinámica (Capítulo 2)?
  - ¿Es de sí o no, o tiene varios valores (Capítulo 5)? ¿Cuántos valores por objeto: exactamente uno, a lo sumo uno, muchos (Capítulo 16)? Cada «uno» escrito como predicado es un invariante que hay que probar (Capítulo 7).
  - ¿Trata de una ausencia, y necesita un hecho propio? ¿Se solapan dos predicados, y quién los mantiene acompasados (Capítulo 2, Capítulo 5)?
  - ¿Se puede deducir de otra propiedad, de modo que no haga falta almacenarla (Capítulo 6)? ¿Es una consecuen-

cia de otras que una regla puede enunciar, mirando un número fijo de hechos o siguiendo una cadena (Capítulo 13)?
    - ¿Es una magnitud? ¿Solo importa un umbral? ¿Está acotada, y qué la mantiene dentro de sus cotas (Capítulo 8)? ¿Es una medida de la métrica, un contador que empieza en cero (Capítulo 9)?
    - ¿Cambia mientras se ejecuta una acción, y a qué ritmo (Capítulo 12)? ¿Cambia por sí sola: qué hecho enciende y apaga el cambio, y qué número mide cuánto ha avanzado (Capítulo 17)? ¿Su ritmo es una constante, otra magnitud que cambia o la propia magnitud (Capítulo 18)?
- ¿Trata un deseo o una regla del camino, sin nada en el estado que lo registre? Entonces un hecho debe registrarlo (Capítulo 14).

Lista D: Dinámica (la tabla de acciones)

- **¿Quién causa cada cambio?** El agente, por una decisión: una acción. El mundo, mientras se cumple una condición: un proceso. El mundo, en el instante en que una condición se vuelve cierta: un evento (Capítulo 17). La prueba: imagina que el agente se queda dormido.
- Para cada acción:
    - ¿Cuándo se aplica? ¿Hay una condición sobre una ausencia, sobre algún objeto, sobre todos los objetos; hay varias maneras de satisfacerla (Capítulo 5)? ¿Pueden dos parámetros ser el mismo objeto (Capítulo 4)? ¿Hace falta cada parámetro (Capítulo 7)?
    - ¿Dice una precondición lo que la acción *necesita*, o lo que alguien *quiere* del plan? Lo segundo es una regla (Capítulo 15).
    - ¿Qué cambia? Para cada propiedad dinámica: la hace cierta, la hace falsa o la deja como está; todo lo que cambia y nada de lo que no cambia (Capítulo 2). ¿Depende lo que cambia de la situación; cambia muchos objetos a la vez (Capítulo 6)?
    - Para cada magnitud: ¿necesita la acción una parte de ella, la cambia, y qué suma a cada contador, ceros incluidos (Capítulo 8, Capítulo 9)?
    - ¿Se puede deshacer? ¿Es una decisión o varias (Capítulo 7)?
    - ¿Importa su duración, y de qué depende? ¿Qué debe cumplirse al inicio, durante toda la acción y al final? ¿Cuándo ocurre cada efecto? ¿Qué puede ejecutarse en paralelo con ella (Capítulo 10)?
    - ¿Debe solaparse con otra acción, y con cuánta holgura? ¿Es su duración una decisión, y depende algo de ella (Capítulo 11)?
    - ¿Lee alguien una magnitud mientras la acción la cambia; puede otra acción cambiarla al mismo tiempo (Capítulo 12)?
- Para cada proceso: ¿está en su condición todo lo que debe ser cierto para que el cambio continúe? ¿Qué procesos tocan la misma magnitud, y cuál es la suma de sus ritmos (Capítulo 17, Capítulo 18)?

- Para cada evento: ¿qué hace falsa su condición una vez que se ha disparado? ¿Qué eventos pueden disparar a cuáles, y dónde pasa el tiempo en cada ciclo (Capítulo 17, Capítulo 18)?

Lista E: Ensayar el modelo (la batería de pruebas)

- **Antes de ejecutar.** Estima las acciones instanciadas y los estados alcanzables, y compáralos con la línea `[ground]` (Capítulo 3). En un modelo híbrido, calcula con lápiz adónde va cada magnitud si no se hace nada (Capítulo 18).
- **La rutina.** `kairos check` con el dominio y el problema, sin ningún aviso sin leer; resolver; leer el plan sobre el estado; `kairos validate` (Capítulo 2).
- **Las cuatro pruebas.** Un problema trivial, uno típico, uno que no debe tener plan, uno que tienta al planificador a hacer trampa. Conoce la longitud, el coste o el *makespan* esperados antes de ejecutar (Capítulo 2, Capítulo 10).
- **Las trampas, a mano.** Cada atajo, como un fichero de plan que el validador debe rechazar, y por la razón esperada (Capítulo 7).
- **Una prueba para cada cosa que promete el modelo:**
  - cada par de parámetros que deben ser distintos (Capítulo 4); cada estado definido por una ausencia; cada objetivo que podría cumplirse por una razón tonta (Capítulo 5);
  - cada efecto condicional, con su condición cierta y falsa (Capítulo 6);
  - cada magnitud: un problema que necesita más de lo que hay, un objetivo al otro lado de una cota (Capítulo 8); dos planes de coste conocido (Capítulo 9);
  - cada agente en dos sitios a la vez; cada solapamiento de un plan, leído como una pregunta (Capítulo 10); cada acción envolvente con un contenido que no cabe (Capítulo 11); cada límite, con un plan corto que lo rompe a mitad (Capítulo 12); cada ventana, con una acción más larga que la ventana (Capítulo 13);
  - cada deseo: el plan sin pesos, y la prueba del precio (Capítulo 14); cada regla: un plan que la rompe (Capítulo 15);
  - cada evento, disparado al menos una vez; cada proceso: ¿hay algún tramo de tiempo en el que funcione gratis, o no funcione (Capítulo 17, Capítulo 18)?
- **Cuando el planificador no encuentra nada.** ¿Es el veredicto una demostración? Si no lo es, escribe el plan a mano y valídalo; da a las pruebas «sin plan» un horizonte corto (Capítulo 11, Capítulo 18).
- **Si algo falla,** el procedimiento de la Sección 19.6.

Lista L: Lectura crítica (la comparativa)

- ¿Qué otras codificaciones existen? Todo patrón tiene un rival (Sección 19.4), y toda característica tiene una vecina más barata (Sección 19.3).
- ¿Cuál es el nivel más bajo (clásico, números, acciones durativas, PDDL+) que no miente sobre aquello de lo que depende el problema (Capítulo 18)?
- Compara dos o tres, escritas y ejecutadas sobre el mismo

problema, según la Fidelidad, el Tamaño, la Guía, la Claridad y la Portabilidad, con cifras (Capítulo 3).
- Si la decisión depende de una pregunta que nadie hizo, pertenece al paso M: vuelve atrás y hazla (Capítulo 7).
- Escribe la decisión y su razón donde la encontrará el próximo lector: en la ficha del modelo.

Cinco listas de comprobación contienen todas las preguntas del libro. Un «no» rápido a una pregunta también es una respuesta, y va a la línea de Alcance de la ficha.

19.3. Qué característica para qué necesidad

Una característica de PDDL es la respuesta a una necesidad del problema, y las listas de comprobación encuentran las necesidades. Esta sección va en sentido contrario: de la necesidad a la característica, al requisito que la declara y al capítulo que la enseña, y después a lo que cuesta la característica.

La Figura 19.2 es la guía en forma de diagrama de flujo. Se lee desde arriba, y no tiene salida por la derecha: un «sí» añade una característica al modelo y la lectura sigue con la pregunta siguiente. El orden es el del libro y también el del coste. El consejo del Capítulo 18 para los sistemas híbridos vale para todo el lenguaje: *quédate en el nivel más bajo que no mienta sobre aquello de lo que depende el problema*.

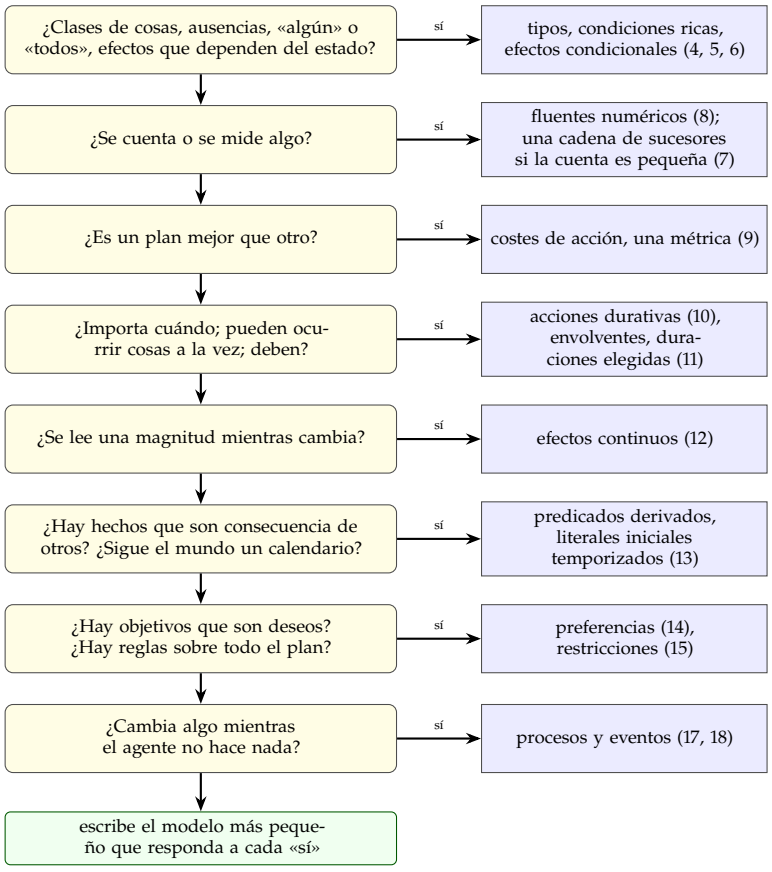


Figura 19.2.: La guía de decisión. Cada pregunta se le hace a la ficha del modelo; cada «sí» añade la característica de su derecha (entre paréntesis, los números de capítulo) y la lectura sigue hacia abajo. Cuanto más abajo está una característica, menos planificadores la aceptan y más difícil es la búsqueda.

Las Tablas 19.3 y 19.4 son la misma guía en detalle: una fila por cada necesidad que ha aparecido en el libro. El requisito `:adl` declara de una vez todos los de la Tabla 19.3 que están por encima de la línea (Capítulo 6).

El problema necesita	Característica	Requisito	Cap.
clases de cosas, algunas tratadas igual	tipos, una jerarquía, `either`	`:typing`	4
un objeto que nombran las reglas	una constante	ninguno	4
dos parámetros que deben ser distintos	`(not (= ?a ?b))`	`:equality`	4
una condición sobre una ausencia	`not`	`:negative-preconditions`	5
varias maneras de satisfacer una condición	`or`, `imply`	`:disjunctive-preconditions`	5
una condición sobre algún objeto	`exists`	`:existential-preconditions`	5
una condición o un objetivo sobre todos los objetos	`forall`	`:universal-preconditions`	5
un resultado que depende del estado; un cambio en muchos objetos	`when`, `forall` en los efectos	`:conditional-effects`	6
una magnitud que se suma, se gasta, se compara	funciones, `increase`, `decrease`, `assign`	`:numeric-fluents`	8
un plan mejor, como suma de costes fijos	`(total-cost)`, `:metric`	`:action-costs`	9
un plan mejor, según una medida que depende del estado	`:metric` sobre fluentes	`:numeric-fluents`	9
una propiedad con exactamente un valor	un fluente de objeto	`:object-fluents`	16

Tabla 19.3.: De la necesidad a la característica: el estado, las condiciones y los efectos.

El problema necesita	Característica	Requisito	Cap.
acciones que duran y pueden solaparse; el instante de terminar	acciones durativas, `(total-time)`	`:durative-actions`	10
algo que solo se cumple mientras se ejecuta una acción	una acción envolvente (concurrencia requerida)	sin indicador propio	11
una duración que es una decisión	cotas sobre `?duration`	`:duration-inequalities`	11
una magnitud que alguien lee mientras cambia	`#t` en una acción durativa	`:continuous-effects`	12
un hecho que se sigue de otros, a cualquier profundidad	`:derived`	`:derived-predicates`	13
un mundo con calendario: horarios, ventanas, plazos	`(at t (fact))` en `:init`	`:timed-initial-literals`	13
números que cambian en momentos conocidos	`(at t (= (f) v))`	`:timed-initial-fluents`	13
objetivos que son deseos, con un precio	`preference`, `is-violated`	`:preferences`	14
una regla sobre todo el plan	`always`, `sometime`, `within` y los demás operadores	`:constraints`	15
un cambio mientras se cumple una condición, actúe quien actúe	`:process`	`:time`	17
un cambio en el instante en que una condición se vuelve cierta	`:event`	`:time`	17
una condición que ningún instante de ningún plan puede romper	`:constraint` en el dominio	`:time`	18

Tabla 19.4.: De la necesidad a la característica: el tiempo, las preferencias y las restricciones, y el mundo que cambia por sí solo.

Lo que cuesta cada característica

Ninguna característica sale gratis. La Tabla 19.5 da, para cada familia, lo que compra en Fidelidad y lo que cobra en los otros cuatro criterios. La columna Portabilidad cuenta, entre los ocho planificadores que compara la documentación de Kairos (Martín Rico et al., 2026) (la tabla del Apéndice B), los que devolvieron un plan válido para un problema pequeño con la característica, con diez segundos cada uno. Una cuenta dice que un planificador lee la construcción y planifica con ella, no lo bien que lo hace con problemas grandes.

Tabla 19.5.: Lo que compra y lo que cuesta cada característica. Portabilidad: planificadores, de ocho, que devolvieron un plan válido para un problema pequeño con la característica.

Característica	Compra (Fidelidad)	Cobra (Tamaño, Guía, Claridad)	Portabilidad
Tipos, constantes, igualdad	mezclar clases se vuelve un error de `check`	nada: el mismo problema instanciado	7 de 8; `either`, 5
Negación, `or`	ausencias y alternativas dichas directamente	`or` puede duplicar acciones instanciadas	5
`forall`, `exists`	reglas que siguen a los objetos	se expanden sobre los objetos del tipo	5
Efectos condicionales	una acción para lo que en el mundo es una sola cosa	los planificadores que no los tienen los compilan en más acciones o en planes más largos	3
Predicados derivados	las consecuencias no pueden desacompasarse	las reglas se instancian y se evalúan en cada estado	3
Fluentes numéricos	las cantidades, como datos	el espacio de estados deja de ser finito	5
Costes de acción; métricas sobre fluentes	qué plan es mejor	el mejor plan se encuentra tarde y se demuestra más tarde	4; 4
Acciones durativas	solapamiento, *makespan*	búsqueda sobre horarios; los planes son válidos, no los más cortos	5
Concurrencia requerida	lo que solo existe mientras se ejecuta una acción	búsqueda sobre inicios y finales; «no plan found» sin demostración	4
Duraciones elegidas	«todo lo que haga falta»	las duraciones salen en sus cotas	3
Efectos continuos	un número que se lee mientras se mueve	los instantes se vuelven decisiones	2
Literales temporizados	el calendario del mundo	las acciones deben caber en las ventanas	3
Preferencias	sobresuscripción, deseos con precio	un problema de optimización: el primer plan puede ser vacío	4
Restricciones	reglas enunciadas una vez, sobre todos los estados	un monitor en cada estado de la búsqueda	2
Fluentes de objeto	las propiedades de un solo valor no pueden romperse	nada: los planificadores construyen esas variables de todos modos	1
Procesos, eventos	un mundo que no espera	búsqueda en pasos de tiempo; hay que reproducir los planes para leerlos	1

Dos observaciones sobre la última columna. Los planificadores que faltan no son siempre los mismos: POPF y OPTIC, dos planificadores temporales, no tienen precondiciones negativas, disyunciones ni efectos condicionales (Capítulo 5, Capítulo 6), y necesitan las condiciones y los efectos numéricos dentro de `(and ...)` (Capítulo 8); LAMA, un

planificador clásico, no tiene fluentes numéricos. Así que la versión portable de un modelo temporal no es la versión portable de uno clásico. Y a menudo merece la pena cambiar una característica de una fila baja por un patrón de una fila alta: predicados complementarios en lugar de la negación, una cadena de sucesores en lugar de un número pequeño, bloques en lugar de una duración elegida, un autómata en lugar de una restricción. La sección siguiente es el índice de esos cambios.

Una característica responde a una necesidad y cobra por ello. Añade características en el orden del diagrama de flujo, y para cuando el modelo ya no mienta sobre aquello de lo que depende el problema.

19.4. El catálogo de patrones

El Capítulo 7 definió un **patrón** como una manera con nombre de codificar algo que muchos problemas tienen en común, y dio once para la planificación clásica. Los capítulos anteriores y posteriores nombraron otros a medida que los encontraban. Esta sección es el índice de todos ellos, con los nombres que usan sus capítulos, cada uno con la necesidad a la que responde y su rival: la otra codificación que el paso L debería considerar. Las definiciones, los ejemplos probados y los costes están en los capítulos; aquí está dónde buscarlos.

Patrón	Úsalo cuando	Rival	Cap.
Tabla estática	una regla puede variar de un problema a otro	la regla escrita en una acción	7
Hechos espejo	una relación se cumple en los dos sentidos	un solo hecho y un `or` en la acción	7
Invariante de a lo sumo uno	una propiedad tiene un valor	un fluente de objeto	7, 16
Cadena de sucesores	una magnitud pequeña; solo importa la cantidad	huecos; un fluente numérico	7, 8
Huecos	las unidades de capacidad son lugares distintos	una cadena de sucesores	7
Testigo de fase	las etapas no deben mezclarse	una precondición sobre lo que ya está hecho	7
Testigo de turno	los agentes, o el agente y el mundo, se alternan	acciones que dan un paso de cada uno	7
Cerrojo de recurso	un usuario cada vez	una acción envolvente, cuando se modela el tiempo	7, 11
Par inverso	el mundo puede deshacerlo	una acción sin vuelta atrás	7
Acción sin vuelta atrás	el mundo no puede deshacerlo	un par inverso	7
Acción fusionada	no hay ninguna decisión entre los pasos	acciones separadas	7
Predicado de tipo	una clase que cambia, o que atraviesa la jerarquía	un tipo	2, 4
Predicado complementario	una ausencia sin `not`; tres o más valores con nombre	una precondición negativa	5
Una acción por caso	pocos casos fijos; el plan debería nombrar el caso	un efecto condicional	6
Acción auxiliar	muchos casos y un planificador sin efectos condicionales	un efecto condicional	6
Deducido, no almacenado	un hecho se puede deducir de otro	un hecho almacenado que mantienen los efectos, o un predicado derivado	6, 13

Tabla 19.6.: Los patrones de la planificación clásica: el catálogo del Capítulo 7 (por encima de la línea) y las codificaciones que compararon los Capítulos 2 a 6 (por debajo).

Patrón	Úsalo cuando	Rival	Cap.
Magnitud acotada	una cantidad se gasta y se repone: una precondición antes de cada disminución, un tope en cada aumento	un hecho de umbral; una cadena de sucesores	8
Fluente contador	una medida de la métrica: empieza en cero, ninguna precondición la lee	costes de acción	9
Límite estricto	algo no debe superarse	un término de la métrica	9
Suma ponderada	varias medidas, una métrica	un límite sobre todas menos una	9
Testigo	un agente o una herramienta hace una sola cosa a la vez: se borra al inicio, se añade al final	ninguno: sin él las acciones se solapan	10
Invariante	algo debe cumplirse todo el tiempo: `over all`	una condición al inicio, cuando basta un instante	10
Tomar al inicio, dar al final	todo efecto de una acción durativa	la excepción es la acción envolvente	10
Acción envolvente y contenido	algo existe solo mientras se ejecuta una acción, y termina por sí solo	un cerrojo, que cierra el objetivo	11
Holgura	dos acciones deben solaparse: una contiene a la otra con margen de sobra	la simultaneidad exacta, que los planificadores no buscan	11
Duración elegida	«¿durante cuánto tiempo?» es una decisión	una duración fija; bloques	11
Bloques	una duración elegida en un planificador que no la tiene	una duración elegida	11
Modelo discreto prudente	una magnitud cambia durante una acción y nadie la lee mientras tanto: consumir al inicio, producir al final	un efecto continuo	12
Predicado derivado para un gemelo	un hecho redundante que se usa en muchas acciones	un gemelo mantenido por efectos	13
Ventana	un hecho cierto entre dos literales temporizados, necesario `over all`	una restricción `hold-during`	13, 15
Plazo	un hecho auxiliar que borra un literal temporizado, necesario `at end`	una restricción `within`	13, 15
Objetivo blando	más objetivos de los que se pueden cumplir	un objetivo estricto	14
Preferencia en una precondición	un deseo sobre el camino, que se paga cada vez	una precondición estricta; un coste de acción	14
Primero la restricción	cualquier regla sobre todo el plan: la restricción es la especificación	precondiciones en todas las acciones; un monitor	15
Monitor	una regla sobre la historia en un planificador sin restricciones: un autómata en predicados adicionales	una restricción	15
Fluente de objeto	exactamente un valor, o a lo sumo uno	un predicado con un invariante	16
Inicio-proceso-parada	una actividad cuyo final depende de cómo evoluciona el mundo	una acción durativa	17
Evento como dispositivo	una parte del sistema reacciona por sí sola: un termostato, un interruptor de flotador	una acción del agente	18
Histéresis	un evento enciende y apaga algo alrededor de un umbral	un solo umbral, que nunca se estabiliza	18
Restricción global	un fallo está prohibido en todo plan	un evento que marca el fallo y un objetivo que rechaza la marca	18

Tabla 19.7.: Los patrones de los números, el tiempo, las preferencias y las restricciones, y los sistemas híbridos.

Los capítulos posteriores al Capítulo 7 no siempre presentaron sus codificaciones como patrones; la segunda tabla usa las palabras de cada capítulo. Las dos tablas son el vocabulario de los pasos O y D. Cuando se escribe una fila de la tabla de estado, la pregunta es qué patrón es: una tabla estática, un invariante de a lo sumo uno, una magnitud acotada. Cuando se escribe una acción: un testigo, un par inverso, una acción envolvente. Y en el paso L la tercera columna de las tablas es la lista de alternativas que merecen una columna de la comparativa.

Cuarenta codificaciones con nombre, cada una con un rival. El nombre dice qué es una fila de la tabla de estado o de la tabla de acciones; el rival es aquello con lo que la compara el paso L.

19.5. El catálogo de antipatrones

Un **antipatrón** es un error lo bastante común como para merecer un nombre (Capítulo 7). Los que más importan tienen algo en común: el fichero es PDDL válido y `kairos check` dice `valid, plannable`. Se reconocen por lo que hace el planificador, así que las tablas que siguen están ordenadas por síntoma. Los errores de forma (un nombre mal escrito, un paréntesis que falta, un requisito olvidado) no están aquí: `check` los señala, y la regla para ellos es la del Capítulo 2, ningún aviso sin leer.

Tabla 19.8.: Antipatrones de los modelos clásicos, por síntoma.

Síntoma	Antipatrón	Cap.
Un plan demasiado corto; algo está en dos sitios; menos grupos mutex que propiedades con un solo valor	Borrado olvidado	2, 7
Un plan que hace lo que la ficha prohíbe	Una precondición que falta	2
Un plan de coste 0; un plan que se salta parte del trabajo	Objetivo en el estado inicial	2, 7
La prueba trivial pasa, la típica no tiene plan	Gemelos desacompasados	5, 7
Muchas más acciones instanciadas de las estimadas; argumentos sin sentido	Parámetro ocioso	3, 7
Un objeto actúa sobre sí mismo, o hace dos papeles	Confiar en que variables distintas son objetos distintos	4
Un objetivo con «todos» o «no» se cumple por una razón tonta: no hay objetos, un artículo que no está en ninguna habitación	Un `forall` que se cumple sin objetos	5
Un plan que hace daño por el camino y aun así llega al objetivo	Un daño que el objetivo no vigila	6
Un plan sin las acciones que pasan el testigo	Dos testigos	7
Planes más largos de lo esperado; una cuenta que nunca pasa de cierto valor	Una cadena con un eslabón roto	7
Un robot que entra en una habitación y no puede salir	La mitad de los hechos espejo	7

Síntoma	Antipatrón	Cap.
Planes demasiado buenos; una magnitud por debajo de cero en la traza	El consumo sin la precondición	8
No hay plan tras un puñado de estados; un plan que evita algo sin razón visible	Un valor inicial olvidado	8, 16
No hay plan para un problema que resolvería un niño, con un objetivo `(= ...)` sobre un número calculado	Igualdad con fracciones	8
Una acción usada todo lo posible; un coste menor que el calculado a mano	Una acción que olvidó su coste	9
Un agente empieza algo y se va; un *makespan* más corto que la suma esperada	Comprobar al inicio lo que debe cumplirse todo el tiempo	10
Una acción se solapa con todas las demás, o el agente no vuelve a actuar	Un testigo olvidado en una acción	10
Un modelo que se resolvía al momento deja de resolverse; `-v` indica el motor M3 para una acción que nadie veía como envolvente	Una acción envolvente por accidente	11
Un mismo objeto sujeta la puerta y la cruza	El agente de la acción envolvente es también su contenido	11
Una carga de la duración más corta permitida, o que pasa de la capacidad	Una duración elegida de la que nada depende	11
Un plan que el validador acepta mientras un número cruza su límite a mitad	Un límite comprobado solo al principio	12
Funciona en mapas cortos y falla en el primer pasillo largo	Una regla que solo mira un paso más allá	13
Una acción antes de la hora de apertura	El hecho ya es cierto	13
Una acción que termina dentro de una ventana cerrada	Una ventana comprobada solo al inicio	13

Tabla 19.9.: Antipatrones de los modelos con números y tiempo, por síntoma.

Síntoma	Antipatrón	Cap.
Un plan vacío de coste 0 para un problema lleno de deseos	Preferencias sin métrica	14
Un plan que hace menos de lo esperado y es `proven optimal`	Pesos en la unidad equivocada	14
Una regla de la historia que se rompe siempre que sale más barato	Una preferencia en una precondición no es una precondición	14
Una restricción rota antes de la primera acción	Olvidar que el estado inicial cuenta	15
Un plan que termina antes de que se abra la ventana de una restricción	Una ventana a la que el plan nunca llega	15
Un plazo que no restringe nada	Pasos tomados por segundos	15
`unsolvable` en un problema que parecía fácil	Una regla más fuerte que el mundo	15
El robot no empieza en ningún sitio; `unsolvable` tras un estado	Las funciones no son predicados	16
Un plan para un problema que no debería tenerlo; cosas que se encienden y se apagan en momentos raros	Un proceso que funciona cuando no debe	17
No hay plan cuando el plan debe pasar por un evento; el validador dice `events keep firing`	Un evento que no se desactiva a sí mismo; dos eventos que se activan el uno al otro	17, 18
Un planificador lento, y herramientas que no se ponen de acuerdo	Un termostato con un solo umbral	18
`no plan found` tras pocos estados, y un plan a mano que es válido	Una ventana más estrecha que un paso	18
Un plan al milisegundo que el validador rechaza por 0,001	Igualdad sobre una cantidad que fluye	18
Un plan válido que espera y carga más de lo necesario	Esperar el plan más corto	18
Un rescate que el modelo prohíbe, o un límite que un problema olvida	Una restricción donde hacía falta un evento, y al revés	18

Tabla 19.10.: Antipatrones de los modelos con preferencias, restricciones, fluentes de objeto, procesos y eventos, por síntoma.

Si se lee de arriba abajo la primera columna, los síntomas se agrupan en cuatro familias, y la familia ya dice dónde mirar.

- **Demasiado fácil**: un plan demasiado corto, demasiado barato o demasiado pronto. El modelo permite demasiado: falta una precondición, un borrado, un testigo, un invariante o un coste.
- **Demasiado difícil**: no hay plan, o el plan da rodeos extraños. El modelo permite demasiado poco: un hecho que nadie produce, un valor que nadie dio, una regla demasiado fuerte.
- **Demasiado grande**: más acciones instanciadas o estados de los estimados, una búsqueda que no termina. Hay un parámetro, una simetría o un horizonte fuera de control.
- **No es lo que se pidió**: el plan es correcto para el modelo, y el modelo responde a otra pregunta. La métrica, los pesos o el objetivo dicen algo que la ficha no decía.

Los antipatrones pasan la comprobación y se detectan por su síntoma: un plan demasiado fácil, demasiado difícil, demasiado grande, o correcto para la pregunta equivocada.

19.6. Depurar, más allá de lo clásico

El procedimiento del Capítulo 7 tenía dos ramas. Sin plan: leer la nota del planificador, relajar el objetivo término a término, quitar precondiciones hasta que aparezca un plan y preguntar qué acción debería haber hecho cierto el hecho que falta. Plan erróneo: buscar el primer paso que el mundo rechaza, y luego la precondición o el borrado que falta. Las dos ramas se quedan como están. Los números, el tiempo y los procesos añaden tres cosas: más veredictos, más maneras de que un plan sea erróneo y un planificador que puede pasar por alto planes que existen.

Primero, leer el veredicto

En la planificación clásica, una ejecución terminaba con un plan o con `unsolvable`. La Tabla 19.11 enumera los veredictos que han aparecido en el libro desde entonces, cada uno con un fichero de este capítulo que lo produce. Solo uno de ellos es una demostración de que no existe plan.

Veredicto	Código	Qué dice, y un ejemplo
un plan	0	El plan es válido para el modelo. `solved` no es `proven optimal` (Capítulo 9)
`unsolvable`	10	Una demostración: no existe plan (Capítulo 3). `laundry-v1-p03`
`no plan found`	14	La búsqueda terminó sin plan y sin demostración (Capítulo 11, Capítulo 12, Capítulo 17). `laundry-v5-p03`
`timeout`	11	Nada: la búsqueda seguía en marcha (Capítulo 3). `causeway-p04`, entre las soluciones
`invalid`	2	Los ficheros no dicen lo que se quería decir. Lee el primer error (Capítulo 1)

Tabla 19.11.: Los veredictos de Kairos y sus códigos de salida. Los guiones que ejecutan una batería de pruebas (`tests.txt`) indican el código que espera cada prueba.

La Figura 19.3 es el procedimiento completo. Las cajas grises son las del Capítulo 7; las demás son nuevas.

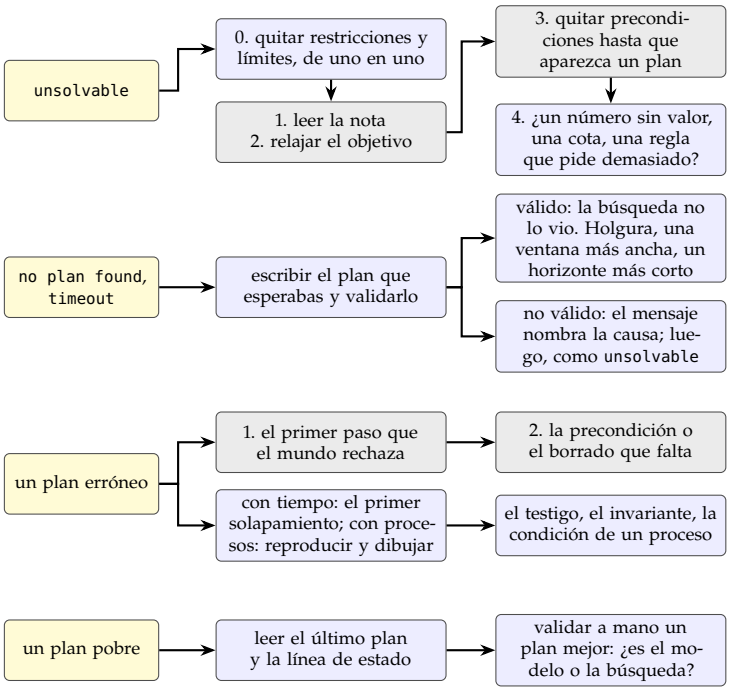


Figura 19.3.: El procedimiento de depuración del paso E, ampliado. En gris, los pasos del Capítulo 7. En azul, lo que añaden los números, el tiempo, las restricciones y los procesos.

Sin plan, con demostración

El veredicto `unsolvable` se trata como en el Capítulo 7, con un paso antes y otro después.

0. Quita lo que elimina planes. Una restricción, un límite estricto sobre un presupuesto, un plazo y una restricción global no intervienen en ninguna acción: eliminan planes enteros. Quítalos de uno en uno, en una copia del problema, hasta que aparezca un plan. Después valida ese plan frente a cada uno de ellos por separado, para ver cuál rompe y dónde (Capítulo 15). Siguen los pasos 1 a 3, sobre el modelo sin ellos.

4. Pregunta a los números. Cuando la precondición cuya eliminación devuelve un plan es numérica, el hecho «que nadie hace cierto» es un valor. Hay tres causas habituales. Un fluente que lee una acción nunca recibió valor, así que la acción nunca se aplica: la señal es una búsqueda de un puñado de estados, y la corrección, una línea en `:init` (Capítulo 8; lo mismo para un fluente de objeto, Capítulo 16). Una cota es más estrecha que la tarea: suma a mano lo que necesita el plan y compáralo con lo que hay. O una prueba pide igualdad exacta sobre un número que se calcula (Capítulo 8) o que fluye (Capítulo 18).

Sin plan, sin demostración

`no plan found` y `timeout` no dicen que el modelo no tenga plan. Antes de cambiar el modelo, pregunta al validador, que no tiene ninguna búsqueda en la que perderse: escribe en un fichero el plan que esperabas y ejecuta `kairos validate`.

Si el plan *no es válido*, el mensaje nombra la condición que falla y el instante. Es el paso 3 del procedimiento clásico, hecho por ti: el modelo

de verdad no tiene ningún plan de esa forma, y la pregunta vuelve a ser qué acción debería haber hecho cierta la condición.

Si el plan *es válido*, el modelo está bien y la búsqueda no supo encontrarlo. Los capítulos del libro recogen cuatro situaciones en las que eso ocurre con Kairos 1.0.6, y para cada una de ellas una manera de modelar sorteándola. La caja las reúne.

Lo que no encuentra el planificador de este libro

- **Una duración entre sus cotas.** Una duración que elige el planificador sale en una de sus cotas, o en el instante en que terminar la acción hace cierta una condición. Una duración intermedia que ninguna condición fija no se prueba: el mensajero del Capítulo 12 carga durante 8 segundos cuando bastaban 6. *Cómo sortearlo*: haz que algo necesite la duración, o usa bloques (Capítulo 11).
- **La misma acción envolvente usada dos veces.** Los planes que repiten una acción envolvente, con un contenido dentro de cada repetición, están entre los que se le escapan a la búsqueda sobre inicios y finales (Capítulo 11, en las soluciones). *Cómo sortearlo*: lee el plan, escribe a mano el mejor y valídalo.
- **Dos inicios exactamente en el mismo instante.** Los acontecimientos se buscan uno tras otro, así que un modelo cuyos únicos planes empiezan dos acciones en el mismo instante obtiene `no plan found` (Capítulo 11). *Cómo sortearlo*: holgura. Una acción contiene a la otra.
- **Lo que ocurre entre dos pasos.** Con procesos, el tiempo avanza en pasos de 0,1, y los eventos se disparan entre pasos. Una ventana más estrecha que lo que cambia en un paso se pasa de largo, y un evento no rescata la búsqueda (Capítulo 17, Capítulo 18). *Cómo sortearlo*: una ventana al menos tan ancha como el ritmo por 0,1; desigualdades, no igualdad.

Y dos que tienen que ver con los planes buenos más que con los planes. Un plan temporal es un horario válido, no el más corto (Capítulo 11). En PDDL+ la búsqueda minimiza el número de acciones, diga lo que diga la métrica, así que los planes esperan y cargan más de lo necesario (Capítulo 18). En todos los casos el plan que imprime Kairos es válido. Lo que puede faltar es un plan, o uno mejor.

Una búsqueda que se queda sin tiempo en un problema que no debería tener plan suele ser un horizonte que nadie acotó (Capítulo 18): las cosas pueden seguir cambiando durante mucho tiempo, y cada instante es un estado. Da a las pruebas «sin plan» un margen pequeño. La Sección 19.10 muestra una que tarda 0,1 segundos y su versión descuidada, que gasta todo el tiempo que se le da.

Un plan erróneo, y uno pobre

Un plan es erróneo cuando el mundo no lo permitiría, y vale la primera rama del Capítulo 7: léelo frente a la historia y detente en lo primero que el mundo rechace. Lo que cambia es qué es «lo primero».

- **En un plan secuencial**, un paso. Su causa es una precondición que falta o una acción anterior que dejó un hecho erróneo (Capítulo 7); con números, una disminución sin su precondición (Capítulo 8).

- **En un plan temporal**, un solapamiento. Dibuja el cronograma, una fila por agente y por recurso, y pregunta, para cada par de barras que se solapan, si el mundo lo permite (Capítulo 10). La causa es un testigo que una acción no toma, una condición `at start` que debería ser `over all` o un efecto en el extremo equivocado.
- **En un plan con procesos**, un tramo de tiempo. Reproduce el plan, calcula las magnitudes y dibújalas (Capítulo 18). La causa es un proceso cuya condición es demasiado débil, de modo que algo cambia gratis, o un evento que debería haberse disparado.

Un plan es pobre cuando es válido y existe uno mejor. Primero asegúrate de que es el último plan de la ejecución y lee por qué se detuvo la búsqueda (Capítulo 9). Después escribe a mano el plan mejor y valídalo frente al mismo problema: el validador imprime su *makespan* y el valor de la métrica. Si el plan mejor es válido y más barato, el modelo está bien y la búsqueda se quedó corta; dilo en la documentación del modelo y, si importa, pon la presión en el modelo (un plazo, una cota más estrecha). Si el plan mejor no es válido, el modelo no permite lo que creías que permitía, y ese es el hallazgo. La lavandería de la sección siguiente se encuentra con los dos casos.

Tres veredictos no son demostraciones. Cuando el planificador no encuentra nada, o nada bueno, escribe el plan que esperabas y pregunta al validador: él dice si la culpa es del modelo o de la búsqueda.

19.7. Hacer crecer un modelo por versiones

Nadie debería escribir de una sentada un modelo PDDL+ con preferencias. La razón es el procedimiento de depuración que acabamos de leer: funciona cambiando una cosa y mirando qué pasa. Un modelo escrito de una vez tiene cien cosas que cambiaron a la vez. El Capítulo 10 aconsejaba dejar bien la lógica con acciones instantáneas antes de añadir el tiempo, y el Capítulo 18 dijo que sus tres niveles eran «también un camino». Esta sección hace explícito el camino (Figura 19.4).

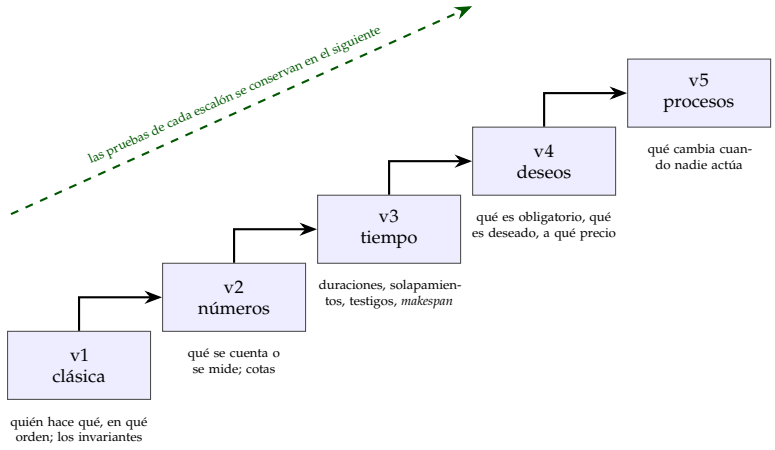


Figura 19.4.: Un modelo crece una dimensión cada vez. Cada escalón conserva las pruebas de los escalones inferiores, traducidas, y añade las suyas.

Reglas para hacer crecer un modelo

1. **Una dimensión por versión.** Primero lo clásico; luego los números; luego el tiempo; luego las preferencias y las restricciones; luego los procesos. Una versión que añade dos son dos versiones.
2. **En verde antes del salto.** Una versión está terminada cuando toda su batería de pruebas da los veredictos esperados. Solo entonces empieza la siguiente.

3. **Primero cambia la ficha.** Cada salto pasa algo de «fuera» a «dentro» en la línea de Alcance, o añade una entrada (una métrica, un deseo, un calendario). Escribe eso, luego las tablas, luego el PDDL.
4. **Las pruebas antiguas se traducen y se conservan.** Los problemas de la versión anterior, con la nueva sintaxis y los mismos veredictos. Un veredicto que cambia debe explicarlo una línea de la ficha que haya cambiado.
5. **Pruebas nuevas para la dimensión nueva.** Al menos un «sin plan» y una trampa que la versión anterior no podía ni siquiera enunciar.
6. **Cada versión es un fichero.** Las versiones antiguas no se borran: son los modelos baratos que responden a las preguntas sencillas, y la referencia cuando una versión nueva se porta mal.
7. **Para** cuando el modelo ya no mienta sobre aquello de lo que depende el problema.

El dominio del ejemplo es nuevo en el libro. La historia:

> *Sábado por la mañana en la lavandería. Hay coladas que lavar: ropa blanca, ropa de color, toallas y algunos días un jersey de lana. Cada colada se lava, luego se seca y luego se dobla. La etiqueta de cada prenda dice en qué máquinas puede ir; la lana no puede ir a la secadora. Hay una lavadora y una secadora.*

Versión 1: clásica

Paso M: la ficha del modelo de la lavandería, versión 1

Historia	Una persona lava, seca y dobla coladas de ropa.
Agentes	La persona. Las máquinas son herramientas: no deciden nada.
Éxito	Todas las coladas del objetivo están dobladas.
Alcance	Dentro: si una colada está sucia, mojada o doblada; dónde está (el cesto o una máquina); qué máquinas admite su etiqueta. Fuera: el tiempo, los kilos, las monedas, el detergente y los programas, el tendedero.
Supuestos	Los clásicos. Una máquina admite una colada. Un ciclo nunca falla.

Los pasos O y D llevan unos minutos con las listas de comprobación y el catálogo. La Tabla 19.12 contiene las dos tablas, con el patrón que hay detrás de cada fila.

Propiedad	De qué trata	Clase, patrón
`(dirty ?l) (wet ?l)`	el estado de una colada: sucia; limpia y mojada; limpia y seca cuando no se cumple ninguno	dinámica; dos hechos, tres estados
`(in-basket ?l) (in ?l ?m)`	dónde está una colada	dinámica; invariante de a lo sumo uno
`(empty ?m)`	la máquina no tiene ninguna colada	dinámica; gemelo de `in`, cerrojo de recurso
`(allowed ?l ?m)`	la etiqueta de la colada admite la máquina	tabla estática
`(folded ?l)`	la colada está terminada	dinámica
Acción	**Se aplica cuando**	**Cambia**
`put ?l ?m`	la colada está en el cesto, y la máquina, vacía y permitida	la colada está en la máquina; par inverso con `take-out`
`wash ?l ?w`	la colada está en la lavadora y sucia	limpia y mojada; sin vuelta atrás
`dry ?l ?d`	la colada está en la secadora y mojada	seca; sin vuelta atrás
`fold ?l`	la colada está en el cesto, ni sucia ni mojada	doblada; sin vuelta atrás

Tabla 19.12.: La tabla de estado y la tabla de acciones de la lavandería, versión 1.

```
;; Laundry, version 1: classical.
;; Loads are washed, dried and folded; a machine holds one load.
(define (domain laundry-v1)
  (:requirements :typing :negative-preconditions)

  (:types load machine - object
          washer dryer - machine)

  (:predicates
    (dirty ?l - load)
    (wet ?l - load)
    (folded ?l - load)
    (in-basket ?l - load)              ; out of every machine
    (in ?l - load ?m - machine)
    (empty ?m - machine)
    (allowed ?l - load ?m - machine))  ; static: the label

  (:action put
    :parameters (?l - load ?m - machine)
    :precondition (and (in-basket ?l) (empty ?m) (allowed ?l ?m))
    :effect (and (not (in-basket ?l)) (not (empty ?m))
                 (in ?l ?m)))

  (:action take-out
    :parameters (?l - load ?m - machine)
    :precondition (in ?l ?m)
    :effect (and (not (in ?l ?m)) (in-basket ?l) (empty ?m)))

  (:action wash
    :parameters (?l - load ?w - washer)
    :precondition (and (in ?l ?w) (dirty ?l))
    :effect (and (not (dirty ?l)) (wet ?l)))

  (:action dry
    :parameters (?l - load ?d - dryer)
    :precondition (and (in ?l ?d) (wet ?l))
    :effect (not (wet ?l)))

  ;; Only what is clean and dry is folded
  (:action fold
    :parameters (?l - load)
    :precondition (and (in-basket ?l) (not (dirty ?l))
                       (not (wet ?l)))
    :effect (folded ?l)))
```

La batería de pruebas tiene los cuatro problemas del método. El trivial es una colada de toallas, y su plan se puede escribir antes de ejecutar: siete pasos.

Listado 19.1: La lavandería, versión 1 (`laundry-v1-domain.pddl`).

```
$ kairos laundry-v1-domain.pddl laundry-v1-p01.pddl
; cost = 7
(put towels w1)
(wash towels w1)
(take-out towels w1)
(put towels d1)
(dry towels d1)
(take-out towels d1)
(fold towels)
```

El problema típico, tres coladas, da 21 pasos, un óptimo demostrado. El problema que no debe tener plan es el jersey de lana: la etiqueta no admite la secadora, y esta versión no tiene otra manera de secar.

```
$ kairos laundry-v1-domain.pddl laundry-v1-p03.pddl
note: no plan: every state reachable from the initial one was
     searched (3 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

Dos atajos tientan al planificador, y los dos se escriben a mano como ficheros de plan: lavar dos coladas a la vez y doblar las toallas en cuanto salen de la lavadora.

```
$ kairos validate laundry-v1-domain.pddl laundry-v1-p04.pddl \
    laundry-v1-p04-cheat.plan
laundry-v1-p04-cheat.plan:3: error: step 2 (put colors w1) at 1.000:
      precondition false: (empty w1) is false [validate.
     precondition_failed]
 3 | (put colors w1)
plan invalid
$ kairos validate laundry-v1-domain.pddl laundry-v1-p01.pddl \
    laundry-v1-p01-wet.plan
laundry-v1-p01-wet.plan:5: error: step 4 (fold towels) at 3.000:
     precondition false: (wet towels) is true [validate.
     precondition_failed]
 5 | (fold towels)
plan invalid
```

La versión 1 está en verde. Ya responde a una pregunta real, en qué orden hacer las cosas, y ha fijado aquello en lo que se apoyará cada versión posterior: una colada está en un solo sitio, una colada limpia no vuelve a ensuciarse y solo se dobla la ropa limpia y seca.

Versión 2: números

La primera prueba de la trampa molesta a cualquiera que haya usado una lavadora: claro que dos coladas van en un mismo lavado, si caben. La ficha cambia. *Alcance*: los kilos entran. *Supuestos*: una máquina admite coladas hasta su capacidad, y un ciclo trata todo lo que hay dentro.

La lista O hace sus tres preguntas sobre la propiedad nueva (Capítulo 8). ¿Es una magnitud? Sí: los pesos se suman. ¿Solo importa un umbral?

No: cada colada ocupa una cantidad distinta. ¿Está acotada? Entre cero y la capacidad del tambor, y la precondición de `put` la mantiene ahí. El gemelo `empty` desaparece: su lugar lo ocupa el espacio que queda en el tambor, un número. La lista D pregunta si una acción cambia muchos objetos a la vez: ahora un ciclo lava todo lo que haya en el tambor, lo que es un efecto condicional universal (Capítulo 6). Un salto en el modelo ha arrastrado consigo una característica de otro capítulo, y eso es normal.

```
  (:functions
    (weight ?l - load)                ; kg, static
    (room ?m - machine))              ; kg still free in the drum

  (:action put
    :parameters (?l - load ?m - machine)
    :precondition (and (in-basket ?l) (allowed ?l ?m)
                       (>= (room ?m) (weight ?l)))
    :effect (and (not (in-basket ?l)) (in ?l ?m)
                 (decrease (room ?m) (weight ?l))))

  (:action take-out
    :parameters (?l - load ?m - machine)
    :precondition (in ?l ?m)
    :effect (and (not (in ?l ?m)) (in-basket ?l)
                 (increase (room ?m) (weight ?l))))

  ;; One cycle washes every load in the drum
  (:action wash
    :parameters (?w - washer)
    :effect (forall (?l - load)
              (when (in ?l ?w)
                (and (not (dirty ?l)) (wet ?l)))))

  (:action dry
    :parameters (?d - dryer)
    :effect (forall (?l - load)
              (when (in ?l ?d) (not (wet ?l)))))
```

Listado 19.2: La lavandería, versión 2: capacidad en kilos y ciclos que tratan todo el tambor (parte de `laundry-v2-domain.pddl`).

Las pruebas antiguas se traducen: las mismas coladas, ahora con pesos, en tambores de 8 kg. La trivial sigue teniendo siete pasos. La típica tiene coladas de 3, 4 y 5 kg, así que dos de ellas caben en un ciclo y las tres no:

```
$ kairos laundry-v2-domain.pddl laundry-v2-p02.pddl
; cost = 21
...
; better plan 2
; cost = 19
(put towels w1)
(wash w1)
(take-out towels w1)
(put towels d1)
(dry d1)
(take-out towels d1)
(fold towels)
(put whites w1)
(put colors w1)
(wash w1)
...
```

Diecinueve pasos: dos lavados y dos secados donde la versión 1 necesitaba tres de cada. Las pruebas nuevas son las dos que el Capítulo 8 pide para cada magnitud. Un problema que necesita más de lo que hay: un edredón de 9 kg, que es `unsolvable`. Y la trampa al otro lado de la cota, las tres coladas en un solo lavado:

```
$ kairos validate laundry-v2-domain.pddl laundry-v2-p02.pddl \
    laundry-v2-p02-cheat.plan
laundry-v2-p02-cheat.plan:4: error: step 3 (put towels w1) at 2.000:
      precondition false: (>= (room ?m) (weight ?l)): 1 >= 5 is
     false [validate.precondition_failed]
 4 | (put towels w1)
plan invalid
```

Un veredicto ha cambiado entre las versiones: la trampa de la versión 1, dos coladas en un tambor, es ahora un plan legal. La regla 4 pide la línea de la ficha que lo explica, y ahí está. Lo que era una regla del modelo («una máquina admite una colada») se ha sustituido por la regla del mundo.

Versión 3: tiempo

Un plan de 19 pasos no dice nada de la mañana. La lavadora tarda media hora y funciona sola; una persona que espera delante de ella pierde esa media hora, y una persona que lee el plan de la versión 2 hace exactamente eso. La ficha cambia. *Éxito*: todas las coladas dobladas, *lo antes posible*. *Agentes*: la persona hace una sola cosa a la vez; las máquinas funcionan solas una vez puestas en marcha. *Alcance*: las duraciones entran, en minutos.

Las cuatro columnas que el Capítulo 10 añade a la tabla de acciones se rellenan en la Tabla 19.13.

Acción	Duración	Debe cumplirse	Puede solaparse con
`put, take-out`	1	la máquina está parada, todo el tiempo	todo lo que no necesite las manos
`wash, dry`	el ciclo de la máquina	parada al inicio; algo en el tambor	todo, salvo abrir su propia puerta
`fold`	5	limpia y seca al inicio	los ciclos

Tabla 19.13.: La lavandería, versión 3: las columnas de tiempo de la tabla de acciones.

La tabla nombra dos testigos (Capítulo 10): `(hands-free)` para la persona y `(stopped ?m)` para cada máquina, los dos tomados al inicio y devueltos al final. Cargar y descargar necesitan la máquina parada *todo el tiempo*, un invariante, para que nadie abra la puerta de una máquina en marcha.

```
(:durative-action put
  :parameters (?l - load ?m - machine)
  :duration (= ?duration 1)
  :condition (and (at start (in-basket ?l))
                  (at start (hands-free))
                  (at start (>= (room ?m) (weight ?l)))
                  (over all (allowed ?l ?m))
                  (over all (stopped ?m)))
  :effect (and (at start (not (in-basket ?l)))
               (at start (not (hands-free)))
               (at start (decrease (room ?m) (weight ?l)))
               (at end (in ?l ?m))
               (at end (hands-free))))
;; The machine works alone: the person is free meanwhile
(:durative-action wash
  :parameters (?w - washer)
  :duration (= ?duration (cycle ?w))
  :condition (and (at start (stopped ?w))
                  (at start (exists (?l - load) (in ?l ?w))))
  :effect (and (at start (not (stopped ?w)))
               (at end (stopped ?w))
               (forall (?l - load)
                 (when (at end (in ?l ?w))
```

```
51                    (at end (and (not (dirty ?l)) (wet ?l)))))))
```

Listado 19.3: La lavandería, versión 3: dos de sus acciones durativas (parte de `laundry-v3-domain.pddl`).

Con la condición escrita `at start` y el efecto `at end`, Kairos 1.0.6 planifica el dominio solo con su motor M2, que no solapa acciones (Capítulo 11); si se le obliga a usar M3, responde que «todavía no puede comprobarlas». Escrita como en el listado, se aplican los dos motores.

Una línea del listado merece un comentario. El efecto condicional de `wash` lee lo que hay en el tambor *al final*. Como la puerta está cerrada durante el ciclo, eso es lo que había en el tambor al inicio, y la condición podría escribirse de las dos maneras.

Las pruebas antiguas se traducen otra vez, y el primer resultado de la típica es una lección. El fichero `laundry-v3-idle-domain.pddl` es la primera versión 3 que escribimos. Su plan para las tres coladas tiene el *makespan* correcto y una línea de más:

```
$ kairos laundry-v3-idle-domain.pddl laundry-v3-p02.pddl
...
79.008: (dry d1) [40.000]
79.009: (fold towels) [5.000]
84.010: (fold whites) [5.000]
90.011: (wash w1) [30.000]
119.009: (take-out colors d1) [1.000]
120.010: (fold colors) [5.000]
```

En el minuto 90 la lavadora hace un ciclo sin nada dentro. En la versión 2 el planificador nunca lo hacía, porque cada acción alargaba el plan. Ahora la medida es el *makespan*, el ciclo vacío termina antes que el plan y no cuesta nada. El modelo siempre lo había permitido; el salto cambió aquello que le da igual al planificador. Es la rama «un plan erróneo» del procedimiento de depuración: no se rompe ninguna regla de la ficha, pero nadie paga un ciclo de una máquina vacía, y lo primero que el mundo rechaza es ese lavado. La causa es una precondición que falta: un ciclo necesita algo en el tambor. La versión 3 tal como aparece en el Listado 19.3 la tiene, y da:

```
$ kairos laundry-v3-domain.pddl laundry-v3-p02.pddl
...
; better plan 7
; cost = 125.01100000000002
; makespan = 125.011
0.000: (put whites w1) [1.000]
1.001: (put towels w1) [1.000]
2.001: (wash w1) [30.000]
32.002: (take-out whites w1) [1.000]
33.003: (take-out towels w1) [1.000]
34.004: (put whites d1) [1.000]
35.005: (put towels d1) [1.000]
36.005: (dry d1) [40.000]
36.006: (put colors w1) [1.000]
37.007: (wash w1) [30.000]
67.008: (take-out colors w1) [1.000]
76.006: (take-out towels d1) [1.000]
77.007: (take-out whites d1) [1.000]
78.008: (put colors d1) [1.000]
79.009: (dry d1) [40.000]
79.009: (fold towels) [5.000]
84.010: (fold whites) [5.000]
119.010: (take-out colors d1) [1.000]
120.011: (fold colors) [5.000]
```

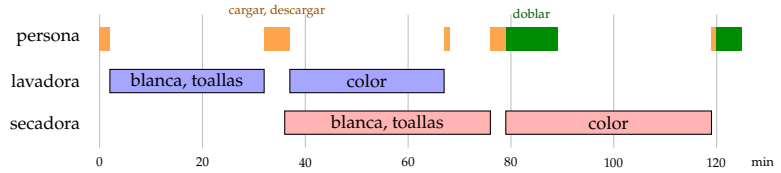


Figura 19.5.: El plan de la versión 3 para las tres coladas, como cronograma. El segundo lavado se hace mientras la secadora trabaja con la primera tanda, y la persona dobla durante el segundo secado.

El cronograma de la Figura 19.5 es la manera de leer un plan temporal (Capítulo 10). La secadora es el cuello de botella: dos ciclos de 40 minutos que no pueden solaparse, tras un lavado de 30 y unos minutos de manipulación; ningún horario puede ser mucho más corto que 125. Conocer el *makespan* honrado antes de fiarse del planificador es la primera prueba de la trampa del Capítulo 10. Las otras dos se escriben a mano, cada una como un plan que rompe un testigo: abrir la lavadora a los diez minutos de empezar su ciclo y cargar dos cosas con las mismas manos.

```
$ kairos validate laundry-v3-domain.pddl laundry-v3-p02.pddl \
    laundry-v3-p02-door.plan
laundry-v3-p02-door.plan:4: error: step 3 (take-out whites w1) at
     11.000: over-all condition broken after the happening at
     11.000000: (stopped w1) is false [validate.invariant_failed]
 4 | 11.000: (take-out whites w1) [1.000]
plan invalid
```

Versión 4: deseos

El plan de 125 minutos es el mejor para una persona que tiene toda la mañana. La mayoría no la tiene, y preferiría dejar las toallas para otro día antes que quedarse por ellas. La ficha cambia donde el Capítulo 14 dijo que lo haría: el *Éxito* se divide en lo obligatorio y lo deseado, y cada deseo recibe un precio en la unidad de la métrica, los minutos.

Paso M, versión 4: los deseos

Nombre	Condición	Peso	Razón
(obligatorio)	la ropa blanca está doblada		la camisa del lunes
`colors-done`	la ropa de color está doblada	60	vale una hora de la mañana
`towels-done`	las toallas están dobladas	30	en casa hay toallas limpias

Calidad: los minutos hasta que termina la última acción, más el peso de cada deseo al que se renuncia. Cuando no se concede nada, el plan lava la ropa blanca sola.

Los pasos O y D no cambian: una preferencia usa los predicados y las acciones que ya hay (Capítulo 14). El dominio gana una palabra, `:preferences`, en sus requisitos, y todo el salto está en el fichero del problema:

```
(:goal (and (folded whites)
            (preference colors-done (folded colors))
            (preference towels-done (folded towels))))
(:metric minimize (+ (total-time)
                     (* 60 (is-violated colors-done))
                     (* 30 (is-violated towels-done)))))
```

```
$ kairos laundry-v4-domain.pddl laundry-v4-p02.pddl
; cost = 169.006
; makespan = 79.006
...
; better plan 2
; cost = 118.01
; makespan = 88.010
0.000: (put whites w1) [1.000]
1.001: (put colors w1) [1.000]
2.001: (wash w1) [30.000]
...
83.010: (fold colors) [5.000]
```

Listado 19.4: La lavandería, versión 4: el objetivo y la métrica de `laundry-v4-p02.pddl`.

El primer plan lava la ropa blanca sola: 79 minutos, y 90 por los dos deseos a los que renuncia. El segundo mete también la ropa de color en el mismo tambor, 9 minutos más por un deseo que vale 60, y deja las toallas: necesitarían una segunda tanda, 37 minutos más (Figura 19.5), por un deseo que vale 30. El plan es el que elegiría una persona sensata.

Las pruebas del Capítulo 14 son dos. Sin nada que se pueda conceder (un edredón de 9 kg como deseo, `p01`), el plan está vacío y cuesta el peso. Y la prueba del precio: cambia un peso hasta que cambie el plan, y comprueba que cambia donde debe. En `p04` la ropa de color vale solo 5 minutos, menos que los 9 que añade, y Kairos la deja fuera; las toallas, de 5 kg, ocupan su lugar junto a la ropa blanca en el tambor, con un coste de 93,010. El peso a partir del cual la ropa de color entra en el plan está entre 5 y 60, y según las cuentas es 9.

La prueba del precio de las toallas es el segundo caso que anunció la Sección 19.6. Su punto de equilibrio son 37 minutos. En `p03` valen 40, así que el mejor plan debería lavarlo todo en 125,011. Kairos devuelve otro:

```
$ kairos laundry-v4-domain.pddl laundry-v4-p03.pddl
...
; better plan 3
; cost = 128.01
; makespan = 88.010
...
$ kairos validate laundry-v4-domain.pddl laundry-v4-p03.pddl \
    laundry-v4-p03-all.plan
plan valid, makespan 125.011, metric 125.011
```

El fichero `laundry-v4-p03-all.plan` es el horario de la versión 3, tecleado. El validador lo acepta para este problema con una métrica de 125,011, mejor que el 128,010 del plan devuelto. Así que el modelo está bien, y la búsqueda se quedó corta: con `-v`, Kairos 1.0.6 muestra que planificó este problema solo con su motor M2, el que pone acciones enteras una tras otra, y una tanda de toallas que no se solapa con la primera cuesta 79 minutos, no 37. Un plan pobre, uno mejor a mano, el validador como juez: la última fila de la Figura 19.3. Lo anotamos en la documentación del modelo y seguimos.

Versión 5: procesos

Una prueba ha dado el mismo veredicto desde la versión 1: el jersey de lana no tiene plan. La historia tiene la respuesta tendida en el patio. *Alcance*: el tendedero entra.

La primera pregunta de la lista D decide cómo. Imagina que la persona se queda dormida después de tender el jersey. Se seca igualmente, así que secar en el tendedero no es una acción. Se seca mientras está tendido y brilla el sol: un proceso. Está seco en un instante, lo mire alguien o no: un evento. Y el sol sigue un calendario que la persona no puede cambiar: literales iniciales temporizados (Capítulo 13). La lista O pregunta, para una magnitud que cambia por sí sola, qué hecho enciende y apaga el cambio y qué número lo mide (Capítulo 17): los hechos son `(hanging ?l)` y `(sunny)`, y el número es la humedad de la colada, 100 cuando sale de la lavadora.

```
(:durative-action hang
  :parameters (?l - load)
  :duration (= ?duration 2)
  :condition (and (at start (in-basket ?l))
                  (at start (hands-free)))
  :effect (and (at start (not (in-basket ?l)))
               (at start (not (hands-free)))
               (at end (hanging ?l))
               (at end (hands-free))))
;; Nobody chooses this: it runs while its condition holds
(:process sun-dry
  :parameters (?l - load)
  :precondition (and (hanging ?l) (wet ?l) (sunny))
  :effect (decrease (moisture ?l) (* #t (sun-rate))))

;; ... and this happens at the instant the load is dry
(:event dried
  :parameters (?l - load)
  :precondition (and (wet ?l) (<= (moisture ?l) 0))
  :effect (not (wet ?l))))
```

Listado 19.5: La lavandería, versión 5: una acción de la persona, un proceso y un evento (parte de `laundry-v5-domain.pddl`).

Todo lo demás sigue como estaba. Lavar y secar en la secadora siguen siendo acciones durativas: su duración se conoce al empezar y nada las interrumpe, que es el consejo del Capítulo 17 y del Capítulo 18 para mezclar las dos cosas. Ahora el lavado, además, pone la humedad a 100.

La antigua prueba «sin plan» cambia de veredicto, y la ficha dice por qué. Con sol toda la mañana (`p01`) el jersey está doblado en el minuto 90: lavado a los 31, tendido a los 34, cincuenta minutos en el tendedero a 2 unidades por minuto. La prueba típica (`p02`) añade la ropa blanca y una nube del minuto 60 al minuto 90:

```
$ kairos laundry-v5-domain.pddl laundry-v5-p02.pddl
...
; better plan 2
; cost = 169
; makespan = 169.000
0.000: (put whites w1) [1.000]
1.000: (put wool w1) [1.000]
2.000: (wash w1) [30.000]
32.000: (take-out whites w1) [1.000]
33.000: (take-out wool w1) [1.000]
34.000: (hang wool) [2.000]
116.000: (collect wool) [1.000]
117.000: (fold wool) [5.000]
122.000: (put whites d1) [1.000]
123.000: (dry d1) [40.000]
163.000: (take-out whites d1) [1.000]
164.000: (fold whites) [5.000]
169.000: @PlanEND
```

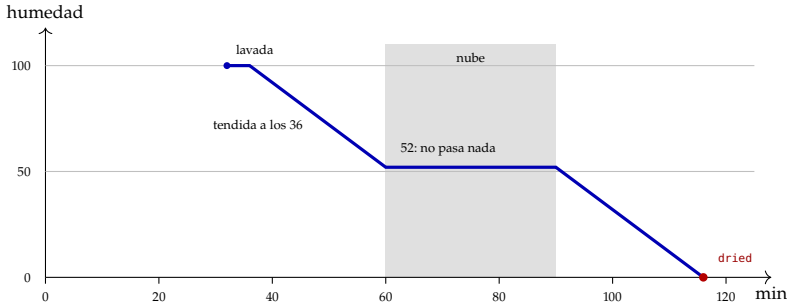


Figura 19.6.: La humedad de la lana a lo largo del plan para `laundry-v5-p02`. El proceso `sun-dry` funciona desde el minuto 36, se detiene bajo la nube y se reanuda a los 90; el evento se dispara a los 116, y solo entonces se puede recoger y doblar la colada.

Reproducir el plan (Figura 19.6) confirma la parte que importa: la lana se recoge a los 116, cuando el evento se ha disparado, tras 24 minutos de sol, 30 de nube y otros 26 de sol. También muestra la debilidad de la que avisó el Capítulo 18. La ropa blanca se queda mojada en el cesto del minuto 33 al 122 y solo va a la secadora cuando la lana está doblada: en PDDL+ la búsqueda minimiza el número de acciones, y nada premia hacer dos cosas a la vez. El plan mejor es fácil de escribir, y el validador lo acepta:

```
$ kairos validate laundry-v5-domain.pddl laundry-v5-p02.pddl \
    laundry-v5-p02-hand.plan
plan valid, makespan 122, metric 122
```

Las pruebas nuevas son las de un modelo híbrido. La trampa que esta versión hace posible es un viaje gratis (Capítulo 17): ropa que se seca a la sombra. El plan `laundry-v5-p02-shade.plan` recoge la lana tras cincuenta minutos en el tendedero, haya nube o no, y se rechaza al doblar: `(wet wool) is true`. La prueba «sin plan» es una mañana nublada, en la que `(sunny)` no se cumple nunca:

```
$ kairos laundry-v5-domain.pddl laundry-v5-p03.pddl
note: the search minimizes the number of actions; the metric of the
     problem is reported for each plan but does not guide the search
      yet [orchestrator.metric]
note: no plan in steps of time; a plan that needs finer timing than
     the steps would be missed, so this is evidence but not a proof
     [orchestrator.steps_exhausted]
note: no plan: the engines that apply ran out of states (8093 states)
      before the time limit, so more time does not help; none of
     them is complete for this task, so a plan may still exist (see
     the note before). [orchestrator.why]
kairos: no plan found (exhausted)
```

El veredicto es `no plan found`, código de salida 14, como en todo problema PDDL+ sin plan (Capítulo 17). Aquí el lápiz del Capítulo 18 da la demostración que el planificador no puede dar: sin sol, la humedad se queda en 100. Los deseos de la versión 4 siguen funcionando por encima de los procesos: en `p04`, una mañana nublada con la lana como deseo que vale 30, el plan lava y seca la ropa blanca en 79 minutos y paga los 30.

Cinco versiones, una al lado de otra

	Añade a la ficha	Prueba típica	Búsqueda
v1	orden del trabajo; una colada por máquina	21 pasos, óptimo	808 estados, 3 ms
v2	kilos; un ciclo trata todo el tambor	19 pasos, óptimo	2106 estados, 12 ms
v3	duraciones; qué puede solaparse; el *makespan*	125,011 min	101 696 estados, 2,1 s
v4	obligatorio y deseado; precios en minutos	coste 118,010	5358 estados, 20 ms
v5	el tendedero, el sol, la nube	169 min (122 a mano)	168 296 estados, 2,0 s

Tabla 19.14.: Las cinco versiones de la lavandería en sus pruebas típicas (el `p02` de cada una), con las cifras de `kairos -v` en un hilo. Los problemas no son los mismos, así que la última columna compara órdenes de magnitud, no versiones.

La Tabla 19.14 es el compromiso de todo el libro en cinco líneas. Cada versión dice algo cierto que la anterior no podía decir, y cada una cuesta más de buscar y funciona en menos planificadores: según las cuentas de la Tabla 19.5, los efectos condicionales de la versión 2 ya dejan tres planificadores de ocho, y los procesos de la versión 5 dejan uno. Las versiones no son borradores de la última. Qué coladas caben juntas lo responde mejor la versión 2, en milisegundos y con demostración; el horario de una mañana corriente, la versión 3; la versión 5 solo hace falta los días con lana.

Haz crecer un modelo una dimensión cada vez: primero la ficha, las pruebas antiguas traducidas y conservadas, pruebas nuevas para lo nuevo. Cada versión se queda como un fichero, y a menudo como la mejor respuesta a una pregunta más sencilla.

19.8. Documentar un modelo

Un modelo se lee más veces de las que se escribe: lo lee la persona que escribirá los problemas, la que lo conectará a un robot, su autor seis meses después. PDDL tiene una herramienta para ellos, el comentario, y el método tiene cinco artefactos. Documentar un modelo es poner cada artefacto donde su lector lo va a buscar.

La cabecera del dominio. Lo que necesita quien lee el fichero antes del primer paréntesis: qué es el modelo, quién decide qué, qué se deja fuera, las unidades, las reglas con el lugar donde se impone cada una, los invariantes, qué debe aportar un problema, dónde están las pruebas y cómo ha llegado el modelo hasta aquí. El Listado 19.6 es la cabecera de la última versión de la lavandería, y una plantilla para cualquier dominio.

```
;; Laundry, version 5: processes.
;; A clothesline dries what hangs on it while the sun shines,
;; whether anybody is there or not.
;;
;; Story    One person washes, dries and folds loads of clothes
;;          with a washer, a dryer and a clothesline.
;; Decides  The person: put, take-out, wash, dry, fold, hang,
;;          collect. The world: sun-dry (process), dried (event).
;; Out      Detergent, programs, ironing, a second person.
;; Units    Minutes, kilograms, moisture in percent.
;; Rules    R1 a drum holds up to its capacity     (put)
;;          R2 a cycle treats what is in the drum  (wash, dry)
;;          R3 the door stays shut during a cycle  (stopped)
;;          R4 the person does one thing at a time (hands-free)
;;          R5 only clean, dry loads are folded    (fold)
;;          R6 the line dries only in the sun      (sun-dry)
;; Keeps    A load is in the basket, in one machine or on the
;;          line. room = capacity minus the weights inside.
;; Problem  Gives weight, room, cycle, sun-rate and allowed;
;;          moisture 0 for every load; the sun as timed
```

```
;;          literals; the wishes and their weights.
;; Tests    tests.txt. p01 trivial, p02 typical, p03 no plan,
;;          p04 wishes; p02-shade.plan is the cheat.
;; History  v1 classical. v2 capacity in kilograms. v3 time.
;;          v4 wishes. v5 the clothesline.
```

Listado 19.6: Un comentario de cabecera como documentación: las primeras líneas de `laundry-v5-domain.pddl`.

Cada apartado tiene su razón. *Decides* separa las acciones del agente de los procesos y eventos del mundo, que es lo primero que debe saber quien use un modelo híbrido. *Units* evita el error silencioso más común de los modelos numéricos, un peso en gramos en un problema y en kilos en el siguiente. *Rules* nombra, para cada regla de la ficha, la acción o el predicado que la impone, de modo que una regla sin sitio se ve enseguida. *Keeps* enuncia los invariantes que comprueban las pruebas. *Problem* es el contrato con quien escriba ficheros de problema: cada función que necesita un valor, cada hecho que debe estar (Capítulo 8).

Dentro del fichero, un comentario va donde un lector se detendría: junto a cada predicado cuyo nombre no lo dice todo, junto a cada función estática con su unidad, encima de cada acción cuya precondición codifica una regla. Los comentarios que repiten el PDDL con palabras son ruido.

La ficha del modelo. La cabecera es la versión corta. La ficha es la completa, y es el documento que hay que enseñar a los dueños del problema, que no leen PDDL. Después de dieciocho capítulos tiene más que sus cinco líneas originales.

La ficha del modelo, completa: una plantilla

Historia	La situación en dos o tres frases, con las palabras de su dueño.
Agentes	Quién decide. Quién y qué solo ocurre. Quién puede actuar al mismo tiempo.
Éxito	Los objetivos obligatorios.
Calidad	Qué hace mejor a un plan, con su unidad; los pesos y su razón.
Deseos	Una tabla: nombre, condición, peso, razón. El plan cuando no se concede ninguno.
Reglas	Una tabla: la frase tal como se dijo, su ámbito (una acción o todo el plan), su fuerza (estricta o blanda), dónde está codificada.
Alcance	Dentro, y fuera. Para cada cosa que se deja fuera, quién se encarga de ella.
Supuestos	Lo que se da por hecho sobre el mundo: qué no puede fallar, qué cambia en momentos conocidos (el calendario), qué cambia por sí solo.
Nivel	Clásico, numérico, temporal o PDDL+, los requisitos y los planificadores con los que se ha ejecutado el modelo.
Decisiones	Las comparativas del paso L: qué se consideró, qué se eligió, por qué.
Límites conocidos	Lo que el modelo hace mal a propósito, y lo que el planificador no encuentra en él.
Versiones	Una línea por versión: qué añadió, y la pregunta a la que mejor responde.

Para la lavandería, la entrada *Límites conocidos* tiene dos líneas que esta sección se ha ganado: los planes de la versión 4 se pueden mejorar cuando

una segunda tanda podría solaparse con la primera, y los planes de la versión 5 no tienen prisa.

Las pruebas. La tabla de estado y la tabla de acciones encajan bien como comentarios del dominio. La batería de pruebas necesita una forma que pueda ejecutar una máquina. En este libro es el fichero `tests.txt` de cada carpeta de ejemplos, una línea por prueba: qué hacer, con qué ficheros y el código de salida esperado (Tabla 19.11). Estas son las líneas de la versión 5:

```
# Version 5: processes
check 0 laundry-v5-domain.pddl laundry-v5-p02.pddl
solve laundry-v5-domain.pddl laundry-v5-p01.pddl 0
solve laundry-v5-domain.pddl laundry-v5-p02.pddl 0
validate laundry-v5-domain.pddl laundry-v5-p02.pddl laundry-v5-p02-hand.
     plan 0
validate laundry-v5-domain.pddl laundry-v5-p02.pddl laundry-v5-p02-
     shade.plan 1
solve laundry-v5-domain.pddl laundry-v5-p03.pddl 14
solve laundry-v5-domain.pddl laundry-v5-p04.pddl 0
```

Un fichero de pruebas de este tipo es documentación que no puede quedarse anticuada sin que alguien lo note: dice para qué problemas es el modelo, cuáles debe rechazar y qué planes debe rechazar. Cada fichero de problema empieza con un comentario que dice cuál de las cuatro pruebas es y qué se espera, y cada plan escrito a mano dice qué intenta.

La cabecera del dominio para quien lee el fichero, la ficha del modelo para quien es dueño del problema, un fichero de pruebas ejecutable para quien cambie el modelo después.

19.9. El método MODEL: el método completo

Este capítulo ha reunido las preguntas de los demás. Añade unas pocas propias, que no tratan de ninguna característica del lenguaje: tratan del modelo como algo que vive más de una tarde.

Preguntas que este capítulo añade al método

M	¿Qué versión es esta, y qué ha pasado de «fuera» a «dentro» desde la anterior? ¿A qué pregunta responde esta versión que la anterior no podía responder?
O, D	¿Qué patrón es cada fila de las tablas, por su nombre en el catálogo?
E	¿Están traducidas las pruebas de la versión anterior, y conservan sus veredictos? ¿Es cada veredicto una demostración? ¿Falla cada trampa por la razón esperada?
L	¿Hay en el modelo alguna característica que pudiera sustituir un patrón de una fila más alta de la guía? ¿Basta una versión inferior para la pregunta que se hace hoy?
Todos	¿Está cada artefacto escrito donde lo buscará su lector: la cabecera, la ficha, el fichero de pruebas?

Alternativas comparadas

El salto a la versión 5 tomó una decisión que el paso L debe revisar: ¿es el tendedero un proceso? El Capítulo 17 aconseja una acción durativa siempre que una actividad tiene una duración conocida al empezar y nada puede interrumpirla. Así que escribimos el tendedero de tres

maneras, sobre la versión 3, y ejecutamos las tres sobre la misma mañana: la lana, la ropa blanca y la nube del minuto 60 al 90.

A. Una acción durativa de duración fija (`laundry-line-fixed`). Una acción `line-dry` dura los 50 minutos de sol que necesita una colada y no mira al cielo.

B. Una acción durativa dentro de una ventana (`laundry-line-window`). La misma acción con `(over all (sunny))`: cincuenta minutos de sol sin interrupción.

C. Un proceso y un evento (`laundry-v5`). El modelo del Listado 19.5.

```
$ kairos laundry-line-fixed-domain.pddl laundry-line-fixed-p02.pddl
...
35.004: (line-dry wool) [50.000]
...
85.004: (collect wool) [1.000]
86.005: (fold wool) [5.000]
$ kairos laundry-line-window-domain.pddl laundry-line-window-p02.
    pddl
...
81.008: (hang wool) [2.000]
90.001: (line-dry wool) [50.000]
140.001: (collect wool) [1.000]
141.002: (fold wool) [5.000]
```

La codificación A dobla la lana en el minuto 86, tras 25 minutos de sol y 25 de nube. La codificación B espera a que pase la nube y empieza los cincuenta minutos a los 90: el sol de antes de la nube no le sirve de nada, porque una acción durativa no puede hacer una pausa.

	A: duración fija	B: dentro de una ventana	C: proceso y evento
Fidelidad	un plan imposible: la lana se dobla mojada. Correcto solo en un día sin nubes	ningún plan imposible; soluciones perdidas: el secado no puede hacer pausas, y un día cuyos ratos de sol duran todos menos de 50 minutos no tiene plan	exacta: el secado se detiene bajo la nube y se reanuda
Tamaño	17 acciones durativas	17 acciones durativas	15 acciones durativas, 2 procesos, 2 eventos
Guía	10 407 estados; *makespan* 91,005	9535 estados; *makespan* 146,002	168 296 estados; *makespan* 169, y 122 a mano
Claridad	una acción más, como las otras	lo mismo, con una condición	la ley del tendedero está aparte de las tareas; hay que reproducir los planes
Portabilidad	acciones durativas, literales temporizados	lo mismo	procesos: solo Kairos entre los planificadores comparados

Tabla 19.15.: Tres codificaciones del tendedero sobre la misma mañana (los problemas `p02` de cada dominio; cifras de `kairos -v`, un hilo).

Veredicto. A es el modelo para un sitio donde uno se puede fiar del sol, y para una primera versión: es la versión 3 con una acción más. No debe usarse cuando el cielo forma parte del problema, porque entonces sus planes son erróneos y nada lo avisa. B es el modelo honrado dentro del nivel temporal. Nunca devuelve un plan erróneo y funciona en los planificadores que leen literales temporizados; su precio son las soluciones que pierde, y esta mañana termina más tarde que el mejor plan de C hecho a mano (146 frente a 122), aunque antes que el plan

que Kairos encuentra para C. Elígelo cuando los ratos de sol sean largos comparados con el tiempo de secado. C es el único que dice la verdad sobre una mañana de sol y nubes, y hace falta justo cuando el secado se interrumpe y se reanuda, que es el caso que el Capítulo 17 reservaba para los procesos. La decisión depende de una pregunta del paso M a la que la historia no había respondido: *¿hasta qué punto es fiable el sol?* Como con el camarero del Capítulo 7, el paso L ha terminado devolviéndonos a la ficha.

19.10. Errores comunes

Los errores de este capítulo no están en una construcción. Son errores en la manera de trabajar, y todos se cometieron mientras se escribía la lavandería.

Una prueba de la trampa que pasa por la razón equivocada

Los ficheros de plan de la versión 2 se llevaron tal cual a la versión 3. La prueba sigue diciendo lo que debe, `plan invalid`:

```
$ kairos validate laundry-v3-domain.pddl laundry-v3-p02.pddl \
    laundry-v2-p02-cheat.plan
laundry-v2-p02-cheat.plan:2: error: step 1 (put whites w1) at
     0.000: duration 0.000000 does not satisfy (= ?duration 1):
     (= ?duration 1): 0 = 1 is false [validate.duration_mismatch
     ]
 2 | (put whites w1)
plan invalid
```

El plan se rechaza en su primera línea porque no tiene duraciones, no porque 12 kg no quepan en el tambor. La prueba está en verde y no prueba nada; si la regla de la capacidad se perdiera en el salto, seguiría en verde. *Corrección*: traduce los ficheros de plan junto con el modelo, y lee el mensaje de cada rechazo, no solo su veredicto.

Una cabecera que cuenta la historia de una versión anterior

`laundry-v2-stale-domain.pddl` es la versión 2 con el comentario y los requisitos del fichero del que se copió. Kairos la planifica y da un aviso:

```
$ kairos check laundry-v2-stale-domain.pddl
laundry-v2-stale-domain.pddl:5:18: warning: uses conditional-
     effects but :requirements does not include :conditional-
     effects [semantic.undeclared_requirement]
 5 |   (:requirements :typing :negative-preconditions
   |                  ^
valid, plannable
  typing: full
  negative-preconditions: full
  conditional-effects: full
  numeric-fluents: full
```

Los requisitos son la parte de la documentación que lee una máquina: otro planificador decide a partir de ellos si puede planificar el fichero. El comentario que tienen encima, que todavía dice que una máquina admite una colada, lo leen personas, y ninguna herramienta lo comprueba. *Corrección*: la cabecera forma parte del salto. La lista que imprime `kairos check` bajo su veredicto es lo que el fichero usa de verdad: copia de ella los requisitos.

Una prueba «sin plan» con el horizonte abierto

La primera prueba «sin plan» de la versión 5 dejaba brillar el sol hasta el minuto 50, dieciséis minutos de secado para un jersey que necesita cincuenta (`laundry-v5-p05.pddl`). Con un límite de tiempo de 3 segundos:

```
$ kairos laundry-v5-domain.pddl laundry-v5-p05.pddl --timeout 3
    -v
...
kairos: no plan found (exhausted)
kairos: parse 0.2 ms, normalize 0.1 ms, ground 0.2 ms, sas 0.2
    ms, search 3004.2 ms, total 3005.1 ms, first plan -1.0 ms
...
```

La búsqueda gastó los tres segundos, y 371765 estados. Con 20 segundos gasta los 20. Hasta el minuto 50, cada instante en el que se puede tender el jersey deja una humedad distinta, y cada uno es un estado que explorar. La mañana nublada de `p03` le hace al modelo la misma pregunta, si el tendedero seca sin sol, y se responde tras 8093 estados en menos de 0,1 segundos. *Corrección*: el consejo del Capítulo 18. Acota el horizonte de los problemas que no deben tener plan.

Un veredicto que se mantiene cuando la ficha ha cambiado

El jersey fue la prueba «sin plan» de cuatro versiones, y un fichero de pruebas copiado hacia delante seguiría esperando para él el código de salida 10 en la versión 5, donde la respuesta correcta es un plan de 90 minutos. El desliz contrario es más peligroso: ver que una prueba cambia de veredicto tras un salto y actualizar el valor esperado sin preguntarse por qué. *Corrección*: la regla 4 de la Sección 19.7. Un veredicto solo cambia junto con una línea de la ficha del modelo, y el comentario del principio del fichero de problema dice cuál.

19.11. Resumen

El método MODEL ya está completo. Nada en él es nuevo desde el Capítulo 1 salvo su contenido: cinco pasos, cada uno con una pregunta, un artefacto y, después de dieciocho capítulos, una lista de comprobación.

- **Cinco listas de comprobación** (Sección 19.2) contienen todas las preguntas que añadieron los capítulos. El paso M decide más que ningún otro: las métricas, el tiempo, los deseos, las reglas y un mundo que se mueve son entradas de la ficha del modelo antes de ser PDDL.
- **La guía de decisión** (Figura 19.2, Tablas 19.3 a 19.5) va de una necesidad a una característica, a su requisito y a su coste. Las características se añaden en el orden del libro, y el modelo se queda en el nivel más bajo que no miente sobre aquello de lo que depende el problema.
- **Cuarenta patrones** (Tablas 19.6 y 19.7) son el vocabulario de los pasos O y D. Cada uno tiene un rival, y los rivales son las columnas del paso L.
- **Los antipatrones** (Tablas 19.8 a 19.10) pasan la comprobación y se notan en los planes: demasiado fácil, demasiado difícil, demasiado grande, o correcto para otra pregunta.

- **La depuración** empieza leyendo el veredicto. `unsolvable` es una demostración; `no plan found` y `timeout` no lo son. Cuando el planificador no encuentra nada, o nada bueno, el plan escrito a mano y el validador distinguen el modelo de la búsqueda.
- **Un modelo crece por versiones**, una dimensión cada vez: primero la ficha, las pruebas antiguas traducidas y conservadas, pruebas nuevas para lo nuevo. La lavandería pasó de 44 líneas de PDDL clásico a procesos y eventos en cinco escalones, y cada versión sigue siendo la mejor respuesta a alguna pregunta.
- **La documentación** son los cinco artefactos en tres sitios: la cabecera del dominio, la ficha del modelo y un fichero de pruebas que ejecuta una máquina.

El capítulo siguiente lo usa todo en un problema del tamaño que tiene un robot real: un día de una flota de mensajeros en un hospital, de la historia a la ejecución del plan.

19.12. Lecturas complementarias

- El libro de Haslum y otros Haslum et al., 2019 es la referencia del lenguaje de todo el libro en un solo volumen, con la semántica de cada nivel. Es el sitio donde comprobar qué significa una construcción cuando un planificador y un validador no se ponen de acuerdo.
- Los libros de texto de Ghallab, Nau y Traverso Ghallab et al., 2004 y de Geffner y Bonet Geffner y Bonet, 2013 explican qué hacen los planificadores con un modelo, que es el fundamento de los criterios Tamaño y Guía. El libro posterior de los tres primeros autores Ghallab et al., 2016 trata modelos que se usan tanto para actuar como para planificar, el tema del capítulo siguiente.
- Los artículos que definen cada nivel discuten sus propias decisiones de modelado: Fox y Long para los números y el tiempo Fox y Long, 2003, Gerevini y otros para las preferencias y las restricciones A. E. Gerevini et al., 2009, y de nuevo Fox y Long para los procesos y los eventos Fox y Long, 2006.
- El artículo de Cushing y otros Cushing et al., 2007 es el análisis que hay detrás de la distinción entre los modelos que necesitan que las acciones se solapen y los que no, que este capítulo ha encontrado como un coste en Guía y en Portabilidad.
- El artículo de Howey, Long y Fox Howey et al., 2004 describe VAL, el validador de las competiciones de planificación. El hábito en el que más se apoya este capítulo, un plan escrito a mano y comprobado por un programa que no busca, viene de ahí.
- La tabla de características y planificadores que usa la Sección 19.3 se mantiene junto a la documentación de Kairos (Martín Rico et al., 2026).

19.13. Ejercicios

Los ejercicios de este capítulo son proyectos de modelado. Cada uno es una historia, y para cada uno los entregables son los mismos cinco artefactos: (1) la **ficha del modelo**; (2) la **tabla de estado**, con el patrón de cada fila; (3) la **tabla de acciones**; (4) la **batería de pruebas**, como ficheros PDDL con un `tests.txt`, que contenga al menos un problema

trivial, uno típico, uno que no debe tener plan y una trampa escrita a mano como fichero de plan; (5) la **comparativa** de la codificación elegida con al menos una alternativa, escrita y ejecutada, según los cinco criterios. Las historias dejan cosas sin decir a propósito: decidirlas, y escribir la decisión en la ficha, es parte del trabajo. Las soluciones dan un modelo de referencia para cada proyecto; un modelo distinto que pase su propia batería de pruebas honrada es igual de bueno.

Ejercicios de la Sección 19.2 — Las cinco listas de comprobación

Ejercicio 19.2.1. *La biblioteca.* Los libros que devuelven los lectores se amontonan en el mostrador de una biblioteca. Un robot los devuelve a las estanterías. Las estanterías están en pasillos, uno por materia, a lo largo de un único corredor que empieza en el mostrador: para llegar al tercer pasillo, el robot pasa por los dos primeros. Cada libro pertenece a un pasillo. El robot tiene una cesta en la que caben tres libros. La tarde termina cuando cada libro está en su estantería y el robot ha vuelto al mostrador.

Modélalo como un problema clásico. Tu prueba típica debería tener más libros de los que caben en la cesta; tus pruebas de la trampa, un cuarto libro en la cesta y un libro dejado en el pasillo equivocado.

Ejercicio 19.2.2. *La planta del hotel.* Dos limpiadoras arreglan las habitaciones de una planta de un hotel. Hay que aspirar todas las habitaciones, y la planta tiene una sola aspiradora, guardada en el cuarto de material, donde debe estar al final. En las habitaciones cuyo huésped se ha ido hoy se cambia también la ropa de cama; la ropa limpia está en el cuarto de material, y una limpiadora lleva un juego cada vez. En una habitación con el cartel de «no molestar» no se entra.

Modélalo como un problema clásico con las condiciones del Capítulo 5; enuncia el objetivo como una regla sobre todas las habitaciones, no como una lista. Busca una situación que la historia no resuelva, y conviértela en tu prueba «sin plan». Cuando tengas el modelo, lee el mejor plan para dos limpiadoras y explica qué hace la segunda.

Ejercicios de la Sección 19.3 — Qué característica para qué necesidad

Ejercicio 19.3.1. *El camión cisterna.* En verano, un camión cisterna riega los parterres de un pueblo. Los parterres están en unos pocos lugares unidos por carreteras de longitud conocida. Cada parterre necesita una cantidad conocida de agua, el depósito admite 1000 litros y se puede rellenar en las bocas de riego, que están en algunos de los lugares. El camión sale lleno de la cochera y vuelve allí. El pueblo quiere todos los parterres regados con los menos metros recorridos posibles.

Usa la guía de decisión para elegir las características, y anota a qué preguntas de la Figura 19.2 respondiste con «no». Decide si un parterre se puede regar en dos visitas.

Ejercicio 19.3.2. *La red de agua.* Un fontanero debe reparar las juntas que pierden agua en la red de un pueblo. El agua sale del depósito por tuberías, cada una con una válvula, y llega a cada junta conectada con el depósito por válvulas abiertas. Una junta solo se puede reparar cuando está seca. La red tiene anillos, así que una junta puede recibir agua por dos lados. El consultorio médico debe tener

agua en todo momento durante el trabajo. Al final, las válvulas que estaban abiertas vuelven a estar abiertas.

El interés de este proyecto está en lo que *no* es efecto de ninguna acción. Incluye una red en la que la reparación obvia en tres pasos rompe la regla del consultorio.

Ejercicio 19.3.3. *La escala.* Un avión que llega a su puerta de embarque debe volver a salir. Los pasajeros desembarcan (10 minutos); después se limpia la cabina (15) y se carga el *catering* (12); se reposta el avión (20), nunca con pasajeros a bordo; y cuando la cabina está limpia y con el *catering* cargado, embarcan los pasajeros nuevos (15). El aeropuerto tiene un equipo de limpieza y un camión de combustible para todas las puertas, y un camión de *catering* para cada avión. La medida es el instante en que el último avión está listo.

Calcula a mano la escala más corta de un avión, y la de dos que llegan juntos, antes de ejecutar tu modelo.

Ejercicio 19.3.4. *La calzada.* Una furgoneta reparte paquetes en una isla unida a tierra firme por una calzada. La calzada solo está seca con la marea baja: hoy, desde el comienzo del día hasta el minuto 60, y del minuto 300 al minuto 360. Cruzar lleva 20 minutos, y entregar un paquete, 15. El turno del conductor termina en el minuto 480, y la furgoneta debe estar de vuelta en tierra firme con todos los paquetes entregados.

Decide qué debe cumplirse durante un cruce y durante una entrega. Entre tus pruebas, incluye una cuya respuesta sea «sin plan» y que tu planificador demuestre, e intenta una que no pueda resolver.

Ejercicio 19.3.5. *El robot aspirador.* Un robot aspirador limpia un piso: un recibidor que da a una cocina, un salón y el cuarto del bebé, con la base de carga junto al recibidor. Limpiar una habitación gasta una parte de la batería que depende de la habitación, moverse entre habitaciones un poco más, y la batería no basta para todo el piso. A la familia le importa sobre todo la cocina, luego el salón, luego el cuarto del bebé y luego el recibidor. Dos reglas de la casa: el recibidor nunca se limpia antes que la cocina, porque las migas acaban ahí; y el robot entra en el cuarto del bebé una vez como mucho, porque el bebé está dormido. El robot debe terminar en su base.

Separa lo obligatorio, lo deseado y lo que es una regla, y da a cada deseo un peso con su razón.

Ejercicio 19.3.6. *La esclusa.* Un barco que está en el tramo bajo de un canal debe llegar al tramo alto, tres metros más arriba, a través de una esclusa: una cámara con una compuerta en cada extremo. El esclusero puede abrir las válvulas de la compuerta de arriba, y la cámara se llena a medio metro por minuto hasta que su nivel es el del tramo alto; o las de la compuerta de abajo, y se vacía del mismo modo. Una compuerta solo se puede abrir con el mismo nivel de agua a ambos lados, y las dos compuertas nunca están abiertas a la vez. Al final, el barco está arriba y las dos compuertas están cerradas.

Decide quién causa cada cambio. Incluye una prueba en la que la cámara empiece llena, con la compuerta de arriba abierta, y otra que intente abrir una compuerta demasiado pronto.

Ejercicios de la Sección 19.7 — Hacer crecer un modelo por versiones

Ejercicio 19.7.1. *La lavandería, con monedas.* La lavandería de este capítulo cobra cada ciclo: 3 monedas un lavado y 2 un secado. Una segunda lavadora, de 12 kg, cuesta 5 monedas por lavado. Haz crecer la versión 2 de la lavandería hasta una versión que encuentre la manera más barata de hacer las tres coladas de su prueba típica, con y sin la segunda lavadora.

Sigue las reglas de la Sección 19.7: di qué cambia en la ficha, traduce las pruebas de la versión 2 y comprueba sus veredictos, y añade las pruebas que pide el Capítulo 9. Después di a qué versión posterior del capítulo añadirías a continuación las monedas, y qué esperas que le pase a la métrica.

Caso de estudio: una misión robótica, de la historia a la ejecución

20.

Hasta ahora, cada capítulo partía de una característica del lenguaje y buscaba las situaciones que la necesitan. El trabajo real va en sentido contrario. Alguien nos da una página escrita por una persona que nunca ha oído hablar de PDDL, con robots, habitaciones, horarios de apertura, cajas urgentes y algún «si es posible», y nos pide un sistema que diga a los robots qué hacer mañana por la mañana. Nada en la página dice qué requisito declarar. El Capítulo 19 reunió todo el método MODEL en un solo sitio; lo que aún falta es verlo llevar un problema de ese tamaño desde la primera lectura hasta un plan que ejecuta un robot.

Eso hace este capítulo con el mundo del mensajero a tamaño real: un día de una flota de hospital, con dos mensajeros, baterías, el horario de apertura de la farmacia, una entrega urgente, otras rutinarias, el laboratorio estéril y algunos deseos. Llevamos la página por los cinco pasos y por cuatro versiones del modelo, cada una un dominio probado con su batería de pruebas: una clásica, una temporal y numérica, una con calendario y deseos, y una variante PDDL+ de la batería. Cada salto se da porque la versión anterior no puede decir algo que dice la página, y en cada uno mostramos las alternativas que escribimos, ejecutamos y descartamos. Dos veces el planificador no consigue hacer lo que el modelo le pide dentro de nuestro límite de tiempo; lo decimos, y mostramos qué hace entonces un modelador.

El capítulo tiene dos mitades. Las Secciones 20.1 a 20.6 construyen las cuatro versiones. Las Secciones 20.7 a 20.9 dejan el planificador y siguen el plan hasta el robot: qué es un plan para un ejecutor, qué puede salir mal, cuándo planificar de nuevo, cómo hace todo esto PlanSys2 en ROS 2 y qué deja fuera PDDL. Los ejercicios son proyectos.

20.1 La historia 472
20.2 Paso M: mapear la situación 473
Un plan para el modelado 474
20.3 Versión 1: quién lleva qué 475
Pasos O y D 475
Paso E: la batería de pruebas 477
Paso L: tres maneras de escribir la regla del laboratorio 478
Lo que la versión 1 no puede decir 479
20.4 Versión 2: tiempo y batería 480
Pasos O y D: lo que cambia 480
Paso E: la ronda en el tiempo 481
Paso L: cuánto dura una carga 483
Lo que la versión 2 no puede decir 485
20.5 Versión 3: un calendario y deseos 485
Pasos O y D: lo que cambia 485
Paso E, primera parte: las pruebas que pasan . . . 486
Paso E, segunda parte: la ronda, y un límite 487
Paso L: tres maneras de planificar los deseos . . 488
Lo que la versión 3 no puede decir 491
20.6 Versión 4: la batería es cosa del mundo 491
Pasos O y D: lo que cambia 491
Paso E: un robot y una persiana 492
Paso L: ¿necesita la ronda la versión 4? 495
20.7 Del plan a la ejecución 496
Qué es un plan para un robot 496
Qué puede salir mal . . 496
Planificar de nuevo . . . 497
20.8 PlanSys2 498
Cuatro nodos 498
Un plan se convierte en un árbol de comportamiento 499
Ejecutantes de acciones 500
La aplicación 500
Lo que esto pide al modelo 501
20.9 Lo que PDDL deja fuera 502
20.10 El método MODEL: una

20.1. La historia

Esta es la página. La escribió la responsable de logística de la planta de hospital que conocemos desde el Capítulo 1; solo hemos añadido números donde ella decía «un rato».

La ronda de la mañana, según la describe la responsable de logística

Tenemos dos robots mensajeros, `r1` y `r2`. Duermen en la base de carga, donde hay un cargador. Por la mañana `r1` está cargado del todo; `r2` hizo los recados de la noche y empieza con más o menos un tercio de la batería. Un robot gasta más o menos una unidad de batería por metro y le caben cien. En el cargador gana cinco unidades por segundo, y un robot que se enchufa se queda hasta llenarse: a los contactos no les sienta bien que los enchufen y desenchufen a cada rato.

Hay cuatro peticiones esperando. Una caja de medicamentos para la UCI, que es *urgente*: tiene que estar allí 80 segundos después de que empiece la ronda. Una caja rutinaria de medicamentos para la sala A. Una muestra que han tomado las enfermeras de la sala B y que el laboratorio tiene que analizar. Y una comida que la cocina ha dejado en el vestíbulo para la sala B. La farmacia aún no está abierta. Abre a los 30 segundos de empezar la ronda y cierra a los 90, y nadie da una caja a un robot a través de una persiana cerrada. El laboratorio es estéril: una muestra solo entra en una bolsa sellada;

los robots llevan bolsas y pueden sellar una en unos segundos. Las cosas solo se entregan donde van; no quiero cajas abandonadas en los pasillos. La ronda termina a los 100 segundos, cuando el turno de día ocupa los pasillos. Lo que no esté entregado para entonces espera a la ronda siguiente. La caja urgente no se negocia. Después de ella, lo que más me importa son los otros medicamentos, luego la muestra y luego la comida. Y me gusta encontrarme los robots de vuelta en la base de carga, si es posible.

La Figura 20.1 dibuja la situación al empezar la ronda. Los tiempos son cortos (una ronda real dura horas, no segundos) porque conservamos las distancias y la velocidad de los capítulos anteriores, 1 metro por segundo; multiplicar todos los números por sesenta no cambia nada de lo que sigue.

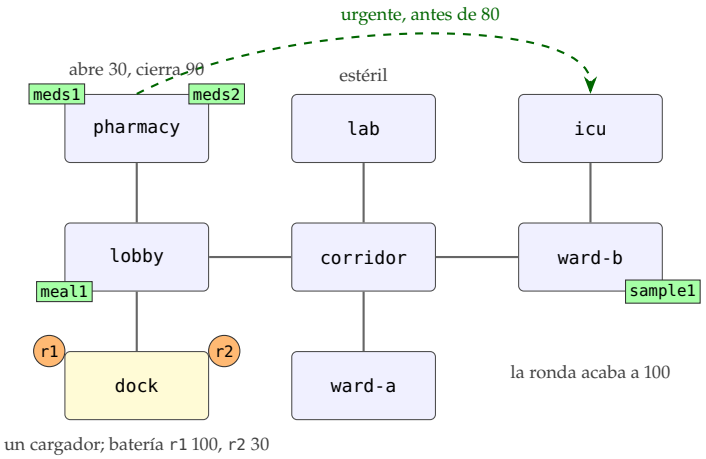


Figura 20.1.: La ronda de la mañana. Dos robots en la base de carga; `meds1` (urgente, para la UCI) y `meds2` (para la sala A) en la farmacia, `sample1` en la sala B para el laboratorio, `meal1` en el vestíbulo para la sala B. Las notas son lo que el mundo del mensajero clásico nunca tuvo que manejar a la vez.

20.2. Paso M: mapear la situación

El paso M se hace con un lápiz y, cuando se puede, con la autora de la página delante. Una primera lectura deja preguntas que solo ella puede responder, y cada respuesta es una línea de la ficha del modelo. Estas son las que le hicimos.

- *¿Puede un robot llevar dos cosas?* No: tiene un solo compartimento. (Un supuesto, como en el Capítulo 2.)
- *¿Puede un robot dejar el cargador antes de llenarse?* No, ya lo había dicho. Esta frase decidirá una codificación en la Sección 20.4.
- *¿Puede un robot pasar una caja al otro, o dejarla en algún sitio por el camino?* No: «nada de cajas en los pasillos». El relevo que inventó el planificador en el Capítulo 10 está prohibido aquí.
- *¿Qué pasa si una muestra entra sin sellar en el laboratorio?* «No puede pasar.» No dijo que la muestra se pierda; dijo que eso no se hace.
- *«Si es posible»: ¿comparado con qué?* Un robot en la base de carga vale menos que cualquier entrega. Una entrega después del final de la ronda no vale nada.
- *¿Gasta batería un robot mientras espera?* «Un poco.» Al principio lo dejaremos fuera y volveremos a ello en la Sección 20.6.

El paso M sobre una página real es una conversación: cada pregunta a la autora de la historia se convierte en una línea de la ficha del modelo.

Paso M: la ficha del modelo de la ronda de la mañana

Historia	Dos robots mensajeros atienden las peticiones de una ronda en una planta de hospital: una caja urgente, entregas rutinarias, una muestra para un laboratorio estéril.
Agentes	Los robots `r1` y `r2`. Actúan a la vez; cada uno hace una cosa cada vez. La farmacia, el cargador y el reloj no son agentes: siguen un horario o una ley.
Éxito	Obligatorio: `meds1` entregada en la UCI antes de 80. Deseado (tabla de abajo): las demás entregas, antes de que acabe la ronda a 100; los robots en la base de carga.
Alcance	Dentro: habitaciones, conexiones y distancias; dónde están los robots y los artículos; qué lleva un robot; el sellado; duraciones; batería y carga; horarios de apertura y plazos. Fuera: la ruta dentro de una habitación, las personas en los pasillos, cómo agarra el brazo, los fallos, todo lo incierto.
Supuestos	Un artículo por robot. Los artículos solo se entregan en su destino. Un robot en el cargador, y se queda hasta llenarse. Batería: 1 por metro, nada parado (revisado en la versión 4). El horario se conoce de antemano: farmacia abierta de 30 a 90, plazo urgente 80, fin de la ronda 100.

Deseo	**Condición**	**Peso**	**Por qué**
`meds2-done`	`meds2` entregada en la sala A	5	primero, los medicamentos
`sample-done`	`sample1` entregada en el laboratorio	4	más que la comida; menos que los medicamentos
`meal-done`	`meal1` entregada en la sala B	2	puede esperar una ronda
`r1-home, r2-home`	el robot acaba en la base de carga	1 cada uno	menos que cualquier entrega

La ficha tiene los cinco apartados del Capítulo 1 y la tabla de *Deseos* que añadió el Capítulo 14. Su línea de Supuestos ya contiene un horario, como pide el Capítulo 13. Los pesos son el orden de la responsable convertido en números ($5 > 4 > 2 > 1$), con una propiedad que comprobamos con ella: nunca se renuncia a una entrega para llevar los robots a casa, porque hasta la comida vale tanto como los dos robots en la base de carga.

Un plan para el modelado

La ficha necesita tiempo, números, un calendario y preferencias. Escribirlo todo de una vez es la manera más segura de acabar con un modelo que no funciona y sin idea de por qué. El Capítulo 19 dio la alternativa: hacer crecer el modelo por versiones, cada una completa, probada y un poco más fiel que la anterior, y pasar a la siguiente solo cuando la versión actual no puede decir algo que importa (Figura 20.2).

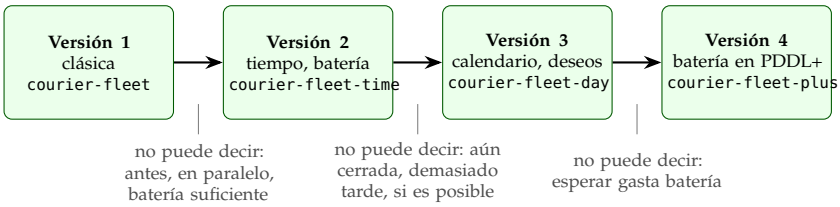


Figura 20.2.: Las cuatro versiones del caso y, bajo cada salto, lo que la versión de su izquierda no puede decir. Cada versión conserva su batería de pruebas, y cada plan de una versión posterior se compara con el plan de la anterior.

20.3. Versión 1: quién lleva qué

La primera versión es clásica. Su Alcance es la primera línea del Alcance de la ficha y nada más: habitaciones, dónde están las cosas, qué se lleva, el sellado, y las dos reglas sobre la entrega y sobre el laboratorio. Ni tiempo, ni batería, ni deseos: cada entrega es un objetivo.

Pasos O y D

Partimos del dominio con tipos del Capítulo 4 y de la muestra del Capítulo 6. La Tabla 20.1 es la tabla de estado; las cuatro últimas filas son nuevas respecto al mundo del mensajero del Capítulo 4.

Propiedad	Sobre	Clase	Codificada como
Se puede ir de una habitación a otra	dos lugares	estática	`(connected ?from ?to)`
Dónde está un robot	robot, lugar	dinámica	`(robot-at ?r ?l)`
Dónde está un artículo	artículo, lugar	dinámica	`(item-at ?i ?l)`
Qué artículo lleva un robot	robot, artículo	dinámica	`(carrying ?r ?i)`
El robot no lleva nada	robot	dinámica	`(free ?r)`
Adónde debe ir un artículo	artículo, lugar	estática	`(destination ?i ?l)`
El artículo se ha entregado	artículo	dinámica	`(delivered ?i)`
La habitación tiene esclusa	lugar	estática	`(sterile ?l)`
La muestra va en bolsa sellada	muestra	dinámica	`(sealed ?s)`

Tabla 20.1.: Paso O para la versión 1. Tipos: `robot`, `location`, `item`, y `medicine`, `sample`, `meal` bajo `item`, como en el Capítulo 4.

En la tabla se ven dos decisiones de la ficha. `destination` es una tabla estática en el sentido del catálogo de patrones del Capítulo 7: las peticiones cambian de un problema a otro, así que son datos y no parte del dominio. Y no hay ningún hecho para «contaminada»: la frase de la responsable era una prohibición, así que el estado malo será inalcanzable y no necesita nombre.

Acción	Parámetros	Se aplica cuando	Cambia
`move`	robot, origen, destino	el robot está en el *origen*; las habitaciones están conectadas; si el *destino* es estéril, toda muestra que lleva el robot está sellada	el robot está en el *destino* y ya no en el *origen*
`pick`	robot, artículo, lugar	los dos están en el lugar; el robot está libre; el artículo aún no se ha entregado	el robot lleva el artículo; el artículo ya no está allí; el robot no está libre
`seal`	robot, muestra	el robot lleva la muestra; no está sellada	la muestra está sellada
`deliver`	robot, artículo, lugar	el robot está allí y lleva el artículo; el lugar es el destino del artículo	el artículo queda allí y está entregado; el robot está libre

Tabla 20.2.: Paso D para la versión 1. No hay `drop`: `deliver` es un drop que solo se aplica en el destino.

Tres filas de la Tabla 20.2 vienen de la conversación del paso M. «Nada de cajas en los pasillos» eliminó una acción: con `deliver` en lugar de `drop`,

un artículo que se recoge solo puede dejarse donde va. Es la acción sin vuelta atrás del catálogo, elegida a propósito, y la Sección 20.7 mostrará su precio. `(not (delivered ?i))` en `pick` impide que un robot vuelva a coger un artículo ya entregado. Y la regla del laboratorio es una precondición de `move`. El Listado 20.1 es el dominio.

```
;; The hospital fleet, version 1 (classical): who carries what,
;; where to, and the rule of the sterile lab.
(define (domain courier-fleet)
  (:requirements :strips :typing :negative-preconditions
                 :disjunctive-preconditions
                 :universal-preconditions)

  (:types robot location item - object
          medicine sample meal - item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (destination ?i - item ?l - location) ; static: the request
    (delivered ?i - item)
    (sterile ?l - location)       ; static: a room with airlock
    (sealed ?i - sample))         ; the sample is in a sealed bag

  ; Nobody enters a sterile room with a sample that is not
  ; sealed: the rule of the lab, asked of every move.
  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and
      (robot-at ?r ?from) (connected ?from ?to)
      (or (not (sterile ?to))
          (forall (?s - sample)
            (or (not (carrying ?r ?s)) (sealed ?s)))))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to)))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r)
                       (not (delivered ?i)))
    :effect (and (not (item-at ?i ?l)) (not (free ?r))
                 (carrying ?r ?i)))

  ; Put the carried sample in a sealed bag.
  (:action seal
    :parameters (?r - robot ?s - sample)
    :precondition (and (carrying ?r ?s) (not (sealed ?s)))
    :effect (sealed ?s))

  ; Items are handed over at their destination and nowhere else:
  ; there is no drop.
  (:action deliver
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i)
                       (destination ?i ?l))
    :effect (and (not (carrying ?r ?i)) (free ?r)
                 (item-at ?i ?l) (delivered ?i))))
```

Listado 20.1: Versión 1: el dominio clásico de la flota.

La condición de las líneas 28 a 30 se lee así: o la habitación en la que se entra no es estéril, o cada muestra o no la lleva este robot o está sellada. Necesita `:disjunctive-preconditions` y `:universal-preconditions` (Capítulo 5). El `forall` está ahí aunque un robot lleve un solo artículo: la regla no debería depender del supuesto sobre el compartimento.

Paso E: la batería de pruebas

La prueba típica es la propia ronda, `courier-fleet-p02.pddl`: las cuatro entregas y los dos robots de vuelta en la base de carga.

```
    (sterile lab)
    (robot-at r1 dock) (free r1)
    (robot-at r2 dock) (free r2)
    (item-at meds1 pharmacy) (destination meds1 icu)
    (item-at meds2 pharmacy) (destination meds2 ward-a)
    (item-at sample1 ward-b) (destination sample1 lab)
    (item-at meal1 lobby) (destination meal1 ward-b))
  (:goal (and (delivered meds1) (delivered meds2)
              (delivered sample1) (delivered meal1)
              (robot-at r1 dock) (robot-at r2 dock))))
```

```
$ kairos courier-fleet-domain.pddl courier-fleet-p02.pddl
; cost = 35
...
; better plan 2
; cost = 31
(move r2 dock lobby)
(move r2 lobby pharmacy)
(pick r2 meds2 pharmacy)
(move r2 pharmacy lobby)
(move r2 lobby corridor)
(move r2 corridor ward-a)
(deliver r2 meds2 ward-a)
(move r2 ward-a corridor)
(move r2 corridor lobby)
(pick r2 meal1 lobby)
...
(move r2 icu ward-b)
(pick r2 sample1 ward-b)
(move r2 ward-b corridor)
(seal r2 sample1)
(move r2 corridor lab)
(deliver r2 sample1 lab)
(move r2 lab corridor)
(move r2 corridor lobby)
(move r2 lobby dock)
```

Leamos el plan como lo leería la responsable. La muestra se sella en el pasillo, un movimiento antes del laboratorio: correcto. Cada artículo acaba donde iba. Y luego, dos cosas que ella no aceptaría. La caja urgente es la tercera de las cuatro entregas. Y las 31 acciones son todas de `r2`: `r1` no sale nunca de la base de carga. Con `-v`, Kairos informa de `solved (proven optimal), cost 31`: ningún plan tiene menos acciones, y usar el segundo robot solo añadiría su viaje de ida y vuelta. El modelo cuenta acciones, así que un robot que lo hace todo es el mejor plan posible.

Las otras tres pruebas:

```
$ kairos courier-fleet-domain.pddl courier-fleet-p01.pddl
; cost = 5
(pick r1 meds2 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(deliver r1 meds2 ward-a)
$ kairos courier-fleet-domain.pddl courier-fleet-p03.pddl
note: the goal needs atoms that no action can reach
(delivered meds1) can never be reached:
  deliver adds it, but is never applicable (types, negative or
```

```
     numeric conditions) [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
$ kairos courier-fleet-domain.pddl courier-fleet-p04.pddl
note: no plan: every state reachable from the initial one was
     searched (22 states) and none satisfies the goal. [orchestrator.
     why]
kairos: unsolvable (exhausted)
```

`p03` es la ronda con la puerta entre la sala B y la UCI cerrada por limpieza. `p04` es la prueba de la trampa, en la forma que el Capítulo 2 llamó «pedir el estado prohibido»: el objetivo `(and (delivered sample1) (not (sealed sample1)))` solo se alcanza entrando en el laboratorio con una muestra abierta. La segunda forma de la prueba de la trampa es un plan escrito a mano, el que seguiría un robot perezoso, que se da al validador:

```
$ kairos validate courier-fleet-domain.pddl courier-fleet-p05.pddl
     courier-fleet-p05-noseal.plan
courier-fleet-p05-noseal.plan:3: error: step 3 (move r1 corridor lab)
      at 2.000: precondition false: (or (not (sterile ?to)) (forall
     (?s - sample) (or (not (carrying ?r ?s)) (sealed ?s)))) with ?r
      = r1 ?from = corridor ?to = lab is false [validate.
     precondition_failed]
 3 | (move r1 corridor lab)
plan invalid
```

Paso E: la batería de pruebas de la versión 1

Prueba	Fichero	Esperado y obtenido
Trivial	`courier-fleet-p01`	5 pasos: pick, tres move, deliver
Típica	`courier-fleet-p02`	31 pasos, óptimo demostrado; muestra sellada antes del laboratorio
Sin plan	`courier-fleet-p03`	irresoluble: no se puede llegar a la UCI
Trampa	`courier-fleet-p04, p05`	irresoluble; el plan sin `seal` se rechaza

Paso L: tres maneras de escribir la regla del laboratorio

El libro ha enseñado tres codificaciones de «ninguna muestra sin sellar en el laboratorio», y escribimos las tres como dominios con el mismo nombre, para que los mismos problemas funcionen con cada uno.

A, la física (`courier-fleet-when-domain.pddl`). La codificación del Capítulo 6: `move` siempre es posible, y un efecto condicional marca como `contaminated` una muestra sin sellar que entra en una habitación estéril; `deliver` rechaza esos artículos.

B, una precondición (Listado 20.1). El movimiento hacia el laboratorio no es aplicable.

C, una restricción (`courier-fleet-rule-domain.pddl`). `move` no tiene ninguna condición sobre muestras, y el dominio enuncia la regla una sola vez, sobre todos los estados del plan, como enseñó el Capítulo 15:

```
; Never in a sterile room with a sample that is not sealed.
(:constraints
  (always (forall (?r - robot ?s - sample ?l - location)
    (imply (and (carrying ?r ?s) (robot-at ?r ?l) (sterile ?l))
           (sealed ?s)))))
```

Las tres dan un plan de 31 pasos para la ronda, óptimo demostrado, responden `unsolvable` a `p04` y rechazan el plan sin `seal`, cada una con sus propias palabras: una precondición falsa de `move` (B), una precondición falsa de `deliver`, porque `(contaminated sample1)` es cierto (A), y `the plan violates the domain :constraints` (C). La Tabla 20.3 las compara; los números son los de `kairos -v` sobre la ronda.

Tabla 20.3.: Paso L para la versión 1: tres codificaciones de la regla del laboratorio, ejecutadas sobre `courier-fleet-p02`.

	A: efecto condicional	B: precondición	C: restricción
Fidelidad	Modela lo que hace la esclusa. Admite planes que estropean una muestra que nadie pidió entregar	Dice lo que dijo la responsable: no se hace. Solo vigila la puerta (ver abajo)	Lo dice para cada estado, sea cual sea la acción que lleve a él
Tamaño	54 acciones instanciadas, 40 hechos	80 acciones instanciadas (el `or` divide `move`), 41 hechos	54 acciones instanciadas, 3 axiomas añadidos por el planificador
Guía	10 749 estados; una muestra contaminada es un callejón sin salida que se descubre tarde	10 432 estados; ningún callejón sin salida	10 749 estados
Claridad	La regla está repartida entre un efecto, un hecho y una precondición	Una condición, en la acción que podría romper la regla	Una línea, con las palabras de la responsable
Portabilidad	`:conditional-effects`	`or`, `forall`: precondiciones ADL	`:constraints`: pocos planificadores

Veredicto. En Tamaño y Guía, las tres están a pocos puntos porcentuales: a esta escala no deciden. Descartamos A por Fidelidad: describe un mundo en el que las muestras se estropean, y la ficha describe un hospital en el que eso no se hace; su hecho de más y su callejón sin salida no aportan nada. Entre B y C se aplica al pie de la letra el consejo del Capítulo 15: la restricción es la especificación, y la precondición es la codificación que conviene usar cuando la regla trata de un solo estado, pocas acciones pueden romperla y el modelo tiene que viajar. El nuestro tiene que viajar: las versiones siguientes añaden tiempo, y la Sección 20.8 lleva el dominio a un ejecutor que comprueba las condiciones de cada acción mientras se ejecuta. Nos quedamos con B, y conservamos C como prueba: cualquier plan de B se puede dar a `kairos validate` con el dominio de C.

Merece la pena ejecutar esa prueba con imaginación. B vigila la puerta del laboratorio; C vigila la habitación. Difieren para una muestra que ya está dentro, sin sellar, y que un robot recoge allí. La ficha dice que las muestras van *al* laboratorio, así que la ronda nunca contiene ese estado y B es correcta para nuestro Alcance; el Ejercicio 20.3.2 construye el problema que las distingue.

Una regla de la historia tiene tres codificaciones: física con un resultado malo, una precondición, una restricción. Escribe la restricción como especificación aunque el modelo use la precondición.

Lo que la versión 1 no puede decir

La versión 1 responde bien a una pregunta: qué robot tiene que estar dónde y con qué, y cuándo sellar. Calla sobre tres cosas que dice la ficha.

No puede decir *antes*: su mejor plan deja un robot en la base de carga, y nada en un recuento de acciones premia trabajar en paralelo. No puede decir *batería suficiente*: `r2` recorre 190 metros en ese plan, con batería para 30. Y no puede decir *urgente*. Las dos primeras necesitan duraciones y números, y son la versión siguiente.

20.4. Versión 2: tiempo y batería

La versión 2 añade al Alcance: duraciones, robots que trabajan a la vez, batería y carga. Es el mensajero del Capítulo 10 y del Capítulo 11 con las reglas de la versión 1.

Pasos O y D: lo que cambia

La tabla de estado gana un hecho y un cerrojo, y seis números (Tabla 20.4). Todas las acciones pasan a ser durativas, y cada una toma el testigo `(idle ?r)` al empezar y lo devuelve al terminar, que es como el Capítulo 10 impide que un robot haga dos cosas a la vez.

Propiedad	Clase	Codificada como
El robot no está haciendo nada	dinámica	`(idle ?r)`
La habitación tiene cargador; no hay nadie enchufado	estática; dinámica	`(has-charger ?l), (charger-free ?l)`
Metros entre habitaciones conectadas; velocidad	estática	`(distance ?from ?to), (speed ?r)`
Carga, de 0 a la capacidad	dinámica; estática	`(battery ?r), (battery-capacity ?r)`
Batería por metro; batería por segundo en el cargador	estática	`(consumption ?r), (charge-rate ?r)`

Tabla 20.4.: Paso O para la versión 2: lo que se añade a la Tabla 20.1. Los nombres son los del Capítulo 8, el Capítulo 10 y el Capítulo 11.

Para cada acción respondemos a las cuatro preguntas que el Capítulo 10 añadió al paso D: cuánto dura, qué tiene que cumplirse y cuándo, cuándo ocurre cada efecto, qué puede ejecutarse en paralelo (Tabla 20.5).

Acción	Duración	Lo nuevo con el tiempo
`move`	distancia / velocidad	La batería de todo el tramo se exige y se descuenta al empezar. El robot no está en ningún sitio mientras se mueve
`pick`	2 s	El robot sigue en la habitación (`over all`). El artículo deja el estante al empezar y está cargado al terminar
`seal`	3 s	La muestra queda sellada al terminar
`deliver`	2 s	El robot sigue en la habitación. Entregado, y el robot libre, al terminar
`charge`	batería que falta / ritmo de carga	Toma el cargador al empezar y lo devuelve al terminar. La batería está llena al terminar. El robot sigue en la habitación

Tabla 20.5.: Paso D para la versión 2. Todas las acciones toman `(idle ?r)` al empezar y lo devuelven al terminar.

Los capítulos anteriores usaban tiempos de manipulación distintos: una función `handling-time` con valor 2 en el Capítulo 10, una constante 2 en el Capítulo 12 y el Capítulo 13, 1 en el Capítulo 18. Tomamos 2 segundos para `pick` y `deliver`, como constantes: nada en la historia hace que un robot sea más rápido que otro en eso. Los movimientos y la carga están en el Listado 20.2.

```
  ; The battery for the whole stretch is taken at the start.
  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
    :condition (and
      (at start (idle ?r))
      (at start (robot-at ?r ?from))
      (at start (connected ?from ?to))
      (at start (>= (battery ?r)
                    (* (distance ?from ?to) (consumption ?r))))
      (at start (or (not (sterile ?to))
                    (forall (?s - sample)
                      (or (not (carrying ?r ?s)) (sealed ?s))))))
    :effect (and
      (at start (not (idle ?r)))
      (at start (not (robot-at ?r ?from)))
      (at start (decrease (battery ?r)
                  (* (distance ?from ?to) (consumption ?r))))
      (at end (robot-at ?r ?to))
      (at end (idle ?r))))
  ; A charge always fills the battery: its length is what is
  ; missing over the rate. One robot at a charger at a time.
  (:durative-action charge
    :parameters (?r - robot ?l - location)
    :duration (= ?duration (/ (- (battery-capacity ?r)
                                 (battery ?r))
                              (charge-rate ?r)))
    :condition (and
      (at start (idle ?r))
      (at start (has-charger ?l))
      (at start (charger-free ?l))
      (at start (< (battery ?r) (battery-capacity ?r)))
      (over all (robot-at ?r ?l)))
    :effect (and
      (at start (not (idle ?r)))
      (at start (not (charger-free ?l)))
      (at end (assign (battery ?r) (battery-capacity ?r)))
      (at end (charger-free ?l))
      (at end (idle ?r)))))
```

Listado 20.2: Versión 2: `move` y `charge` (parte de `courier-fleet-time-domain.pddl`).

El cargador es el cerrojo de recurso del catálogo: `(charger-free ?l)` se borra al empezar una carga y se añade al terminarla. La batería de un movimiento se descuenta entera al empezar, como en el Capítulo 11: un cambio discreto, que basta, como mostró el Capítulo 12, cuando nadie lee la magnitud mientras la acción se ejecuta.

Paso E: la ronda en el tiempo

El problema de la ronda, `courier-fleet-time-p02.pddl`, tiene los números de la ficha y la métrica `(:metric minimize (total-time))`. Los robots no tienen por qué volver a la base de carga: eso es un deseo, y esta versión no tiene deseos. Una búsqueda temporal es *anytime* y gastaría todo su límite de tiempo, así que a partir de aquí le damos tres segundos.

```
$ kairos courier-fleet-time-domain.pddl courier-fleet-time-p02.pddl
    --timeout 3
; cost = 170.62600000000006
; makespan = 170.626
...
; better plan 2
; cost = 113.01700000000002
; makespan = 113.017
0.000: (move r1 dock lobby) [4.000]
```

```
0.000: (charge r2 dock) [14.000]
4.001: (pick r1 meal1 lobby) [2.000]
6.002: (move r1 lobby corridor) [10.000]
14.001: (move r2 dock lobby) [4.000]
16.003: (move r1 corridor ward-b) [12.000]
18.002: (move r2 lobby pharmacy) [6.000]
24.003: (pick r2 meds1 pharmacy) [2.000]
26.004: (move r2 pharmacy lobby) [6.000]
28.004: (deliver r1 meal1 ward-b) [2.000]
30.005: (pick r1 sample1 ward-b) [2.000]
32.005: (move r2 lobby corridor) [10.000]
32.006: (seal r1 sample1) [3.000]
...
55.009: (deliver r1 sample1 lab) [2.000]
57.010: (move r1 lab corridor) [8.000]
59.008: (deliver r2 meds1 icu) [2.000]
...
81.013: (pick r1 meds2 pharmacy) [2.000]
...
111.017: (deliver r1 meds2 ward-a) [2.000]
```

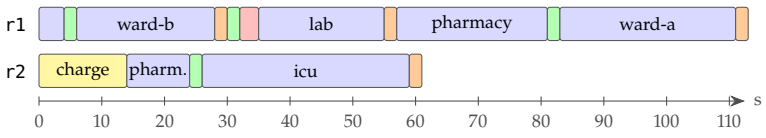


Figura 20.3.: La ronda en la versión 2 (*makespan* 113,017). Azul: movimientos, unidos y etiquetados con el lugar donde terminan; verde: pick; naranja: deliver; rojo: seal; amarillo: charge. `r2` carga durante 14 segundos y hace una entrega; `r1` hace tres.

La Figura 20.3 lo dibuja entero (el fichero `courier-fleet-time-p02.plan` tiene sus 27 líneas). Los dos robots trabajan desde el primer segundo. `r2` carga de 30 a 100 en $70/5 = 14$ segundos y lleva la caja urgente a la UCI, a 43 metros de la base de carga: llega con 57. `r1` lleva la comida, recoge la muestra donde deja la comida, la sella sin moverse de la sala B y termina con la caja rutinaria: 98 metros, con una batería de 100. También merece la pena mirar las pausas del plan: ningún robot espera entre dos de sus acciones. El makespan ha bajado de «31 acciones» a 113 segundos porque el modelo ya puede decir *antes*.

El resto de la batería de pruebas sigue al Capítulo 10 y al Capítulo 11: la prueba de la trampa se convierte en una familia, con un miembro por cada cosa que no puede solaparse.

```
$ kairos courier-fleet-time-domain.pddl courier-fleet-time-p03.pddl
note: engine M3 found no plan; as POPF, it keeps one partial plan
     per state, so this is strong evidence but not a proof [
     orchestrator.m3_exhausted]
note: no plan: the engines that apply ran out of states (20 states)
     before the time limit, so more time does not help; none of them
      is complete for this task, so a plan may still exist (see the
     note before). [orchestrator.why]
kairos: no plan found (exhausted)
$ kairos validate courier-fleet-time-domain.pddl courier-fleet-time-
     p01.pddl courier-fleet-time-p01-hurry.plan
courier-fleet-time-p01-hurry.plan:2: error: step 2 (move r1 pharmacy
      lobby) at 0.001: at start condition false: (idle r1) is false
     [validate.precondition_failed]
 2 | 0.001: (move r1 pharmacy lobby) [6.000]
plan invalid
$ kairos validate courier-fleet-time-domain.pddl courier-fleet-time-
     p04.pddl courier-fleet-time-p04-both.plan
courier-fleet-time-p04-both.plan:2: error: step 2 (charge r2 dock)
     at 0.001: at start condition false: (charger-free dock) is
     false [validate.precondition_failed]
 2 | 0.001: (charge r2 dock) [18.000]
```

```
plan invalid
```

En `p03` un robot está en la sala B con 20 de batería; la base de carga está a 26 metros y la entrega necesita 34. La respuesta es `no plan found`, con código de salida 14, y no `unsolvable`: como explicó el Capítulo 11, los motores temporales no lo demuestran. Bastaron veinte estados para agotar la búsqueda, y la demostración es nuestra propia aritmética. Los dos planes escritos a mano son los tentadores: un robot que empieza a moverse mientras aún está recogiendo, y dos robots en el único cargador. En `p04` los dos robots tienen 10 de batería y una entrega cada uno, y el plan del planificador se turna:

```
$ kairos courier-fleet-time-domain.pddl courier-fleet-time-p04.pddl
     --timeout 3
...
; better plan 3
; cost = 65.60499999999999
; makespan = 65.605
0.000: (move r2 dock lobby) [4.000]
0.000: (charge r1 dock) [18.000]
4.001: (pick r2 meal1 lobby) [2.000]
6.002: (move r2 lobby dock) [4.000]
18.001: (move r1 dock lobby) [4.000]
18.001: (charge r2 dock) [19.600]
...
58.007: (deliver r1 meds2 ward-a) [2.000]
63.605: (deliver r2 meal1 ward-b) [2.000]
```

Mientras `r1` carga, `r2` gasta 8 de sus 10 unidades en ir a por la comida al vestíbulo y está de vuelta en el cargador, cargado, en el instante en que queda libre. No habíamos pensado en ese plan, y es correcto.

Paso E: la batería de pruebas de la versión 2

Prueba	Fichero	Esperado y obtenido
Trivial	`...-time-p01`	32,004: pick, 28 m, deliver
Típica	`...-time-p02`	113,017; una carga de 14 s; las baterías acaban en 2 y 57
Sin plan	`...-time-p03`	no plan found (salida 14); a mano, 20 <26
Trampa	`...-time-p04 y dos planes`	las cargas se turnan (65,605); se rechazan un robot que hace dos cosas y dos robots en el cargador

Paso L: cuánto dura una carga

El Capítulo 11 comparó tres codificaciones de una carga y el Capítulo 12 añadió una cuarta. Escribimos tres para la flota, de nuevo como dominios con un mismo nombre. **Completa** (Listado 20.2): la carga dura lo que falta dividido por el ritmo y llena la batería. **Elegida** (`courier-fleet-time-chosen-domain.pddl`): la duración está entre 1 segundo y el tiempo hasta llenarse, y al terminar se suma `?duration` por el ritmo. **Continua**

(`courier-fleet-time-cont-domain.pddl`): la batería baja con `(* #t (drain-rate ?r))` durante un movimiento y sube con `#t` durante una carga de duración elegida. Sobre la ronda, las dos últimas devuelven el mismo plan:

```
$ kairos courier-fleet-time-chosen-domain.pddl courier-fleet-time-
    p02.pddl --timeout 3
...
; better plan 2
; cost = 107.01500000000003
; makespan = 107.015
0.000: (charge r2 dock) [1.000]
0.000: (move r1 dock lobby) [4.000]
1.001: (charge r2 dock) [1.000]
2.002: (charge r2 dock) [1.000]
3.003: (charge r2 dock) [1.000]
4.001: (move r1 lobby pharmacy) [6.000]
4.004: (move r2 dock lobby) [4.000]
...
100.014: (move r1 ward-b icu) [5.000]
105.015: (deliver r1 meds1 icu) [2.000]
```

Cuatro cargas de un segundo cada una. El Capítulo 12 nos dijo que lo esperáramos: Kairos prueba una duración elegida en sus cotas, y a un robot que necesita 46 unidades y tiene 30 le bastan cuatro veces la cota inferior. El plan es válido y seis segundos más corto que el nuestro. También enchufa el robot cuatro veces en cuatro segundos.

	Carga completa	Duración elegida	Continua
Fidelidad	Lo que dice la ficha: un robot que se enchufa se queda hasta llenarse	Tiene planes que la responsable prohibió (cargas parciales); valdría para otro cargador	Como la elegida; también valdría si alguien leyera la batería durante un movimiento
Tamaño	82 acciones instanciadas	84	84
Guía	Una carga lo decide todo. Ronda: 113,017	El planificador tiene que elegir un número y toma las cotas. Ronda: 107,015, con cuatro cargas de 1 s	El mismo plan, 107,015
Claridad	Una duración calculada y un `assign`	Dice «todo el tiempo que quieras»	La lectura más natural de una batería
Portabilidad	Acciones durativas con números	`:duration-inequalities`: Kairos, OPTIC, POPF	`:continuous-effects`: solo Kairos y POPF

Tabla 20.6.: Paso L para la versión 2: tres codificaciones de la carga, ejecutadas sobre la ronda. Las listas de planificadores son las del Capítulo 11 y el Capítulo 12.

Veredicto. La carga completa, y por un motivo que está en la historia: «un robot que se enchufa se queda hasta llenarse» convierte el plan más corto en un plan imposible, así que la codificación que en el Capítulo 11 parecía perder soluciones es aquí la fiel. Si la responsable hubiera contestado «puede irse cuando quiera», la elección sería más difícil y en ella entraría el *planificador*: un modelo cuyos buenos planes necesitan una duración estrictamente entre las cotas pide a Kairos algo que sus motores rápidos no prueban, y la respuesta práctica serían los bloques del Ejercicio 20.4.2. Es el criterio de Guía trabajando sobre una decisión real. La batería continua no aporta nada mientras nadie lea la batería durante un movimiento; alguien lo hará, en la Sección 20.6.

Lo que la versión 2 no puede decir

Vuelve a mirar la Figura 20.3 con la ficha en la mano. `r2` recoge la caja urgente a 24, en una farmacia que abre a 30. `r1` recoge la caja rutinaria a 81, bien, y la entrega a 113, después del final de la ronda. En el otro plan, la caja urgente llegó a la UCI la última de todas, a 107. La versión 2 tiene duraciones y no tiene calendario: no puede decir *aún cerrada* ni *demasiado tarde*. Y solo tiene objetivos: dale una ronda en la que no todo cabe antes de 100 y su única respuesta posible es «no hay plan», cuando la respuesta de la responsable es «pues deja la comida».

20.5. Versión 3: un calendario y deseos

La versión 3 añade las dos últimas líneas de la ficha: el horario bajo Supuestos y la tabla de deseos bajo Éxito. Son los temas del Capítulo 13 y del Capítulo 14.

Pasos O y D: lo que cambia

El dominio cambia poco. Entran dos hechos en la tabla de estado, con los nombres del Capítulo 13: `(is-open ?l)`, la habitación está abierta ahora, e `(in-time ?i)`, el plazo del artículo no ha vencido. Ninguna acción los cambia. Aun así son dinámicos, porque el mundo los cambia en instantes conocidos, y cada problema dice cuándo con literales iniciales temporizados. Dos acciones ganan una condición cada una (Listado 20.3): la habitación tiene que estar abierta durante todo el `pick`, y una entrega tiene que terminar a tiempo.

```
59  ; Somebody hands the item to the robot: the room must be open
60  ; during the whole pick.
61  (:durative-action pick
62    :parameters (?r - robot ?i - item ?l - location)
63    :duration (= ?duration 2)
64    :condition (and
65      (at start (idle ?r))
66      (at start (item-at ?i ?l))
67      (at start (free ?r))
68      (at start (not (delivered ?i)))
69      (over all (robot-at ?r ?l))
70      (over all (is-open ?l)))
71  ; A delivery counts only if it ends before the deadline.
72  (:durative-action deliver
73    :parameters (?r - robot ?i - item ?l - location)
74    :duration (= ?duration 2)
75    :condition (and
76      (at start (idle ?r))
77      (at start (carrying ?r ?i))
78      (at start (destination ?i ?l))
79      (over all (robot-at ?r ?l))
80      (at end (in-time ?i)))
```

Todo lo demás está en el problema. El calendario y los deseos de la ficha son el final de `courier-fleet-day-p02.pddl`:

```
43    ; the calendar of the day
44    (is-open lobby) (is-open ward-b)
45    (at 30 (is-open pharmacy))
46    (at 90 (not (is-open pharmacy)))
47    (in-time meds1) (in-time meds2)
48    (in-time sample1) (in-time meal1)
```

Listado 20.3: Versión 3: `pick` y `deliver` leen el calendario (parte de `courier-fleet-day-domain.pddl`).

```
    (at 80 (not (in-time meds1)))
    (at 100 (not (in-time meds2)))
    (at 100 (not (in-time sample1)))
    (at 100 (not (in-time meal1))))
  (:goal (and
    (delivered meds1)                       ; urgent: required
    (preference meds2-done (delivered meds2))
    (preference sample-done (delivered sample1))
    (preference meal-done (delivered meal1))
    (preference r1-home (robot-at r1 dock))
    (preference r2-home (robot-at r2 dock))))
  (:metric minimize (+ (* 5 (is-violated meds2-done))
                       (* 4 (is-violated sample-done))
                       (* 2 (is-violated meal-done))
                       (* 1 (is-violated r1-home))
                       (* 1 (is-violated r2-home)))))
```

El final de la ronda es el patrón del plazo usado tres veces: después de 100, la entrega de un artículo rutinario ya no es posible, y así dice el modelo que no vale nada. La caja urgente es el único objetivo estricto. Los requisitos son `:timed-initial-literals` y `:preferences`.

Paso E, primera parte: las pruebas que pasan

La primera prueba no cuesta nada: el plan de la versión 2 es un fichero, y el validador puede leerlo contra la versión 3.

```
$ kairos validate courier-fleet-day-domain.pddl courier-fleet-day-
    p02.pddl courier-fleet-time-p02.plan
courier-fleet-time-p02.plan:8: error: step 8 (pick r2 meds1 pharmacy)
     at 24.003: over-all condition broken after the happening at
    24.003000: (is-open pharmacy) is false [validate.
    invariant_failed]
 8 | 24.003: (pick r2 meds1 pharmacy) [2.000]
plan invalid
```

La versión nueva rechaza el plan antiguo por el motivo que dimos al leerlo. Es una costumbre que merece la pena mantener en cada salto: los planes de la versión n son casos de prueba de la versión $n + 1$, y el mensaje debería nombrar lo que la versión n no podía decir.

La prueba trivial tiene un robot, la caja urgente, la farmacia que abre a 30 y el plazo a 80:

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p01.pddl --
    timeout 3
; cost = 67.005
; makespan = 67.005
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
30.000: (pick r1 meds1 pharmacy) [2.000]
32.001: (move r1 pharmacy lobby) [6.000]
38.002: (move r1 lobby corridor) [10.000]
48.003: (move r1 corridor ward-b) [12.000]
60.004: (move r1 ward-b icu) [5.000]
65.005: (deliver r1 meds1 icu) [2.000]
```

El robot llega a la persiana a 10 y se queda allí veinte segundos. La caja está en la UCI a 67,005, trece segundos antes de su plazo. La trampa es recoger a 10; el plan que lo hace se rechaza con el mensaje que acabamos

de ver. Para la prueba sin plan adelantamos el plazo a 30, con la farmacia abierta desde el principio (`p03`), y Kairos lo demuestra:

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p03.pddl
note: no plan: the timed initial literals take away what the goal
     needs before it can be reached, even ignoring deletes and with
     the shortest durations: (delivered meds1) [orchestrator.
     deadlines]
kairos: unsolvable (exhausted)
```

Un problema más sutil tampoco tiene plan: la farmacia abre a 30, el plazo es a 60, y solo llegaría a tiempo una recogida antes de la apertura (`p04`). Aquí el planificador no llega a ninguna conclusión:

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p04.pddl --
     timeout 2
note: engine M3 found no plan; as POPF, it keeps one partial plan
     per state, so this is strong evidence but not a proof [
     orchestrator.m3_exhausted]
note: no plan within the time limit of 2 s (200924 states searched).
      More time may find one: --timeout (SolveOptions::timeout).
     More threads run the phases of the search at once: --threads (
     SolveOptions::threads). [orchestrator.why]
kairos: timeout (timeout)
```

La demostración de `p03` razona sobre hechos que los literales temporizados quitan; en `p04` el obstáculo es un hecho que un literal temporizado trae tarde, y además el robot puede recargar sin fin, así que la búsqueda no termina nunca. La respuesta es un límite de tiempo, código de salida 11, y completarla nos toca a nosotros: $30+2+33+2 = 67 > 60$. El Capítulo 11 dio la regla para estas pruebas, y aquí se cumple: prefiere las que tienen una respuesta que puedes calcular, y da el plan tentador al validador.

Paso E, segunda parte: la ronda, y un límite

Ahora, la ronda con los dos robots.

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p02.pddl --
     timeout 3
; cost = 13
; makespan = 67.005
...
; better plan 2
; cost = 12
; makespan = 67.005
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
30.000: (pick r1 meds1 pharmacy) [2.000]
32.001: (move r1 pharmacy lobby) [6.000]
38.002: (move r1 lobby corridor) [10.000]
48.003: (move r1 corridor ward-b) [12.000]
60.004: (move r1 ward-b icu) [5.000]
65.005: (deliver r1 meds1 icu) [2.000]
```

El plan es válido y es el plan de la prueba trivial. Su coste es 12 de un máximo de 13: la caja urgente se entrega y el único deseo concedido es `r2-home`, gracias a un robot que no se ha movido. El Capítulo 14 avisó de que una búsqueda con objetivos blandos empieza por el plan más barato de encontrar y mejora a partir de ahí. Aquí no mejora en nuestros tres

segundos. Con 25 segundos, la misma orden llegó a coste 7; sabemos, por las páginas siguientes, que 1 es posible.

Antes de cambiar nada seguimos el procedimiento de depuración del Capítulo 7, que empieza por relajar. El fichero `courier-fleet-day-p07.pddl` es la ronda sin su calendario: todo abierto, ningún plazo, los mismos deseos.

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p07.pddl -v
...
[   0.002s] [orchestrator] engine M2: 76 durative actions compressed
...
; better plan 7
; cost = 0
; makespan = 211.030
...
kairos: solved (exhausted), cost 0
```

Todos los deseos concedidos, en menos de una décima de segundo. Así que el modelo de los deseos es correcto, y el planificador maneja con soltura dos robots y cinco preferencias. Lo que no maneja en tres segundos es todo eso *con* literales temporizados. La salida detallada de la ronda nombra la diferencia: `engine M3: timed initial literals: actions must happen within time windows`. Sin calendario, Kairos usa su motor M2, que busca en el problema casi como si fuera clásico; un calendario necesita el motor que busca sobre inicios y finales (Capítulo 11), que es más lento, y las preferencias multiplican entonces un espacio que ya era grande.

Cuando el planificador decepciona, relaja el problema hasta que deje de hacerlo: la característica cuya eliminación arregla la respuesta es la que cuesta.

Esto es un hecho sobre Kairos 1.0.6 y no sobre PDDL, y otro planificador pondría el límite en otro sitio. Pero un modelo se usa con un planificador, y los criterios de Guía y Portabilidad existen para dar cabida a hechos como este. No podemos quitar el calendario. Podemos cambiar lo que pedimos en cada llamada.

Paso L: tres maneras de planificar los deseos

A, un problema. La ronda tal como está escrita: dos robots, un objetivo estricto, cinco preferencias (`p02`).

B, un robot cada vez. Dos problemas para el mismo dominio. El primero tiene a `r1` solo, la caja urgente y todos los deseos (`p05`). El segundo tiene a `r2` solo y los deseos que dejó fuera el plan de `r1` (`p06`). Los dos planes se juntan en un fichero, y el validador lo juzga contra el problema A.

C, una escalera de objetivos estrictos. Sin preferencias: planificar la caja urgente; después, la caja urgente y la rutinaria; y así, bajando por el orden de la responsable, quedarse con el último problema que tenga plan. `p08` es el tercer peldaño: las dos cajas y la muestra.

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p05.pddl --
     timeout 3
...
; better plan 4
; cost = 7
; makespan = 121.014
0.000: (move r1 dock lobby) [4.000]
4.001: (move r1 lobby pharmacy) [6.000]
30.000: (pick r1 meds1 pharmacy) [2.000]
...
```

```
65.005: (deliver r1 meds1 icu) [2.000]
67.006: (move r1 icu ward-b) [5.000]
72.007: (pick r1 sample1 ward-b) [2.000]
74.008: (move r1 ward-b corridor) [12.000]
86.009: (seal r1 sample1) [3.000]
89.010: (move r1 corridor lab) [8.000]
97.011: (deliver r1 sample1 lab) [2.000]
99.012: (move r1 lab corridor) [8.000]
107.013: (move r1 corridor lobby) [10.000]
117.014: (move r1 lobby dock) [4.000]
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p06.pddl --
     timeout 3
...
; better plan 5
; cost = 2
; makespan = 88.007
0.000: (charge r2 dock) [14.000]
14.001: (move r2 dock lobby) [4.000]
18.002: (move r2 lobby pharmacy) [6.000]
30.000: (pick r2 meds2 pharmacy) [2.000]
32.001: (move r2 pharmacy lobby) [6.000]
38.002: (move r2 lobby corridor) [10.000]
48.003: (move r2 corridor ward-a) [12.000]
60.004: (deliver r2 meds2 ward-a) [2.000]
62.005: (move r2 ward-a corridor) [12.000]
74.006: (move r2 corridor lobby) [10.000]
84.007: (move r2 lobby dock) [4.000]
```

Solo, `r1` entrega la caja urgente y la muestra y vuelve a casa; renuncia a la caja rutinaria y a la comida, coste 5 + 2 = 7, encontrado en menos de un segundo. A `r2` se le piden entonces esas dos y su propia vuelta: entrega la caja, vuelve a casa y renuncia a la comida, coste 2. El plan conjunto es `courier-fleet-day-p02-merged.plan`:

```
$ kairos validate courier-fleet-day-domain.pddl courier-fleet-day-
     p02.pddl courier-fleet-day-p02-merged.plan
plan valid, makespan 121.014, metric 2
```

Todo menos la comida. Antes de llevárselo a la responsable, leamos el plan como lo leería ella. `r1` cruza el vestíbulo a 4, donde está la comida, y luego espera veinte segundos ante la persiana de la farmacia. No tenía motivo para llevarse la comida: en su propio problema la caja urgente iba primero, y un robot que lleva la comida a la sala B no vuelve a la farmacia a tiempo para la UCI. Pero fuimos *nosotros* quienes decidimos que `r1` elegiría primero. El otro orden son dos ficheros más, `p10` (`r2` solo, con la caja urgente y todos los deseos) y `p11` (`r1` solo, con lo que queda). Solo, `r2` carga y luego hace lo que hizo `r1`: la caja urgente, la muestra y a casa, coste 7. Y `r1`:

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p11.pddl --
     timeout 3
...
; better plan 6
; cost = 1
; makespan = 90.012
0.000: (move r1 dock lobby) [4.000]
4.001: (pick r1 meal1 lobby) [2.000]
6.002: (move r1 lobby corridor) [10.000]
16.003: (move r1 corridor ward-b) [12.000]
28.004: (deliver r1 meal1 ward-b) [2.000]
30.005: (move r1 ward-b corridor) [12.000]
42.006: (move r1 corridor lobby) [10.000]
52.007: (move r1 lobby pharmacy) [6.000]
```

```
58.008: (pick r1 meds2 pharmacy) [2.000]
60.009: (move r1 pharmacy lobby) [6.000]
66.010: (move r1 lobby corridor) [10.000]
76.011: (move r1 corridor ward-a) [12.000]
88.012: (deliver r1 meds2 ward-a) [2.000]
$ kairos validate courier-fleet-day-domain.pddl courier-fleet-day-
    p02.pddl courier-fleet-day-p02-merged2.plan
plan valid, makespan 121.014, metric 1
```

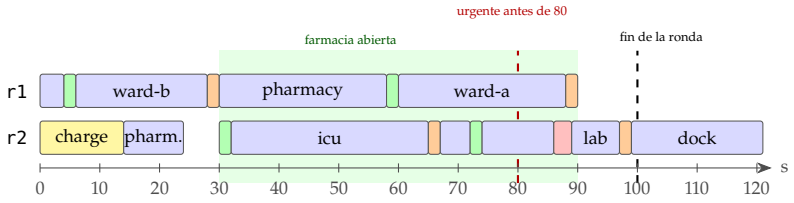


Figura 20.4.: La ronda en la versión 3 (métrica 1, makespan 121,014). Colores como en la Figura 20.3. `r1` entrega la comida antes de que abra la farmacia y la caja rutinaria después; `r2` carga, espera ante la persiana de 24 a 30 y entrega la caja urgente a 67 y la muestra a 99, un segundo antes del final de la ronda. Solo `r1` no llega a casa.

La Figura 20.4 dibuja el segundo plan conjunto. Con `r2` encargado de la caja urgente, `r1` queda libre para aprovechar el medio minuto antes de que abra la farmacia, y todas las entregas llegan a tiempo. La única pérdida es `r1-home`: `r1` acaba en la sala A con 18 de batería, a 26 metros de la base de carga. Un día con la comida, la caja rutinaria y la vuelta a casa son 108 metros, y en la batería caben 100. Creemos que 1 es el mejor coste posible; no lo hemos demostrado, y el planificador tampoco.

La escalera tuvo menos suerte. Su tercer peldaño, tres entregas obligatorias bajo el calendario, es un problema que tiene plan (cualquiera de los dos anteriores) y recibe esta respuesta:

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p08.pddl --
    timeout 2
note: no plan within the time limit of 2 s (114881 states searched).
     More time may find one: --timeout (SolveOptions::timeout).
    More threads run the phases of the search at once: --threads (
    SolveOptions::threads). [orchestrator.why]
kairos: timeout (timeout)
```

	A: un problema	B: un robot cada vez	C: escalera de objetivos estrictos
Fidelidad	Exacta: la ficha, línea a línea	Pierde los planes en los que el segundo robot debería haber elegido primero; cualquier plan que devuelve es válido para A	Pierde los pesos: solo queda un orden. Un peldaño que falla no dice nada de una mezcla más barata
Tamaño	76 acciones durativas, 5 preferencias	La mitad, dos veces	Como A sin preferencias, varias veces
Guía	Mala con calendario: coste 12 en 3 s, 7 en 25 s	Buena: cada llamada termina de mejorar en menos de un segundo. Total 2 con `r1` primero, 1 con `r2` primero	Mala: el tercer peldaño agota el tiempo
Claridad	Un fichero es toda la ronda	Dos ficheros y una fusión; quién va primero es una decisión nueva	Muchos ficheros; la respuesta queda repartida entre ejecuciones
Portabilidad	Literales temporizados junto con preferencias: pocos planificadores	Las mismas características, problemas más pequeños	Solo literales temporizados

Tabla 20.7.: Paso L para la versión 3: tres maneras de planificar la ronda de deseos, con Kairos 1.0.6 y tres segundos por llamada.

Veredicto. A es el modelo y B es cómo planificamos con él. El problema A se queda como especificación, porque es el único de los tres que dice lo que dijo la responsable, y el validador pone precio a cualquier plan contra él: `metric 2`, `metric 1`. B es una descomposición, y es honesta solo gracias a ese último paso. Un plan conjunto puede fallar donde se encuentran los robots, aquí en el único cargador, y lo que lo detectaría es el validador del problema conjunto. Y B tiene un coste, que acabamos de pagar y de recuperar: decide quién va primero antes de planificar. Nuestra primera elección tenía un motivo sensato (`r1` puede salir enseguida, `r2` tiene que cargar) y era peor en una entrega. Con dos robots hay dos órdenes, y ejecutamos los dos; con cinco hay ciento veinte, y B se convierte en una búsqueda por sí misma. Con otro planificador, o con la versión de otro año de este, puede bastar A sola; los ficheros son los mismos.

Lo que la versión 3 no puede decir

En la Figura 20.4, `r2` espera seis segundos ante una persiana cerrada, y en nuestro primer plan `r1` esperó allí veinte. En el modelo esos segundos son gratis. La responsable dijo que un robot que espera gasta «un poco». Que un poco importe o no depende de los números, y para averiguarlo necesitamos un modelo en el que la batería baje aunque nadie actúe.

20.6. Versión 4: la batería es cosa del mundo

En las versiones 2 y 3 la batería cambia cuando lo dice una acción del robot. El Capítulo 17 preguntaba de cada cambio quién lo causa, y para una batería la respuesta honesta es: el mundo. Se descarga mientras el robot está encendido, más deprisa mientras se mueve, y un cargador la llena a su propio ritmo y la suelta cuando termina. La versión 4 es la flota con la batería del Capítulo 18: el nivel que la guía de decisión de ese capítulo llama «mixto», acciones durativas junto con procesos y eventos.

Pasos O y D: lo que cambia

La Tabla 20.8 es la nueva respuesta a «quién causa cada cambio». Dos ritmos sustituyen al consumo por metro: 0,2 por segundo siempre que el robot está encendido y 0,8 más mientras se mueve, de modo que un metro sigue costando 1; el cargador da 5,2, que son 5 netos. Los números son los del Capítulo 18.

Cambio	Lo causa	En el dominio
El robot va a la habitación siguiente	el robot	acción durativa `move`; activa `(driving ?r)` mientras se ejecuta y necesita `(powered ?r)` todo el tiempo
El robot se enchufa	el robot	acción `plug`; toma `idle` y el cargador
La batería se descarga	el mundo, si está encendido	procesos `drain` y `motor-drain`
La batería se llena	el mundo, si está enchufado	proceso `charging`
El cargador lo suelta	el mundo, en un instante	evento `battery-full`; devuelve `idle` y el cargador
El robot se apaga	el mundo, en un instante	evento `shutdown`
La farmacia abre	el reloj	literal inicial temporizado

Tabla 20.8.: Pasos O y D para la versión 4: cada cambio con su causa, como pide el Capítulo 17.

```
  ; Plugging in is the whole decision: the robot is held (not
  ; idle) until the charger lets go.
  (:action plug
    :parameters (?r - robot ?l - location)
    :precondition (and (idle ?r) (robot-at ?r ?l)
                       (has-charger ?l) (charger-free ?l)
                       (< (battery ?r) (battery-capacity ?r)))
    :effect (and (not (idle ?r)) (not (charger-free ?l))
                 (plugged ?r ?l)))

  ;; --- What the world does ---
  (:process drain
    :parameters (?r - robot)
    :precondition (powered ?r)
    :effect (decrease (battery ?r) (* #t (drain-rate ?r))))

  (:process motor-drain
    :parameters (?r - robot)
    :precondition (and (powered ?r) (driving ?r))
    :effect (decrease (battery ?r) (* #t (motor-rate ?r))))

  (:process charging
    :parameters (?r - robot ?l - location)
    :precondition (plugged ?r ?l)
    :effect (increase (battery ?r) (* #t (charge-rate ?r))))

  ; The charger lets go by itself when the battery is full.
  (:event battery-full
    :parameters (?r - robot ?l - location)
    :precondition (and (plugged ?r ?l)
                       (>= (battery ?r) (battery-capacity ?r)))
    :effect (and (not (plugged ?r ?l)) (charger-free ?l)
                 (idle ?r)))

  ; An empty battery switches the robot off, wherever it is.
  (:event shutdown
    :parameters (?r - robot)
    :precondition (and (powered ?r) (<= (battery ?r) 0))
    :effect (not (powered ?r))))
```

Listado 20.4: Versión 4: enchufarse, y lo que hace el mundo (parte de `courier-fleet-plus-domain.pddl`).

La frase «un robot que se enchufa se queda hasta llenarse» ha encontrado su forma natural. No hay `unplug` ni ninguna duración que calcular: `plug` toma el testigo `idle`, y solo el evento `battery-full` lo devuelve. Un plan no menciona el evento; el evento ocurre.

Para que esta versión funcionara se dejaron fuera dos cosas, y las dos son decisiones moldeadas por el planificador. *La manipulación es instantánea*: `pick` y `deliver` son acciones normales, como en los mensajeros PDDL+ del Capítulo 17 y del Capítulo 18. Con acciones durativas, la búsqueda en pasos de tiempo seguía resolviendo las pruebas fáciles, pero no la prueba de la trampa de esta sección en cinco segundos. *No hay muestras*: el tipo, `seal` y la regla del laboratorio han desaparecido. Con la condición disyuntiva del Listado 20.1 en `move`, Kairos empieza con una fase de búsqueda que, en este dominio, no encuentra nada y gasta la mitad del límite de tiempo: la prueba trivial pasó de una centésima de segundo a tres segundos de un límite de seis. Podríamos haber reescrito la regla de otra forma. Preferimos escribir en el Alcance para qué sirve esta versión: *la versión 4 trata de la batería; el laboratorio se prueba en las versiones 1 a 3.* Una variante no tiene por qué cargar con todo.

Paso E: un robot y una persiana

El paso E de un modelo híbrido sigue los seis puntos del Capítulo 18, y el primero es el lápiz. La prueba de la trampa, `courier-fleet-plus-p04.pddl`,

tiene un robot en la puerta de la farmacia con 32 de batería; la farmacia abre a 30 y la entrega está a 28 metros. La versión 3 tiene una opinión sobre este problema, que guardamos como `courier-fleet-day-p09.pddl`: esperar en la puerta, recoger a 30 y llegar con 4 de sobra.

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-p09.pddl --
     timeout 3
; cost = 62.00399999999999
; makespan = 62.004
30.000: (pick r1 meds2 pharmacy) [2.000]
32.001: (move r1 pharmacy lobby) [6.000]
38.002: (move r1 lobby corridor) [10.000]
48.003: (move r1 corridor ward-a) [12.000]
60.004: (deliver r1 meds2 ward-a) [2.000]
```

Con lápiz: treinta segundos de espera cuestan 6, el robot sale con 26, y 28 metros necesitan 28. El mismo plan en el vocabulario de la versión 4 es `courier-fleet-plus-p04-wait.plan`, y el validador, que integra los procesos de forma exacta, coincide con el lápiz:

```
$ kairos validate courier-fleet-plus-domain.pddl courier-fleet-plus-
     p04.pddl courier-fleet-plus-p04-wait.plan
courier-fleet-plus-p04-wait.plan:5: error: step 5 (deliver r1 meds2
     ward-a) at 58.100: precondition false: (powered r1) is false [
     validate.precondition_failed]
 5 | 58.100: (deliver r1 meds2 ward-a) [0.000]
plan invalid
```

El robot se apagó a dos metros de la sala A. El planificador, con los mismos ficheros:

```
$ kairos courier-fleet-plus-domain.pddl courier-fleet-plus-p04.pddl
...
; better plan 2
; cost = 97.2
; makespan = 97.200
30.000: (move r1 pharmacy lobby) [6.000]
36.000: (move r1 lobby pharmacy) [6.000]
42.000: (pick r1 meds2 pharmacy) [0.000]
42.000: (move r1 pharmacy lobby) [6.000]
48.000: (move r1 lobby dock) [4.000]
52.000: (plug r1 dock) [0.000]
71.200: (move r1 dock lobby) [4.000]
75.200: (move r1 lobby corridor) [10.000]
85.200: (move r1 corridor ward-a) [12.000]
97.200: (deliver r1 meds2 ward-a) [0.000]
97.200: @PlanEND
```

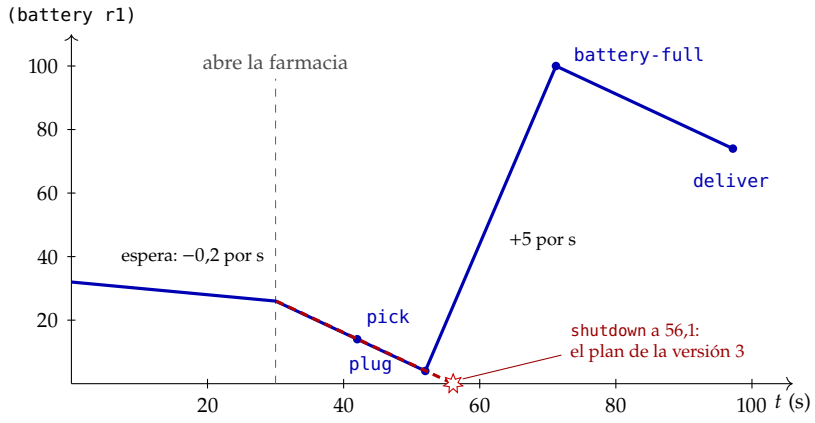


Figura 20.5.: La batería en la prueba de la trampa de la versión 4. A trazos: el plan que la versión 3 daba por bueno; el robot sale de la puerta con 26 y se apaga tras 26 de los 28 metros. Continua: el plan de la versión 4, que va a la base de carga con la caja y carga. El evento `battery-full` a 71,2 no está en el plan; el robot sigue cuando se dispara.

La Figura 20.5 reproduce los dos, como aconseja el Capítulo 18 para cualquier plan híbrido. El plan acierta donde importa: recoge la caja, la lleva a la base de carga, llega con 4, carga hasta llenarse y entrega con 74 de sobra. También es un plan con costumbres que ya conocemos. No hace nada hasta 30, porque la búsqueda deja correr el tiempo hasta lo siguiente que cambia, la apertura. Luego va al vestíbulo y vuelve antes de recoger, doce segundos que nada exige: Kairos dice en una nota que para PDDL+ *la búsqueda minimiza el número de acciones; la métrica del problema se informa para cada plan pero todavía no guía la búsqueda*. Una persona recogería a 30 y habría terminado a 85. El modelo también admite ese plan; el planificador no lo buscó.

La prueba típica es la mañana de `r2` con sus números reales (`p02`: batería 30, la caja para la sala A, vuelta a la base de carga). Su plan se enchufa a 0, el cargador lo suelta a 14, sale a 30 y está en casa a 94, con la entrega a 68. Un plan escrito a mano que sale a 14,2 es válido y diez segundos más corto (`courier-fleet-plus-p02-hand.plan`, makespan 84,1). Y la prueba sin plan, `p03`, tiene 5 de batería ante la persiana cerrada:

```
$ kairos courier-fleet-plus-domain.pddl courier-fleet-plus-p03.pddl
note: the search minimizes the number of actions; the metric of the
     problem is reported for each plan but does not guide the search
      yet [orchestrator.metric]
note: no plan in steps of time; a plan that needs finer timing than
     the steps would be missed, so this is evidence but not a proof
     [orchestrator.steps_exhausted]
note: no plan: the engines that apply ran out of states (28278
     states) before the time limit, so more time does not help; none
      of them is complete for this task, so a plan may still exist (
     see the note before). [orchestrator.why]
kairos: no plan found (exhausted)
```

Esperar vacía la batería en 25 segundos y la base de carga está a 10 metros. En la versión 3 este robot esperaría sin fin y sin coste, y solo fallaría en el primer movimiento.

Paso E: las baterías de pruebas de las versiones 3 y 4

Prueba	**Fichero**	**Esperado y obtenido**
Trivial	`...-day-p01`	67,005; la recogida espera a 30
Típica	`...-day-p02`	una llamada: coste 12. Por robot y juntando: métrica 2 (p05, p06), después 1 (p10, p11)
Sin plan	`...-day-p03, p04`	irresoluble (demostrado); límite de tiempo, y a mano 67 > 60
Trampa	`dos planes`	se rechazan la recogida temprana y el plan de la versión 2
Trivial	`...-plus-p01`	28,1: pick, 28 m, deliver
Típica	`...-plus-p02`	94; a mano 84,1, válido
Sin plan	`...-plus-p03`	no plan found (salida 14); a mano, 5/0,2 = 25 < 30
Trampa	`...-plus-p04`	97,2 con una carga; se rechaza la espera de la versión 3

Paso L: ¿necesita la ronda la versión 4?

La comparativa es entre dos niveles, y el Capítulo 18 tiene la tabla general. La nuestra trata de esta ronda (Tabla 20.9).

	Versión 3: cada movimiento descuenta batería	Versión 4: batería con procesos y eventos
Fidelidad	Esperar es gratis: admite el plan de la Figura 20.5 que acaba en un apagado	Esperar cuesta; el cargador decide cuándo termina una carga. Sin muestras, manipulación instantánea
Tamaño	76 acciones durativas para la ronda	17 acciones instanciadas para un robot, y un estado con reloj
Guía	Segundos para un robot con deseos	Menos de un segundo para un robot y un objetivo; dos robots y dos entregas no cupieron en nuestros cinco segundos; el makespan no guía
Claridad	Quien lee ve el horario en el plan	La mitad de lo que ocurre no está en el plan; hace falta una gráfica para leerlo
Portabilidad	Planificadores temporales con literales temporizados	PDDL+ con acciones durativas: Kairos

Tabla 20.9.: Paso L para la versión 4: la misma batería en dos niveles.

Veredicto. La versión 3 planifica la ronda, y la versión 4 la audita. En el plan de la Figura 20.4 los robots están parados 8 y 17 segundos, contando recogidas y entregas; a 0,2 por segundo son 1,6 y 3,4 unidades, y el robot más justo, `r2`, llega a casa con menos de 7 donde la versión 3 decía 10. La auditoría se supera, con poco margen. No sustituimos la versión 3: es la única que lleva los deseos y los dos robots dentro de nuestros límites. Usamos la versión 4 para lo que solo ella puede decir, en los trozos donde se juntan la espera y la batería, un robot cada vez; y llevamos su lección de vuelta a la versión 3 de la manera más barata posible, como un margen (guardar 10 unidades de reserva) y no como un proceso. La pregunta del Capítulo 18, *cuál es el nivel más bajo que no miente sobre aquello de lo que depende el problema*, tiene una respuesta distinta para la ronda, que depende del calendario, y para un robot aparcado delante de una persiana.

20.7. Del plan a la ejecución

Tenemos un plan: las 31 líneas de la Figura 20.4. El trabajo de un planificador termina ahí. Los robots no se han movido.

Qué es un plan para un robot

Para el planificador, un plan es una demostración: empezando en el estado inicial, bajo los supuestos del modelo, estas acciones en estos instantes alcanzan el objetivo. Para un robot es un conjunto de instrucciones, y cada línea necesita tres cosas que el modelo no contiene.

Alguien que la lleve a cabo. `(move r1 lobby pharmacy)` es una línea y seis segundos en el plan. En el robot es una tarea de navegación con un mapa, una localización, una trayectoria, obstáculos y un controlador: el nivel de detalle que el Capítulo 1 dejaba «al componente que sabe de geometría». Cada acción del dominio necesita una pieza de software que la realice e informe: hecho, fallido, cuánto lleva.

Un motivo para empezar. El plan dice `32.001`. Pero `r2` sale de la farmacia a 32,001 porque la recogida terminó a 32, no por el reloj; si la recogida tarda tres segundos, el movimiento tiene que empezar a 33. Los instantes de un plan son una manera coherente de cumplir su contenido real, que es un conjunto de dependencias: esta acción necesita lo que produce aquella, estas dos no pueden solaparse, esta no puede empezar antes de que abra la farmacia. Un ejecutor que lee el plan como un horario está **disparado por tiempo**; uno que arranca cada acción cuando han terminado las acciones de las que depende, y se cumplen sus condiciones, está **disparado por eventos**. Enviar cada acción a quien la realiza en el momento justo se llama **despacho** (*dispatching*). Las redes temporales simples del Capítulo 11 son la herramienta habitual para guardar las dependencias con sus cotas de tiempo (Dechter et al., 1991).

Un modelo que siga siendo cierto. El plan era válido en un mundo con las duraciones, los ritmos de batería y el horario del fichero de problema. El ejecutor está en el otro mundo.

Qué puede salir mal

La Tabla 20.10 enumera lo que les pasa a los planes. Nuestro plan nos dice dónde mirar. La caja urgente llega a 67 con un plazo a 80: trece segundos de holgura en 43 metros. La muestra se entrega a 99,011 y la ronda acaba a 100. El validador dice `plan valid`, y lo es; un robot que frena por una camilla en el pasillo lo vuelve falso. Un plan es tan robusto como su holgura más pequeña, y ni la métrica ni el planificador se fijaron en eso.

Qué ocurre	En la ronda	Una reacción razonable
Una acción tarda más	un movimiento de 12 s tarda 15	nada, mientras se sigan cumpliendo las dependencias y los plazos; si no, planificar de nuevo
Una acción falla	la caja se escurre al recogerla	reintentar; tras varios intentos, planificar de nuevo sin ese artículo
Una condición se rompe mientras se ejecuta una acción	la farmacia cierra antes de hora durante una recogida	parar la acción, planificar de nuevo
El estado no es el que suponía el plan	el cargador dio a `r2` 92, no 100	comprobar las condiciones de lo que queda; planificar de nuevo si falla una
La tarea cambia	una petición urgente nueva a 30	planificar de nuevo con el objetivo nuevo

Tabla 20.10.: Lo que puede salir mal entre un plan y su ejecución.

Planificar de nuevo

Casi todas las reacciones de la tabla acaban en el mismo paso: parar, mirar el mundo, escribir un problema nuevo y llamar al planificador. La **replanificación** es la reparación por defecto de la autonomía basada en modelos, y es barata precisamente por la separación del Capítulo 1: el dominio no cambia, solo el problema (Figura 20.6).

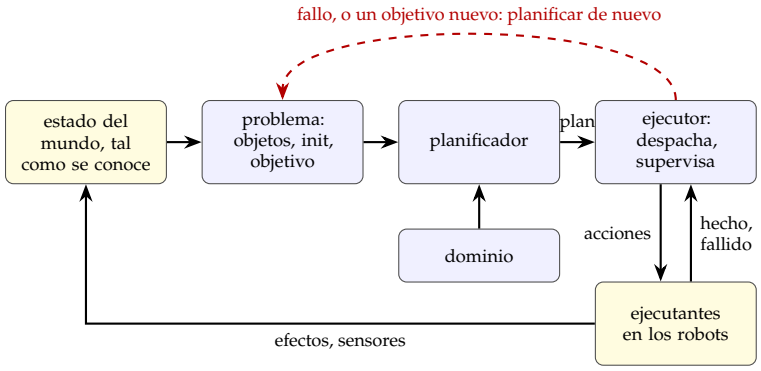


Figura 20.6.: Planificar, ejecutar, supervisar, planificar de nuevo. El dominio es fijo. El problema se escribe de nuevo cada vez a partir del estado del mundo tal como lo conoce el sistema.

Hay tres cosas que hacen la replanificación más difícil que la figura, y las dos primeras tienen que ver con el modelo.

El estado intermedio tiene que ser un estado del modelo. Supongamos que a 70 se descubre que la puerta de la sala A está bloqueada, con `r1` en el pasillo llevando la caja rutinaria. El problema nuevo empieza con `(carrying r1 meds2)`, y en nuestro dominio hay exactamente una manera de dejar de llevar algo: `deliver`, en el destino, al que no se puede llegar. `r1` no sirve para nada el resto de la ronda. «Nada de cajas en los pasillos» eliminó `drop`, y con ella cualquier manera de renunciar a una entrega. Un modelo que se va a ejecutar necesita las acciones de recuperación (devolver la caja a su sitio), aunque ningún buen plan las use nunca.

Ha pasado el tiempo. Un problema escrito a 30 tiene un calendario que empieza a 30: la ronda acaba dentro de 70 segundos, no de 100. Todos los literales temporizados se desplazan, y los que ya han pasado se convierten en hechos normales del estado inicial. Y *mientras* el planificador piensa, el reloj corre: un plan que costó tres segundos encontrar tiene que ser válido para un mundo tres segundos más viejo.

Siempre hay algo en marcha. A 30, `r2` está en mitad de una recogida. Las opciones honestas son esperar a un instante tranquilo, cancelar lo

que está en marcha, o dejar que termine y planificar desde el estado que deje. El Ejercicio 20.7.1 toma la tercera.

La ejecución necesita ejecutantes, dependencias en lugar de horas de reloj y un camino de vuelta al planificador. Un modelo pensado para ejecutarse conserva acciones de recuperación y holgura para los retrasos.

20.8. PlanSys2

PlanSys2, el ROS 2 Planning System (Martín et al., 2021), es un entorno de código abierto que implementa el bucle de la Figura 20.6 para robots que funcionan con ROS 2. Esta sección lo describe tal como está en sus fuentes en la versión 4.0.0; solo afirmamos lo que leímos allí, y no hemos ejecutado la flota sobre él.

Cuatro nodos

Un sistema PlanSys2 son cuatro nodos de ROS 2 (Figura 20.7), y el comentario de su programa de arranque lo dice: «los cuatro nodos de PlanSys2, y ningún otro». Los cuatro son *nodos con ciclo de vida* (*lifecycle nodes*), que se configuran y después se activan en orden. Pueden ejecutarse en un solo proceso o por separado, y en ese caso los arranca un quinto programa, un gestor del ciclo de vida.

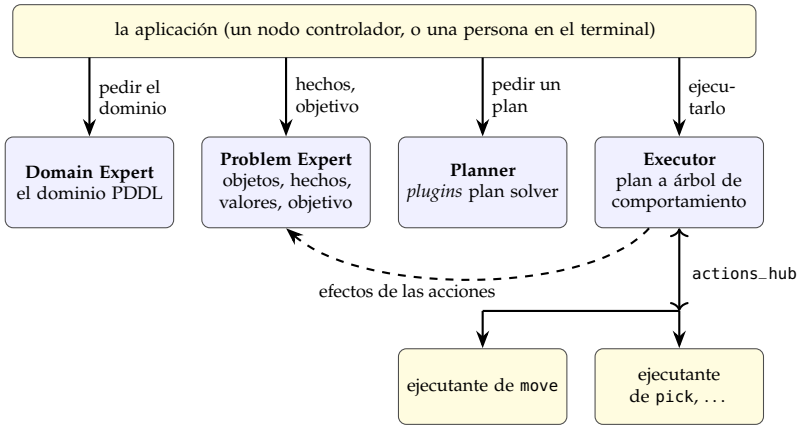


Figura 20.7.: PlanSys2: cuatro nodos, la aplicación que los dirige y un nodo por cada acción del dominio. El Executor aplica los efectos de las acciones al Problem Expert a medida que ocurren.

El Domain Expert guarda el dominio PDDL. Lo lee al configurarse, del fichero que indica su parámetro `model_file`. El parámetro puede enumerar varios ficheros separados por dos puntos; se fusionan en un solo dominio, de modo que cada componente de un robot puede aportar sus propias acciones. El dominio tiene que superar tres comprobaciones: el analizador sintáctico propio de PlanSys2 tiene que poder leerlo; no puede contener nada que PlanSys2 no sepa ejecutar; y después se comprueba como PDDL, por defecto con el comprobador de Kairos usado como biblioteca, el que hay detrás de `kairos check`. A partir de ahí, el nodo responde a preguntas sobre el dominio: sus tipos, predicados, funciones, acciones y acciones durativas, y los detalles de cada uno.

El Problem Expert guarda la otra mitad: las instancias (objetos), los predicados y funciones que son ciertos ahora, y el objetivo. Es el conocimiento que el sistema tiene del mundo. Cada cambio que se pide se verifica contra el dominio, y en cualquier momento el nodo puede escribir lo que guarda como el texto de un problema PDDL. También puede partir de un fichero de problema.

El Planner convierte un dominio y un problema, ambos como texto, en un plan. Los planificadores en sí son *plugins*, los *plan solvers*, enumerados

en el parámetro `plan_solver_plugins`. En las fuentes que leímos, el predeterminado es Kairos, ejecutado como biblioteca dentro del nodo; con el sistema viene un *plugin* para POPF (Coles et al., 2010), que funciona como lo haría uno a mano: escribe `domain.pddl` y `problem.pddl` en una carpeta temporal, ejecuta el planificador y analiza lo que imprime. Cada resolutor declara los requisitos que admite, y una tarea solo se da a los resolutores que cubren lo que usa. El límite de tiempo es el parámetro `plan_solver_timeout`, 15 segundos por defecto. Un plan es una lista de elementos con tres campos, `time`, `action` y `duration`: las líneas que este libro viene imprimiendo desde el Capítulo 10.

El Executor recibe un plan como objetivo de una acción de ROS 2 llamada `execute_plan` y lo lleva a cabo.

Un plan se convierte en un árbol de comportamiento

El Executor no recorre la lista del plan. Construye a partir de ella un *árbol de comportamiento*, una estructura habitual en robótica para componer tareas que tienen éxito, fallan o siguen en marcha, y ejecuta el árbol (con la biblioteca BehaviorTree.CPP). El constructor también es un *plugin*; el predeterminado, `STNBTBuilder`, construye el árbol a partir de una red temporal simple del plan. Cada acción del plan se convierte en un pequeño subárbol. La plantilla para una acción completa es uno de los ficheros más instructivos del sistema para un modelador PDDL (el constructor predeterminado usa los mismos nodos, divididos en una parte para el inicio de la acción y otra para su final):

```
<Sequence name="ACTION_ID">
WAIT_PREV_ACTIONS
  <WaitAtStartReq action="ACTION_ID"/>
  <ApplyAtStartEffect action="ACTION_ID"/>
  <ReactiveSequence name="ACTION_ID">
    <CheckOverAllReq action="ACTION_ID"/>
    <ExecuteAction action="ACTION_ID"/>
  </ReactiveSequence>
  <CheckAtEndReq action="ACTION_ID"/>
  <ApplyAtEndEffect action="ACTION_ID"/>
</Sequence>
```

Esta es la acción durativa del Capítulo 10, ejecutada. El subárbol espera a las acciones de las que depende esta. Espera hasta que las condiciones `at start` se cumplan en el Problem Expert (durante un tiempo limitado, 30 segundos por defecto) y aplica allí los efectos `at start`. Después ejecuta la acción, y mientras tanto comprueba una y otra vez las condiciones `over all`: eso es lo que hace una secuencia *reactiva*. Cuando la acción informa de que ha terminado, se comprueban las condiciones `at end` y se aplican los efectos `at end`. Los tres lugares de una condición y los dos lugares de un efecto, que en el Capítulo 10 eran una cuestión de semántica, son aquí cinco nodos de un árbol. Una condición en el lugar equivocado se comprueba en el momento equivocado en un robot real.

Si un subárbol falla (la acción informó de un fallo, o una condición no se cumplía), el plan falla: el Executor cancela las acciones en marcha y termina la acción `execute_plan` con el resultado `FAILURE`. No vuelve a planificar por su cuenta. Esa decisión corresponde a la aplicación, y el Executor la ayuda de dos maneras que encontramos en las fuentes: puede dar el problema «de ahora», desde el que planificar de nuevo mientras se ejecuta un plan, y acepta un plan nuevo mientras el antiguo se está ejecutando, en cuyo caso las acciones que aparecen en los dos siguen adelante.

Ejecutantes de acciones

¿Qué ejecuta `(move r1 lobby pharmacy)`? Un nodo escrito por el usuario de PlanSys2, uno (al menos) por cada acción del dominio: un **ejecutante de acción**. La clase base es `plansys2::ActionExecutorClient`; el nodo dice en su parámetro `action_name` qué acción del dominio realiza, e implementa un método, `do_work`, al que se llama periódicamente mientras la acción se ejecuta. Este es el `move` del ejemplo más sencillo que viene con PlanSys2, que solo finge moverse:

```
class MoveAction : public plansys2::ActionExecutorClient
{
public:
  MoveAction()
  : plansys2::ActionExecutorClient("move") { progress_ = 0.0; }

private:
  void do_work()
  {
    // As long as the plan says it lasts
    progress_ += 1.0 / std::max(1.0,
      get_planned_duration() * get_parameter("rate").as_double());
    if (progress_ < 1.0 - 1e-3) {
      send_feedback(progress_, "Move running");
    } else {
      finish(true, 1.0, "Move completed");
      progress_ = 0.0;
    }
  }
  float progress_;
};
```

Un ejecutante real enviaría un objetivo al sistema de navegación y llamaría a `finish(true, ...)` cuando el robot llegara, o a `finish(false, ...)` cuando no pudiera. Tiene a su disposición los argumentos de la acción (qué robot, desde dónde, hacia dónde), y un parámetro llamado `specialized_arguments` permite que un ejecutante acepte solo algunos, por ejemplo solo los movimientos de `r1`: en una flota, cada robot ejecuta los suyos.

El Executor y los ejecutantes no se llaman directamente. Hablan por un *topic*, `actions_hub`, con un pequeño protocolo. El Executor publica una *petición* de una acción con sus argumentos; los ejecutantes que pueden hacerla y están libres responden; el Executor *confirma* al primero y *rechaza* a los demás; el elegido envía *realimentación* mientras trabaja y un mensaje de *fin* al terminar, y se le puede *cancelar*. El Executor no necesita ninguna lista de quién hace qué, y un segundo robot es un conjunto más de nodos que responden. Un ejecutante también puede ser a su vez un árbol de comportamiento: el paquete `plansys2_bt_actions` proporciona uno que ejecuta un árbol dado en un fichero.

La aplicación

La última pieza es el programa que sabe para qué sirve el robot. Dice al Problem Expert qué existe y qué es cierto, fija un objetivo, pide un plan y lo envía a ejecutar. En el ejemplo de patrulla de PlanSys2 las llamadas son estas (abreviadas):

```
problem_expert_->addInstance(plansys2::Instance{"r2d2", "robot"});
problem_expert_->addPredicate(
  plansys2::Predicate("(robot_at r2d2 wp_control)"));
problem_expert_->setGoal(plansys2::Goal("(and(patrolled wp1))"));
```

```
auto domain = domain_expert_->getDomain();
auto problem = problem_expert_->getProblem();
auto plan = planner_client_->getPlan(domain, problem);
if (plan.has_value()) {
  executor_client_->start_plan_execution(plan.value());
}
```

El dominio y el problema viajan como cadenas de texto, el mismo texto que este libro ha guardado en ficheros. Ese ejemplo es una pequeña máquina de estados alrededor de estas llamadas: cuando un plan termina, fija el siguiente objetivo y planifica de nuevo; cuando uno falla, decide qué hacer. Para el uso interactivo hay un terminal con órdenes como `set instance`, `set predicate`, `set goal`, `get plan` y `run`.

Lo que esto pide al modelo

¿Qué PDDL se puede usar? El analizador de PlanSys2 lee dominios con tipos, constantes, predicados, funciones, acciones y acciones durativas, y también predicados derivados, procesos, eventos y restricciones, y acepta como nombres los requisitos de todos los capítulos de este libro. Hay otros dos filtros que importan más. El *plan solver* tiene que admitir lo que usa la tarea: el *plugin* de POPF, por ejemplo, declara `:strips`, `:typing`, `:equality`, `:universal-preconditions`, fluentes numéricos, `:durative-actions` y `:timed-initial-literals`. Y el Executor tiene que poder comprobar y aplicar las condiciones y los efectos de cada acción contra el Problem Expert, porque así es como se ejecuta el plan; un dominio con construcciones que no tiene forma de ejecutar se rechaza al cargarlo, con el mensaje `PlanSys2 cannot execute this domain`. No hemos probado nuestras cuatro versiones contra ese filtro, y un modelador que apunte a PlanSys2 debería hacerlo lo primero, con el terminal, antes de escribir un solo ejecutante.

Sea cual sea el resultado, el entorno explica a posteriori varias decisiones de este capítulo.

- **Las acciones son lo que saben hacer los ejecutantes.** `move`, `pick`, `seal`, `deliver` y `charge` son cinco piezas de software que un especialista en robótica puede escribir y probar por separado. La granularidad del paso D es también la arquitectura de software del robot.
- **Las condiciones se comprueban en un robot en marcha.** `(over all (is-open ?l))` en `pick` se evalúa mientras se mueve el brazo. Es cierta o falsa en el Problem Expert, así que alguien tiene que decirle al Problem Expert que la farmacia ha abierto. Un literal inicial temporizado es una promesa sobre el mundo; en la ejecución, algo tiene que cumplirla o avisar de que se ha roto.
- **La carga completa es una acción con un final natural.** Un ejecutante de `charge` atraca en el cargador, espera y avisa. Las cuatro cargas de un segundo de la Sección 20.4 serían cuatro peticiones, cuatro atraques.
- **El estado son hechos.** Todo lo que la aplicación pueda necesitar decir sobre el mundo en mitad de una ronda tiene que ser un predicado o una función del dominio. Un robot que sostiene una caja que no puede entregar era uno de esos estados, y nuestro dominio no tenía ninguna acción para él.

20.9. Lo que PDDL deja fuera

El Capítulo 1 enumeró los supuestos clásicos y prometió levantarlos uno a uno. El libro ha levantado tres: los números hicieron infinito el espacio de estados, el tiempo hizo que las acciones duraran y se solaparan, PDDL+ hizo que el mundo cambiara por sí solo. Dos siguen en pie después de veinte capítulos, y es en la ejecución donde se notan.

Determinismo. En todos los modelos de este libro una acción tiene un solo resultado. En el pasillo, una recogida a veces falla, una puerta a veces está bloqueada, un movimiento tarda entre diez y veinte segundos. PDDL, tal como lo hemos estudiado, no puede decir «a veces», ni puede decir «normalmente». Un plan es una secuencia, no una estrategia con una rama para cada resultado. La respuesta práctica habitual es la de este capítulo: planificar como si el mundo fuera determinista, con el resultado más probable y algo de margen, vigilar y planificar de nuevo. Funciona bien cuando los fallos son raros y no son peligrosos, y cuando ninguna acción es irreversible de una manera que importe. Funciona mal cuando lo correcto es *prepararse* para un fallo: ningún plan determinista lleva una segunda bolsa por si la primera se rompe.

Conocimiento completo. El estado inicial de un fichero de problema es completo: la hipótesis del mundo cerrado del Capítulo 2 dice que lo que no aparece es falso. Un robot no sabe tanto. Puede no saber si la muestra está lista, en qué sala se dejó el carro, o si la puerta está cerrada con llave hasta que llega allí. Para averiguarlo tiene que mirar, y mirar es una acción cuyo efecto recae sobre lo que el robot *sabe*, no sobre el mundo: una acción de **percepción**. Un modelo con ellas tiene planes que se ramifican según lo que se ve, y un estado que es un conjunto de posibilidades en lugar de un conjunto de hechos; esto es planificar con **observabilidad parcial**. Nuestros modelos lo esconden en el ejecutor: el ejecutante de `pick` busca la caja con una cámara, y el planificador solo oye «hecho» o «fallido».

Otras cosas quedan fuera por motivos más modestos. La *geometría*: si la caja cabe, dónde colocarse exactamente. *Otros agentes* con sus propios objetivos: las personas del pasillo no son ni acciones ni procesos. *Magnitudes que se aprenden*, no que se conocen: cada duración de nuestros problemas es la estimación de alguien. *Objetos nuevos*: un problema enumera sus objetos, y una petición que llega durante la ronda es un problema nuevo.

La investigación en planificación tiene lenguajes y algoritmos para casi todo esto: modelos no deterministas y probabilísticos, planificación contingente y conforme, modelos que mezclan tarea y movimiento. Los libros de texto citados en las lecturas complementarias los tratan. Para el modelador, la cuestión práctica es dónde poner la frontera. Nada de lo que se deja fuera del modelo PDDL desaparece; acaba en un ejecutante, en el supervisor o en el margen de un número. La ficha del modelo tiene una línea exactamente para esto, el Alcance, y al final de un proyecto es la línea que merece la pena volver a leer: para cada cosa que hay en «Fuera», ¿quién se ocupa de ella ahora?

Lo que el modelo deja fuera no desaparece: va a un ejecutante, al supervisor o a un margen. La línea de Alcance de la ficha del modelo debería decir adónde.

20.10. El método MODEL: una historia, cuatro modelos

El método quedó completo en el Capítulo 19; un caso de estudio no le añade ningún paso. Añade experiencia sobre cómo se comportan los

pasos a este tamaño, y la recogemos en forma de preguntas, como hizo cada capítulo.

Lo que el caso añade al método

M	¿Qué tuve que preguntar a la autora de la historia? (Cada respuesta es una línea de la ficha.) ¿En qué orden vendrán las versiones, y qué no podrá decir cada una?
O, D	Para cada versión: ¿qué se añade a las tablas de la anterior? Si la respuesta es «casi todo», el salto es demasiado grande.
E	¿Conserva cada versión sus cuatro pruebas? ¿Se dio cada plan de la versión anterior al validador de esta, y se rechaza por el motivo que esperaba? Para cada prueba sin plan: ¿el planificador lo demuestra, se rinde o agota el tiempo, y puedo calcular la respuesta a mano?
L	¿Incluye la comparativa el planificador que voy a usar y el tiempo que puedo darle? ¿Qué frase de la historia, si fuera distinta, cambiaría el veredicto?
Después	¿Quién ejecuta cada acción? ¿Cuál es la holgura más pequeña del plan? ¿Desde qué estado intermedio de un plan no podría partir el modelo?

Dos de estas merecen un comentario. La primera es el papel de la historia en el paso L. Tres de nuestros cuatro veredictos los decidió una frase de la responsable: «no puede pasar» eligió la precondición frente al efecto condicional; «se queda hasta llenarse» eligió la carga fija; «un poco» relegó la batería PDDL+ al papel de auditora. Los cinco criterios no eligen en abstracto. La Fidelidad es fidelidad *a una ficha*, y otra ficha da otro ganador con la misma tabla.

La segunda es el papel del planificador. Dos veces un modelo correcto obtuvo una respuesta pobre dentro de nuestro límite de tiempo, y las dos veces la reacción tuvo los mismos tres movimientos: *relajar* el problema para encontrar qué característica cuesta; conservar el modelo completo como *especificación*; y obtener planes de una pregunta más fácil, con el validador del modelo completo como juez. El validador es el punto fijo de todo el capítulo. Leyó planes contra una restricción en la versión 1, el plan de una versión contra la siguiente, planes conjuntos contra un problema conjunto, y un plan escrito a mano en tiempo continuo en la versión 4.

Alternativas comparadas

Las alternativas de cada versión se compararon en sus secciones (Tablas 20.3, 20.6, 20.7 y 20.9). Queda comparar las propias versiones, como respuestas a la página de la Sección 20.1. La Tabla 20.11 toma las tres que tienen tiempo; la versión 1 es su base común.

	Versión 2: tiempo, batería	Versión 3: calendario, deseos	Versión 4: batería PDDL+
Fidelidad	Planes imposibles: recogidas con la farmacia cerrada, entregas después de la ronda. Sin deseos	La ficha, salvo que esperar es gratis	La batería tal como es; sin deseos, sin laboratorio
Tamaño	82 acciones durativas	76, y 5 preferencias	17 para un robot
Guía	La mejor: los dos robots, 113,017 en menos de un segundo	Un robot cada vez: coste 1 con cuatro llamadas de 3 s; una llamada: coste 12 en 3 s	Un robot, un objetivo: menos de un segundo
Claridad	Los planes se leen como horarios	El fichero de problema se lee como la ficha	Los planes necesitan una gráfica
Portabilidad	La mayoría de los planificadores temporales	Planificadores con literales temporizados y preferencias	Kairos

Tabla 20.11.: Las versiones como modelos de la misma página, con sus planes para la ronda o, en la versión 4, para uno de sus robots.

Veredicto. Ninguna versión sustituye a las demás, y ese es el final normal de un proyecto de modelado. La *versión 3* es el modelo de la ronda: úsala para planificar, un robot cada vez mientras el planificador lo necesite. La *versión 2* es a la que recurrir cuando no hay calendario, por ejemplo en un turno de noche con todo abierto; sus planes llegan enseguida y son los mejores de los tres. La *versión 4* es el instrumento para las preguntas sobre energía y espera, un robot cada vez. La *versión 1* sigue ganándose su sitio: es donde las reglas sobre quién puede llevar qué se prueban en milisegundos y se demuestran, antes de que el tiempo haga raras las demostraciones. Un equipo que conservara solo la última versión habría perdido sus pruebas más rápidas.

20.11. Errores comunes

Escribir primero la última versión

La tentación ante una página como la nuestra es abrir un fichero y escribir acciones durativas, una batería, literales temporizados y preferencias de una sentada. Cuando el planificador responde entonces

```
kairos: timeout (timeout)
```

no hay manera de saber si hay una regla mal, un número mal, o si el problema solo es difícil. **Solución:** haz crecer el modelo por versiones. Cuando la versión 3 nos decepcionó pudimos decir enseguida que las reglas estaban bien (la versión 1 las demuestra), que las duraciones y la batería estaban bien (la versión 2 las planifica), y mirar solo lo que era nuevo.

Quedarse con la primera respuesta de un problema con deseos

```
; better plan 2
; cost = 12
; makespan = 67.005
```

El plan es válido y el código de salida es 0. Concede un deseo de cinco. Con objetivos blandos, «resuelto» significa que se cumplen los objetivos estrictos y nada más. **Solución:** antes de ejecutar, ten claro cuál sería un buen coste (aquí: como mucho el peso de la entrega más barata, 2), lee el coste y no solo el estado, y si la distancia es grande, relaja el problema para averiguar por qué.

Leer un límite de tiempo como «no hay plan»

```
note: no plan within the time limit of 2 s (114881 states
     searched). More time may find one: --timeout (SolveOptions
     ::timeout). More threads run the phases of the search at
     once: --threads (SolveOptions::threads). [orchestrator.why]
kairos: timeout (timeout)
```

Esta fue la respuesta para `courier-fleet-day-p08`, que tiene plan: lo teníamos en un fichero. El libro ya ha visto tres respuestas distintas para «no devuelvo ningún plan»: `unsolvable` (código de salida 10, una demostración), `no plan found` (14, la búsqueda se agotó sin demostración) y `timeout` (11, la búsqueda no terminó). Una escalera de problemas que trata la tercera como la primera descarta entregas que eran posibles. **Solución:** para los problemas que deciden algo, ten al otro lado un plan escrito a mano o un cálculo.

Un plan válido sin holgura

```
97.011: (deliver r2 sample1 lab) [2.000]
```

La entrega termina a 99,011 y su plazo es 100. `plan valid`, dice el validador, y la primera puerta lenta lo vuelve falso. Los planificadores que minimizan un makespan o un coste empujan las acciones contra los plazos, porque nada en el modelo dice que llegar pronto tenga valor. **Solución:** pon el margen en el modelo, donde el planificador tiene que respetarlo: un plazo de 95 en el problema para uno real de 100, una reserva de batería. Y después de planificar, calcula la holgura más pequeña del plan como una prueba más.

Un modelo que no puede empezar desde la mitad

Nuestro dominio no tiene `drop`, por decisión propia. Da al planificador el estado que deja una ronda interrumpida, un robot que lleva una caja cuyo destino está cerrado, y cualquier objetivo para ese robot es inalcanzable: la nota dice que `deliver` «is never applicable», como en el `p03` de la versión 1. En planificación, este era un modelo fiel. En ejecución, es un robot perdido para todo el día. **Solución:** antes de que un modelo vaya a un ejecutor, enumera los estados en los que se puede interrumpir un plan y comprueba que cada uno tiene una salida; añade las acciones de recuperación, con un coste o una preferencia en contra si los buenos planes deben evitarlas.

20.12. Resumen

Una página de una responsable de logística se ha convertido en cuatro modelos probados. La Tabla 20.12 reúne para qué sirvió cada uno.

Versión	Qué dice	Su plan para la ronda
1, clásica	quién lleva qué; la regla del laboratorio; entregar solo en el destino	31 acciones, óptimo demostrado, todas de un robot
2, temporal y numérica	duraciones, trabajo en paralelo, batería, un cargador para uno	113,017 s, una carga; ignora el calendario
3, calendario y deseos	horarios de apertura, plazos, obligatorio y deseado	métrica 1: todas las entregas a tiempo, un robot no llega a casa
4, batería PDDL+	esperar cuesta; el cargador termina la carga	un robot: válido donde la versión 3 se apagaría

Tabla 20.12.: Las cuatro versiones de la ronda de la mañana.

- **El paso M es una conversación.** Las frases que decidieron nuestras codificaciones fueron respuestas a preguntas que la página no había previsto.
- **Versiones, cada una con sus pruebas.** Cada salto se dio por algo que la versión anterior no podía decir, y los planes de una versión pusieron a prueba la siguiente.
- **Las alternativas se ejecutaron.** Tres codificaciones de una regla, tres de una carga, tres maneras de planificar deseos, dos niveles para una batería; en cada caso, las descartadas son ficheros que siguen pasando sus pruebas.
- **El planificador forma parte de la decisión.** Duraciones en sus cotas, un motor lento con calendario, una fase de búsqueda costosa para una disyunción en PDDL+: cada uno moldeó una codificación, y cada uno queda anotado junto a la decisión.
- **El validador es el juez.** De planes escritos a mano, de planes de otra versión, de planes conjuntos, de planes en tiempo continuo.
- **Un plan aún no es comportamiento.** La ejecución necesita ejecutantes, dependencias, un supervisor y el camino de vuelta al planificador; PlanSys2 los proporciona para ROS 2, con el dominio como contrato entre todas las partes.
- **Lo que queda fuera del modelo es trabajo de alguien.** La incertidumbre, la percepción y la geometría fueron a los ejecutantes, al supervisor y a los márgenes.

El libro termina donde empezó el Capítulo 1: un resolutor general y un modelo. El resolutor es trabajo de otros y mejora cada año. El modelo es tuyo.

20.13. Lecturas complementarias

Para actuar con planes, la referencia es el segundo libro de Ghallab, Nau y Traverso (Ghallab et al., 2016), organizado en torno a la idea de que la deliberación continúa durante la ejecución; su primer libro (Ghallab et al., 2004) tiene capítulos sobre planificación con incertidumbre y sobre planificación con tiempo. Las redes temporales simples que se usan para despachar planes temporales las introdujeron Dechter et al. (1991). Geffner y Bonet (Geffner & Bonet, 2013) dan una presentación compacta de los modelos más allá del clásico, con acciones no deterministas y con observabilidad parcial. PlanSys2 lo describen sus autores en (Martín et al., 2021); sus fuentes y tutoriales son el sitio donde comprobar cualquier cosa dicha aquí, porque el sistema no deja de cambiar. La validación de planes, en la que este capítulo se ha apoyado a cada paso, se remonta a VAL (Howey et al., 2004), y Kairos se describe en (Martín Rico et al., 2026). Para las características combinadas en el caso, los artículos que las definen son los de PDDL 2.1 (Fox & Long, 2003), PDDL 2.2 (Edelkamp &

Hoffmann, 2004), PDDL3 (A. Gerevini & Long, 2005) y PDDL+ (Fox & Long, 2006).

20.14. Ejercicios

Los ejercicios de este capítulo son proyectos. Cada uno pide ficheros que pasen sus pruebas y los motivos de las decisiones; las soluciones de referencia están en `examples/ch20/solutions`. Da a los problemas temporales un límite de tiempo de tres segundos, como hace el capítulo.

Ejercicios de la Sección 20.3 — Versión 1: quién lleva qué

Ejercicio 20.3.1. La UCI pasa a ser una habitación restringida: solo puede entrar un robot con tarjeta de acceso, y solo `r1` tiene una. Amplía la versión 1 con el vocabulario del Capítulo 5 (`restricted`, `has-keycard`), manteniendo la ronda como prueba típica. Escribe las cuatro pruebas. ¿Qué línea de la ficha del modelo cambia? ¿Sigue usando un solo robot el mejor plan?

Ejercicio 20.3.2. La Sección 20.3 afirma que la precondición de `move` vigila la puerta del laboratorio y la restricción vigila la habitación. Escribe un problema en el que el dominio del Listado 20.1 y el de la restricción (`courier-fleet-rule-domain.pddl`) no estén de acuerdo. Después repara el primer dominio para que coincidan, y comprueba que la ronda sigue teniendo su plan de 31 pasos. ¿Qué te dice esto sobre la fila de Fidelidad de la Tabla 20.3?

Ejercicios de la Sección 20.4 — Versión 2: tiempo y batería

Ejercicio 20.4.1. El hospital instala un segundo cargador en la sala B. Escribe el problema (no hace falta cambiar el dominio: ¿por qué?) y ejecuta la ronda. Compara el plan con el de la Figura 20.3 y explica lo que ves. ¿Se planifica siempre mejor un modelo con más opciones?

Ejercicio 20.4.2. Supón que la responsable hubiera contestado «un robot puede dejar el cargador cuando quiera». Escribe la tercera codificación del Capítulo 11, una carga en bloques de 5 segundos, como variante de la versión 2, ejecuta la ronda y añade una columna a la Tabla 20.6. ¿Qué codificación elegirías ahora, y qué decidió el tamaño del bloque?

Ejercicios de la Sección 20.5 — Versión 3: un calendario y deseos

Ejercicio 20.5.1. La recepción del laboratorio cierra a 90, y un artículo solo se puede entregar a alguien: el destino tiene que estar abierto durante toda la entrega. Cambia la versión 3 (¿qué acción, qué clase de condición?) y planifica el día de `r1` cuando elige primero (`p05`). ¿Qué pierde `r1`, y podría recuperarlo el otro robot? ¿Qué le pasa al plan de la Figura 20.4?

Ejercicio 20.5.2. Escribe el plazo de la caja urgente como una restricción del Capítulo 15, `(within 80 (delivered meds1))`, en la prueba trivial de la versión 3, sin su literal temporizado. Ejecútala. Después di cuáles de los literales temporizados de la ronda se podrían escribir así y cuáles no.

Ejercicio 20.5.3. La cocina se queja, y la comida pasa a valer 6. Planifica el día de `r2` cuando elige en segundo lugar (`p06`) con el peso nuevo. ¿Qué entrega hace? Comprueba lo que se dijo sobre los pesos después de la ficha del modelo: ¿sigue siendo cierto? Explica a la responsable, en dos frases, qué significa su nuevo número para los medicamentos.

Ejercicios de la Sección 20.6 — Versión 4: la batería es cosa del mundo

Ejercicio 20.6.1. Da a los robots de la versión 4 el interruptor del Capítulo 17: un robot ocioso se puede apagar y encender, y un robot apagado no se descarga. Escribe a mano el plan para la prueba de la trampa `courier-fleet-plus-p04` que espera apagado, y valídalo. Después ejecuta el planificador sobre el dominio nuevo. ¿Encuentra tu plan? ¿Qué le dirías a alguien que concluya, a partir de la respuesta del planificador, que el interruptor no sirve para nada?

Ejercicios de la Sección 20.7 — Del plan a la ejecución

Ejercicio 20.7.1. Se está ejecutando la ronda de la Figura 20.4. A 30, cuando `r1` acaba de entregar la comida en la sala B, llega una petición urgente nueva: una muestra, `sample2`, espera en la sala B y tiene que estar en el laboratorio antes de que acabe la ronda. Escribe el problema de replanificación para `r1`: decide qué hacer con `r2`, que está en mitad de una recogida; desplaza el calendario; calcula la batería de `r1`; decide qué pasa con la caja rutinaria que `r1` iba a buscar. Planifícalo. Después comprueba, sobre la línea de tiempo, que el plan nuevo y el resto del plan de `r2` no entran en conflicto.

Ejercicios de la Sección 20.10 — El método MODEL: una historia, cuatro modelos

Ejercicio 20.10.1. Una historia nueva. *«Nuestro almacén trabaja de noche con dos carretillas elevadoras. Los palés llegan al muelle y cada uno tiene su sitio en la estantería A o en la estantería B. Entre el muelle y las estanterías hay un pasillo, y es estrecho: cabe una carretilla, dos no, y lo mismo pasa delante de cada estantería. La zona del muelle es amplia. Ir de un sitio al siguiente lleva unos 8 segundos; cargar o colocar un palé, unos 5.»* Llévala por el método: la ficha del modelo (con las preguntas que harías), una versión clásica con sus cuatro pruebas y una versión temporal. Presta atención a lo que significa «cabe una carretilla, dos no» mientras una carretilla va de camino de un sitio al siguiente.

Apéndices

Kairos A.

A.1 Instalación 510
Con pip 510
Desde el código fuente . . 510
En un espacio de trabajo de colcon 510
A.2 Las órdenes 511
A.3 Planificar 511
Opciones 512
Motores y búsquedas . . 512
A.4 Formatos de plan 513
A.5 Estado y códigos de salida 514
A.6 Comprobar un modelo . 514
A.7 Validar un plan 515
A.8 Mirar por dentro 516
A.9 Desde Python 517

Kairos es el planificador que usan todos los ejemplos de este libro: un planificador PDDL temporal y numérico de código abierto, escrito en C++20, que se puede usar como biblioteca, desde la línea de órdenes y desde Python (Martín Rico et al., 2026). Se desarrolla dentro de la organización PlanSys2 (https://github.com/PlanSys2/kairos) y no depende de ROS. Este apéndice reúne lo que necesita un lector del libro: cómo instalarlo, las órdenes y opciones que usan los capítulos, cómo leer lo que imprime y un breve vistazo a su módulo de Python. La referencia completa es la documentación que acompaña al planificador (`README.md` y `docs/cli.md`). Las salidas que se muestran aquí se obtuvieron con la versión 1.0.6.

A.1. Instalación

Con pip

La forma más rápida, en Linux (x86-64 y arm64), es el paquete de Python. Instala la orden `kairos` junto con el módulo de Python de la Sección A.9:

```
$ pip install kairos-planner
$ kairos --version
kairos 1.0.6
```

Desde el código fuente

Kairos se compila con CMake, sin nada más. Necesita CMake 3.22 o posterior y GCC 11 o Clang 17, o posteriores. Desde la raíz de su repositorio:

```
$ cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
$ cmake --build build
$ ctest --test-dir build --output-on-failure
```

Las pruebas son opcionales; usan GoogleTest. El módulo de Python también se puede compilar desde el árbol de fuentes, con `pip install .`

En un espacio de trabajo de colcon

Quienes trabajan en robótica suelen tener su software en un espacio de trabajo de colcon. Con el repositorio de Kairos dentro de la carpeta `src` del espacio de trabajo:

```
$ colcon build --packages-select kairos
$ source install/setup.bash
```

Después de la línea `source`, `kairos` está en el `PATH`, con autocompletado de órdenes, opciones y ficheros PDDL en bash y zsh. Fuera de colcon, el autocompletado se activa con una línea en `~/.bashrc`:

```
$ eval "$(kairos --completion bash)"
```

Para zsh el argumento es `zsh`. Para fish, la línea es `kairos --completion fish | source`.

A.2. Las órdenes

Orden	Qué hace
`kairos D P [options]`	planifica; el plan va a la salida estándar
`kairos check D [P]`	comprueba un dominio, o un problema frente a su dominio, sin planificar
`kairos check --problem P`	comprueba un problema por sí solo
`kairos validate D P PLAN`	comprueba que un plan resuelve un problema
`kairos --features`	enumera las características de PDDL y hasta dónde planifica Kairos con cada una
`kairos --version`, `--help`	la versión; el texto de uso

Tabla A.1.: Las órdenes de Kairos. `D` es un fichero de dominio y `P` uno de problema. La primera orden también se puede escribir `kairos solve D P`.

Dos reglas valen para todas. La salida estándar recibe solo resultados (planes, la versión, el texto de uso); todo lo demás (diagnósticos, progreso, estadísticas, los veredictos de `check` y `validate`) va a la salida de error estándar, de modo que redirigir la salida a un fichero da un plan limpio. Y toda orden termina con un código de salida que un script puede comprobar (Sección A.5).

A.3. Planificar

Desde la carpeta `examples/ch02`, con el problema del mensajero del Capítulo 2:

```
$ kairos courier-domain.pddl courier-p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Kairos lee y comprueba los dos ficheros, planifica y valida cada plan frente al problema antes de imprimirlo. La búsqueda es *anytime* por defecto: después del primer plan sigue buscando otros mejores y los

imprime a medida que los encuentra, separados por una línea `; better plan N`. El último plan impreso es el mejor. La búsqueda termina cuando demuestra que no existe nada mejor, cuando se acaba el tiempo o cuando lo indica alguna de las opciones de parada.

El **coste** de un plan, impreso en su primera línea, es el valor de la `:metric` del problema cuando la tiene; si no, es el número de acciones en un plan secuencial y el *makespan* en uno temporal.

Opciones

Opción	Significado (valor por defecto)
`--timeout S`	tiempo para la búsqueda, en segundos (30)
`--no-anytime`	parar en el primer plan
`--target-cost C`	parar en el primer plan de coste C o menor
`--no-improvement S`	parar cuando pasen S segundos sin un plan mejor
`--output FILE`	escribir los planes en ficheros, no en la salida estándar
`--format F`	formato del plan: `ipc`, `popf` o `json` (`ipc`)
`--threads N`	fases de la búsqueda que se ejecutan a la vez (los núcleos, hasta 4); con 1, las ejecuciones son reproducibles
`--memory MB`	límite de memoria (la mitad de la disponible)
`--engine E`	motor de planificación: `auto`, `m1`, `m2`, `m3` (`auto`)
`--search NAME`	configuración de la búsqueda (`auto`)
`--epsilon E`	separación entre acontecimientos que interfieren en un plan temporal, en segundos (0,001)
`-v`	progreso y estadísticas en la salida de error
`--debug[=CHANNELS]`	mensajes de depuración, todos o de algunas etapas (`--debug=search,ground`)
`--dump[=WHAT], --dump-dir DIR`	escribir en ficheros las estructuras intermedias

Tabla A.2.: Opciones de la orden de planificación. Las opciones y los ficheros se pueden dar en cualquier orden; `--timeout 5` y `--timeout=5` son lo mismo.

Con `--output FILE` y búsqueda anytime, cada plan va a su propio fichero, `FILE.1`, `FILE.2` y así sucesivamente, y el número más alto es el mejor plan. Con `--no-anytime`, el único plan va a `FILE`.

```
$ kairos courier-domain.pddl courier-p01.pddl --output plan
$ ls plan*
plan.1
```

Motores y búsquedas

El valor por defecto, `auto`, elige cómo planificar a partir de la tarea, y casi siempre es la elección correcta. Una tarea sin acciones durativas ni literales iniciales temporizados se busca directamente (motor M1). Una tarea temporal en la que ninguna acción necesita solaparse con otra se planifica primero por compresión (motor M2: cada acción durativa se convierte en una acción instantánea, y el plan se programa en el tiempo después) y luego, con búsqueda anytime, la mejora el motor M3. El motor M3 planifica con el inicio y el final de cada acción durativa como pasos separados sobre una red temporal, como hace el planificador POPF (Coles et al., 2010); es el que sirve para las tareas que necesitan acciones ejecutándose a la vez (Capítulo 11) y para los literales iniciales

temporizados (Capítulo 13). PDDL+ (Capítulo 17) se planifica en pasos de tiempo.

Las búsquedas llevan el nombre de los planificadores en los que se inspiran. La que usa `auto` para una tarea clásica, `lama`, es una búsqueda voraz primero el mejor guiada por dos heurísticas, el plan relajado de FF y una cuenta de hitos, como en LAMA (Richter & Westphal, 2010); el Capítulo 3 explica estos términos. Las demás (`ff`, `eager-lama`, `novelty`, `lama-novelty`, `ehc`, `goals`) se describen en `docs/cli.md`.

A.4. Formatos de plan

El formato por defecto, `ipc`, es el de las competiciones de planificación y el del validador VAL (Howey et al., 2004). Un plan secuencial tiene una acción por línea, como antes. Un plan temporal da para cada acción su instante de inicio y, entre corchetes, su duración; esta es la plantilla temporal del Apéndice B, desde la carpeta `examples/apB`:

```
$ kairos temporal-domain.pddl temporal-p01.pddl --no-anytime
; cost = 67.505
; makespan = 67.505
0.000: (charge r1 dock) [47.500]
47.501: (move r1 dock lobby) [4.000]
51.502: (move r1 lobby pharmacy) [6.000]
57.503: (pick r1 meds1 pharmacy) [2.000]
59.504: (move r1 pharmacy lobby) [6.000]
65.505: (drop r1 meds1 lobby) [2.000]
```

El formato `popf` imprime lo que imprime el planificador POPF, que es lo que lee el *plugin* de POPF de PlanSys2 (Martín et al., 2021): una marca, el coste y cada acción con un instante y una duración, también en los planes secuenciales, donde el instante es la posición.

```
$ kairos courier-domain.pddl courier-p01.pddl --no-anytime --format
    popf
;;;; Solution Found
; Cost: 7.000
0.000: (move r1 dock lobby)  [0.000]
1.000: (move r1 lobby pharmacy)  [0.000]
2.000: (pick r1 meds1 pharmacy)  [0.000]
3.000: (move r1 pharmacy lobby)  [0.000]
4.000: (move r1 lobby corridor)  [0.000]
5.000: (move r1 corridor ward-a)  [0.000]
6.000: (drop r1 meds1 ward-a)  [0.000]
```

Con `popf` en la salida estándar solo se imprime el mejor plan, cuando termina la búsqueda, y el código de salida es siempre 0. El formato `json` está pensado para scripts: una línea por plan, un objeto con los campos `temporal`, `cost`, `makespan` y `steps`, y cada paso con su `time`, su `duration` y su `action`.

```
$ kairos courier-domain.pddl courier-p01.pddl --no-anytime --format
    json
{"temporal":false,"cost":7,"makespan":7,"steps":[{"time":0,"duration
    ":0,"action":"(move r1 dock lobby)"},{"time":1,"duration":0,"
    action":"(move r1 lobby pharmacy)"},
...
```

A.5. Estado y códigos de salida

Cuando una tarea no se resuelve, o siempre con `-v`, una línea de estado en la salida de error dice cómo terminó la ejecución:

```
kairos: STATUS (STOP REASON)[, cost C]
```

Código	Estado	Significado
0	`solved`	al menos un plan válido, también cuando después la búsqueda se detuvo por el tiempo, por la memoria o por una opción
10	`unsolvable`	demostrado: no existe ningún plan
11	`timeout`	sin tiempo antes del primer plan
12	`out of memory`	límite de memoria alcanzado antes del primer plan (también `grounding too large`)
13	`unsupported`	la tarea usa algo con lo que Kairos no sabe planificar
14	`no plan found`	la búsqueda agotó los estados, pero no es completa para esta tarea: probablemente no existe ningún plan, y nada lo demuestra
2	`invalid input`	opciones incorrectas, ficheros ilegibles, errores en el PDDL
1		error interno, o detenido por una señal antes de cualquier plan

Tabla A.3.: Códigos de salida de la orden de planificación. Los códigos 10 y 14 son respuestas distintas: la primera es una demostración; la segunda, no.

El motivo de parada de una tarea resuelta es uno de estos: `proven optimal` (tareas secuenciales: no existe nada más barato), `exhausted` (las búsquedas terminaron; en tareas temporales esto no demuestra que el makespan sea el menor), `timeout`, `no improvement`, `target reached`, `stopped` y `memory limit`. Sin plan, una nota encima de la línea de estado dice por qué y qué opción podría cambiarlo; el Capítulo 3 muestra los tres casos habituales. Un script lee el código de salida:

```
kairos domain.pddl problem.pddl --timeout 10 --output plan
case $? in
  0)  echo "solved" ;;
  10) echo "no plan exists" ;;
  11) echo "no plan in time" ;;
  14) echo "no plan found (and no proof that none exists)" ;;
  *)  echo "error" ;;
esac
```

A.6. Comprobar un modelo

`kairos check` verifica el PDDL sin planificar: sintaxis, declaraciones, tipos, aridades, variables. Imprime un veredicto y las características de PDDL que usan los ficheros. Para el mensajero con tipos del Capítulo 4, en `examples/ch04`:

```
$ kairos check courier-domain.pddl courier-p01.pddl
valid, plannable
  typing: full
```

El veredicto es `valid, plannable`; `valid, not plannable yet`, para un PDDL correcto que usa una característica con la que Kairos no

planifica (ninguna hoy: `kairos --features` da todas las características como `full`); o `invalid`. El código de salida es 0 para ficheros válidos y 2 en otro caso.

Un diagnóstico tiene una forma fija: el fichero, la línea y la columna; la gravedad (`error`, `warning` o `note`); el mensaje; un código entre corchetes; y, debajo, la línea culpable con una marca bajo el lugar exacto. Si intercambiamos los dos argumentos de `(robot-at r1 dock)` en el problema:

```
$ kairos check courier-domain.pddl broken-p01.pddl
broken-p01.pddl:16:15: error: in the initial state: argument 1 of '
     robot-at' must be robot, 'dock' is location [semantic.
     type_mismatch]
 16 |     (robot-at dock r1) (free r1)
    |              ^
broken-p01.pddl:16:20: error: in the initial state: argument 2 of '
     robot-at' must be location, 'r1' is robot [semantic.
     type_mismatch]
 16 |     (robot-at dock r1) (free r1)
    |                    ^
broken-p01.pddl:16:5: note: arguments 1 and 2 of 'robot-at' seem to
     be swapped [semantic.swapped_arguments]
 16 |     (robot-at dock r1) (free r1)
    |     ^
invalid
```

Un diagnóstico se lee desde la izquierda: dónde, qué gravedad, qué. Una nota (`note`) que sigue a un error suele traer la corrección, como aquí; también la trae una pregunta entre paréntesis, como cuando `:effect` está mal escrito en el dominio:

```
$ kairos check broken-domain.pddl
broken-domain.pddl:29:5: error: unknown key :efect in action (did
     you mean ':effect'?) [parse.unknown_key]
 29 |     :efect (and (not (carrying ?r ?i))
    |     ^
invalid
```

Los avisos no hacen que un fichero sea inválido. El más habitual en una asignatura es el de una característica que se usa y no se declara en `:requirements` (`semantic.undeclared_requirement`). La orden de planificación muestra los avisos y las notas del análisis sintáctico solo con `-v`; los errores se muestran siempre.

A.7. Validar un plan

`kairos validate` comprueba que un plan resuelve un problema: que cada paso es aplicable en su instante, que los acontecimientos que interfieren están separados y que el objetivo se cumple al final. El plan puede haberlo escrito Kairos, una persona a mano u otro planificador: acepta líneas con instante (`t: (action) [d]`, con la duración opcional), líneas simples (`(action)`), la salida completa de POPF u OPTIC y comentarios que empiezan por punto y coma.

```
$ kairos validate courier-domain.pddl courier-p01.pddl plan.1
plan valid, makespan 7
```

El veredicto es `plan valid` (código de salida 0), con el makespan y la métrica cuando el problema la tiene, o `plan invalid` (código de salida 1), tras un error que nombra el paso y la condición que falló. El Capítulo 3 muestra las dos clases de plan inválido: un paso que no se puede aplicar y un objetivo que no se alcanza.

Una advertencia: la salida anytime de la orden de planificación contiene varios planes uno detrás de otro, y leídos como un único fichero no forman un plan. Para validar un plan de Kairos, escríbelo con `--no-anytime` o con `--output` (y valida el último fichero).

A.8. Mirar por dentro

Tres opciones muestran lo que hace el planificador con un modelo; el Capítulo 3 las usa todas.

`-v` imprime, en la salida de error, los avisos del análisis sintáctico, una línea por etapa con su tiempo (`[ground]`: los átomos y las acciones instanciadas; `[sas]`: las variables; `[orchestrator]`: el motor y la búsqueda elegidos, y el estado final; `[search]`: cada plan), la línea de estado y dos líneas de estadísticas: en qué se fue el tiempo, y el tamaño de la tarea y de la búsqueda (estados, expandidos, evaluados, planes, pico de memoria).

`--debug` añade mensajes de depuración. Sin valor activa todos los canales; con una lista, solo esos: `--debug=search` imprime cada nuevo mejor valor de la heurística. Los canales son `general`, `parse`, `semantic`, `normalize`, `ground`, `sas`, `search`, `heuristic`, `temporal`, `postprocess`, `validate`, `orchestrator` y `memory`. El signo igual es obligatorio.

`--dump` escribe en ficheros las estructuras intermedias, en la carpeta actual o en la que se indique con `--dump-dir`:

Nombre	Fichero	Contenido
`ast`	`ast-domain.pddl`, `ast-problem.pddl`	los ficheros tal como se analizaron, reescritos como PDDL
`lifted`	`lifted.txt`	la tarea después de normalizarla, antes de instanciarla
`ground`	`ground.txt`	los átomos y las acciones instanciadas
`sas`	`sas.txt`	las variables, los grupos de hechos excluyentes y las acciones sobre ellos
`plan-trace`	`plan-trace.txt`	el mejor plan paso a paso, con los hechos que se cumplen después de cada paso

Tabla A.4.: Lo que escribe `--dump`. `--dump=ground,plan-trace` escribe solo esos dos.

Cuando un modelo se porta mal, conviene conocer dos de ellos. `ground.txt` muestra qué acciones se instanciaron: si falta una, apunta a una precondición que nunca puede cumplirse. `plan-trace.txt` muestra el estado después de cada paso del plan, que es la forma más rápida de ver dónde hace un plan algo que el modelador no pretendía.

A.9. Desde Python

El paquete `kairos-planner` proporciona el módulo `kairos`. Planificar a partir de ficheros lleva tres líneas:

```
import kairos
result = kairos.solve("courier-domain.pddl", "courier-p01.pddl",
     timeout=10)
print(result.status, result.plan)
```

Un ejemplo más largo carga el modelo una vez, planifica y valida el plan:

```
import kairos

domain = kairos.load_domain("courier-domain.pddl")
problem = kairos.load_problem("courier-p01.pddl", domain)

result = kairos.Planner().solve(problem, timeout=10)
if result.status == kairos.Status.SOLVED:
    print(result.plan)       # IPC format; result.plan.steps for the
     steps
    print(kairos.validate(problem, result.plan).valid)
elif result.status == kairos.Status.UNSOLVABLE:
    print("no plan exists (proved)")
```

Ejecutado en `examples/ch04`, imprime el plan de siete acciones y `True`. El resto del módulo sigue a la orden:

- **Las opciones** son argumentos con nombre de `solve`: `timeout`, `memory_limit_mb`, `anytime`, `engine`, `threads`, `epsilon`, `target_cost` y otros.
- **Anytime**: `Planner().plans(problem, timeout=30)` entrega cada plan mejor a medida que llega, y `Planner().start(problem, on_plan=...)` ejecuta la búsqueda en su propio hilo.
- **Los estados** son los de la Tabla A.3: `SOLVED`, `UNSOLVABLE`, `NO_PLAN_FOUND`, `TIMEOUT`, `OUT_OF_MEMORY`, `UNSUPPORTED`; `result.diagnostics` dice por qué no hay plan.
- **Los errores en el PDDL** lanzan `kairos.PddlError`. Cada uno de sus `diagnostics` tiene un `code`, un `message` y una `location`: la misma información que los diagnósticos de la Sección A.6.
- **Comprobación y validación**: `check_domain`, `check_problem`, `validate`, `validate_plan`, `parse_plan` y `pddl_support()`.

Referencia rápida de PDDL | B.

Este apéndice es para consultar. Tiene tres partes: los requisitos de PDDL, con lo que permite cada uno y el capítulo que lo enseña; la sintaxis de un dominio y un problema en cinco plantillas, una por nivel del lenguaje, cada una de ellas una versión pequeña del mundo del mensajero que Kairos comprueba y resuelve; y una tabla de qué planificadores planifican con qué características.

B.1 Requisitos 518
B.2 Plantillas de sintaxis . . . 519
Clásico: PDDL 1.2 519
Numérico: PDDL 2.1, nivel 2 521
Temporal: PDDL 2.1, niveles 3 y 4, y PDDL 2.2 . . . 522
Preferencias, restricciones y fluentes de objeto: PDDL 3 524
Híbrido: PDDL+ 525
B.3 Qué admite cada planificador 527

B.1. Requisitos

Un dominio declara en `:requirements` las características que usa. Las Tablas B.1 y B.2 los enumeran en el orden del libro. La versión es la de PDDL que introdujo la característica.

Tabla B.1.: Los requisitos de PDDL: planificación clásica y numérica.

Requisito	Qué permite	PDDL	Cap.
`:strips`	acciones con una conjunción de hechos como precondición, efectos de adición y de borrado	1.2	2
`:typing`	`:types`, objetos, parámetros y predicados con tipo; `either`	1.2	4
`:equality`	`(= ?a ?b)` entre objetos	1.2	4
`:negative-preconditions`	`not` en precondiciones y objetivos	1.2	5
`:disjunctive-preconditions`	`or` e `imply` en precondiciones y objetivos	1.2	5
`:existential-preconditions`	`exists` en precondiciones y objetivos	1.2	5
`:universal-preconditions`	`forall` en precondiciones y objetivos	1.2	5
`:quantified-preconditions`	los dos anteriores, juntos	1.2	5
`:conditional-effects`	`when` en efectos; con `forall`, efectos sobre muchos objetos	1.2	6
`:adl`	todo lo anterior, en una palabra	1.2	6
`:numeric-fluents`	`:functions` con valores numéricos, comparaciones, `increase`, `decrease`, `assign`, `scale-up`, `scale-down`; se escribe `:fluents` en PDDL 2.1	2.1	8
`:action-costs`	`(total-cost)`, que las acciones incrementan y la métrica minimiza	3.1	9

Requisito	Qué permite	PDDL	Cap.
`:durative-actions`	`:durative-action`, con `at start`, `over all`, `at end`	2.1	10
`:duration-inequalities`	una duración entre cotas, elegida por el planificador	2.1	11
`:continuous-effects`	cambio durante una acción, con `#t`	2.1	12
`:derived-predicates`	`:derived`: hechos definidos por una regla	2.2	13
`:timed-initial-literals`	`(at t (fact))` en `:init`: hechos que cambian en instantes conocidos	2.2	13
`:timed-initial-fluents`	lo mismo para valores numéricos; una extensión que aceptan varios planificadores	—	13
`:preferences`	`preference` en objetivos y precondiciones, `is-violated` en la métrica	3.0	14
`:constraints`	`:constraints` con `always`, `sometime`, `within` y los demás	3.0	15
`:object-fluents`	funciones cuyo valor es un objeto	3.1	16
`:fluents`	desde PDDL 3.1: fluentes numéricos y de objeto juntos	3.1	16
`:time`	`:process` y `:event`	PDDL+	17

Tabla B.2.: Los requisitos de PDDL: tiempo, PDDL 3 y PDDL+.

Una `:metric` no necesita ningún requisito propio. Cuando un fichero usa una característica que no declara, Kairos planifica igualmente y da un aviso (`semantic.undeclared_requirement`); otros planificadores pueden rechazar el fichero, así que declara lo que uses.

B.2. Plantillas de sintaxis

Las cinco plantillas son ficheros de `examples/apB`. Cada dominio usa tantas construcciones de su nivel como caben en una página, y las señala con comentarios; cada problema tiene un plan. No son modelos que imitar, porque un modelo real usa solo las características que necesita, sino sitios de los que copiar la sintaxis.

Clásico: PDDL 1.2

Tipos con una jerarquía, constantes, igualdad, las condiciones lógicas (`not`, `or`, `imply`, `exists`, `forall`) y efectos condicionales (`when`, aquí bajo un `forall`). El plan encontrado tiene 15 acciones: el robot coge la tarjeta de acceso antes que nada, porque la farmacia es de acceso restringido.

```
;; Appendix B, template 1: classical PDDL (PDDL 1.2), the
;; language of chapters 2 to 7.
(define (domain courier-classical)
  (:requirements :strips :typing :equality
                 :negative-preconditions
                 :disjunctive-preconditions
                 :existential-preconditions
                 :universal-preconditions
                 :conditional-effects)    ; or all of them: :adl

  (:types robot location item - object      ; a type hierarchy
          medicine sample - item)
```

```

  (:constants dock - location)              ; in every problem

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (restricted ?l - location) (has-keycard ?r - robot)
    (sterile ?l - location)
    (sealed ?i - item) (contaminated ?i - item))

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and
      (robot-at ?r ?from) (connected ?from ?to)
      (not (= ?from ?to))                          ; equality
      (or (not (restricted ?to)) (has-keycard ?r)))   ; or, not
    :effect (and
      (not (robot-at ?r ?from)) (robot-at ?r ?to)   ; delete, add
      (forall (?i - sample)                         ; for each...
        (when (and (carrying ?r ?i) (sterile ?to)   ; ...if
                   (not (sealed ?i)))
          (contaminated ?i)))))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
    :effect (and (not (item-at ?i ?l)) (not (free ?r))
                 (carrying ?r ?i)))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i))
                 (item-at ?i ?l) (free ?r)))

  (:action take-keycard
    :parameters (?r - robot)
    :precondition (and
      (robot-at ?r dock)                            ; a constant
      (forall (?i - item) (not (carrying ?r ?i)))   ; forall
      (exists (?l - location) (restricted ?l)))     ; exists
    :effect (has-keycard ?r)))
```

Listado B.1: PDDL clásico: el dominio.

```
;; Appendix B, template 1: a problem for the classical domain.
(define (problem courier-classical-p01)
  (:domain courier-classical)
  (:objects
    r1 - robot
    lobby pharmacy lab - location           ; dock is a constant
    meds1 meds2 - medicine
    sample1 - sample)
  (:init
    (connected dock lobby) (connected lobby dock)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (connected lobby lab) (connected lab lobby)
    (restricted pharmacy) (sterile lab)
    (robot-at r1 dock) (free r1)
    (item-at meds1 pharmacy) (item-at meds2 pharmacy)
    (item-at sample1 lobby) (sealed sample1))   ; the rest: false
  (:goal (and
    (forall (?i - medicine) (item-at ?i lobby))
    (item-at sample1 lab)
    (not (contaminated sample1))            ; a negative goal
    (imply (has-keycard r1) (robot-at r1 dock)))))
```

Listado B.2: PDDL clásico: un problema.

Numérico: PDDL 2.1, nivel 2

Funciones, precondiciones numéricas, los cinco efectos numéricos, un objetivo numérico y una métrica. La función `(total-cost)` es la de `:action-costs`; aquí la incrementa la distancia, una función estática. Algunos planificadores, entre ellos POPF y OPTIC, solo aceptan una condición o un efecto numérico dentro de `(and ...)`.

```
;; Appendix B, template 2: numeric PDDL (PDDL 2.1 level 2, and the
;; action costs of PDDL 3.1; chapters 8 and 9).
(define (domain courier-numeric)
  (:requirements :typing :numeric-fluents)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (has-charger ?l - location))

  (:functions                                ; numbers in the state
    (battery ?r - robot) (battery-capacity ?r - robot)
    (consumption ?r - robot)
    (load ?r - robot) (capacity ?r - robot)
    (weight ?i - item)
    (distance ?from ?to - location)
    (total-cost))

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and
      (robot-at ?r ?from) (connected ?from ?to)
      (>= (battery ?r)                       ; < <= = >= >
          (* (distance ?from ?to) (consumption ?r))))   ; + - * /
    :effect (and
      (not (robot-at ?r ?from)) (robot-at ?r ?to)
      (decrease (battery ?r)
                (* (distance ?from ?to) (consumption ?r)))
      (increase (total-cost) (distance ?from ?to))))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and
      (robot-at ?r ?l) (item-at ?i ?l)
      (<= (+ (load ?r) (weight ?i)) (capacity ?r)))
    :effect (and (not (item-at ?i ?l)) (carrying ?r ?i)
                 (increase (load ?r) (weight ?i))
                 (increase (total-cost) 1)))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i)) (item-at ?i ?l)
                 (decrease (load ?r) (weight ?i))
                 (increase (total-cost) 1)))

  (:action charge
    :parameters (?r - robot ?l - location)
    :precondition (and (robot-at ?r ?l) (has-charger ?l)
                       (< (battery ?r) (battery-capacity ?r)))
    :effect (and (assign (battery ?r) (battery-capacity ?r))
                 (increase (total-cost) 20)))

  (:action save-energy                       ; scale-up, scale-down
    :parameters (?r - robot)
    :precondition (> (consumption ?r) 0.5)
    :effect (and (scale-down (consumption ?r) 2)
                 (increase (total-cost) 5))))
```

```
;; Appendix B, template 2: a problem for the numeric domain.
(define (problem courier-numeric-p01)
  (:domain courier-numeric)
  (:objects r1 - robot
            dock lobby pharmacy - location
            meds1 meds2 - item)
  (:init
    (connected dock lobby) (connected lobby dock)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (has-charger dock)
    (robot-at r1 dock)
    (item-at meds1 pharmacy) (item-at meds2 pharmacy)
    (= (battery r1) 15) (= (battery-capacity r1) 100)
    (= (consumption r1) 1)
    (= (load r1) 0) (= (capacity r1) 5)
    (= (weight meds1) 2) (= (weight meds2) 4)
    (= (distance dock lobby) 4) (= (distance lobby dock) 4)
    (= (distance lobby pharmacy) 6)
    (= (distance pharmacy lobby) 6)
    (= (total-cost) 0))
  (:goal (and (item-at meds1 lobby) (item-at meds2 lobby)
              (>= (battery r1) 10)))        ; a numeric goal
  ;; minimize or maximize, any expression of fluents
  (:metric minimize (total-cost)))
```

Listado B.3: PDDL numérico: el dominio.

Listado B.4: PDDL numérico: un problema.

Temporal: PDDL 2.1, niveles 3 y 4, y PDDL 2.2

Acciones durativas con una duración fija (`pick`), una calculada (`move`) y una que elige el planificador (`charge`); condiciones y efectos al inicio, al final y durante toda la acción; un efecto continuo; un predicado derivado, definido por una regla recursiva; y, en el problema, literales iniciales temporizados y la métrica `(total-time)`. Un plan temporal da cada acción con su instante de inicio y su duración; el Apéndice A muestra el de este problema.

```
;; Appendix B, template 3: temporal PDDL (PDDL 2.1 levels 3 and 4,
;; PDDL 2.2; chapters 10-13).
(define (domain courier-temporal)
  (:requirements :typing :numeric-fluents :durative-actions
                 :duration-inequalities :continuous-effects
                 :derived-predicates :timed-initial-literals
                 :disjunctive-preconditions
                 :existential-preconditions)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (reachable ?from ?to - location)        ; derived, see below
    (robot-at ?r - robot ?l - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (idle ?r - robot)
    (has-charger ?l - location)
    (is-open ?l - location))

  (:functions
    (distance ?from ?to - location) (speed ?r - robot)
    (battery ?r - robot) (battery-capacity ?r - robot)
    (drain-rate ?r - robot) (charge-rate ?r - robot))

  ;; A fact defined by a rule, never by an effect.
  (:derived (reachable ?from ?to - location)
    (or (connected ?from ?to)
```

```
      (exists (?mid - location)
        (and (connected ?from ?mid) (reachable ?mid ?to)))))

  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
    :condition (and
      (at start (robot-at ?r ?from))        ; when it starts
      (at start (idle ?r))
      (at start (connected ?from ?to))
      (at start (>= (battery ?r)
                    (* (drain-rate ?r)
                       (/ (distance ?from ?to) (speed ?r)))))
      (over all (is-open ?to)))             ; while it runs
    :effect (and
      (at start (not (robot-at ?r ?from)))
      (at start (not (idle ?r)))
      (at end (robot-at ?r ?to))            ; when it ends
      (at end (idle ?r))
      (at end (decrease (battery ?r)
                        (* (drain-rate ?r) ?duration)))))

  (:durative-action charge                  ; the planner chooses
    :parameters (?r - robot ?l - location)  ; how long it lasts
    :duration (and (>= ?duration 1)
                   (<= ?duration (/ (- (battery-capacity ?r)
                                       (battery ?r))
                                    (charge-rate ?r))))
    :condition (and
      (at start (idle ?r))
      (over all (robot-at ?r ?l))
      (over all (has-charger ?l)))
    :effect (and
      (at start (not (idle ?r)))
      (at end (idle ?r))
      (increase (battery ?r)                ; continuous: #t is
                (* #t (charge-rate ?r)))))  ; the time elapsed

  (:durative-action pick
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration 2)
    :condition (and (at start (idle ?r))
                    (at start (item-at ?i ?l))
                    (over all (robot-at ?r ?l)))
    :effect (and (at start (not (idle ?r)))
                 (at start (not (item-at ?i ?l)))
                 (at end (carrying ?r ?i)) (at end (idle ?r))))

  (:durative-action drop
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration 2)
    :condition (and (at start (idle ?r))
                    (at start (carrying ?r ?i))
                    (over all (robot-at ?r ?l)))
    :effect (and (at start (not (idle ?r)))
                 (at start (not (carrying ?r ?i)))
                 (at end (item-at ?i ?l)) (at end (idle ?r)))))
```

```
;; Appendix B, template 3: a problem for the temporal domain.
(define (problem courier-temporal-p01)
  (:domain courier-temporal)
  (:objects r1 - robot
            dock lobby pharmacy - location
            meds1 - item)
  (:init
    (connected dock lobby) (connected lobby dock)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (has-charger dock)
    (robot-at r1 dock) (idle r1)
    (item-at meds1 pharmacy)
    (is-open dock) (is-open lobby)
```

Listado B.5: PDDL temporal: el dominio.

```
    ; timed initial literals: true at 30, false again at 300
    (at 30 (is-open pharmacy))
    (at 300 (not (is-open pharmacy)))
    (= (distance dock lobby) 4) (= (distance lobby dock) 4)
    (= (distance lobby pharmacy) 6)
    (= (distance pharmacy lobby) 6)
    (= (speed r1) 1)
    (= (battery r1) 5) (= (battery-capacity r1) 100)
    (= (drain-rate r1) 1) (= (charge-rate r1) 2))
  (:goal (and (item-at meds1 lobby)
              (reachable lobby dock)))
  (:metric minimize (total-time)))          ; the makespan
```

Listado B.6: PDDL temporal: un problema.

Preferencias, restricciones y fluentes de objeto: PDDL 3

Una preferencia en una precondición y tres en el objetivo, una de ellas bajo un `forall`; restricciones en el dominio y en el problema, una de ellas como preferencia; una métrica que pondera las preferencias violadas; y un fluente de objeto, `(location-of ?r)`, en lugar del predicado `robot-at`. El mejor plan tiene coste 6: sirve la comida, y acepta no volver a la base de carga (2), entrar en la UCI (3) y entrar en una zona de silencio (1).

```
;; Appendix B, template 4: PDDL 3.0 and 3.1 (chapters 14-16):
;; preferences, constraints on the whole plan, object fluents.
(define (domain courier-pddl3)
  (:requirements :typing :equality :negative-preconditions
                 :universal-preconditions
                 :preferences :constraints :object-fluents)

  (:types robot location item)

  (:constants dock - location)

  (:predicates
    (connected ?from ?to - location)
    (item-at ?i - item ?l - location)
    (carrying ?r - robot ?i - item)
    (free ?r - robot)
    (quiet ?l - location))

  (:functions
    (location-of ?r - robot) - location)    ; holds an object

  (:constraints                             ; for every plan
    (forall (?r - robot)
      (at-most-once (= (location-of ?r) dock))))

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and
      (= (location-of ?r) ?from) (connected ?from ?to)
      (preference quiet-moves (not (quiet ?to))))   ; a wish
    :effect (assign (location-of ?r) ?to))

  (:action pick
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (= (location-of ?r) ?l) (item-at ?i ?l)
                       (free ?r))
    :effect (and (not (item-at ?i ?l)) (not (free ?r))
                 (carrying ?r ?i)))

  (:action drop
    :parameters (?r - robot ?i - item ?l - location)
    :precondition (and (= (location-of ?r) ?l) (carrying ?r ?i))
    :effect (and (not (carrying ?r ?i)) (free ?r)
                 (item-at ?i ?l))))
```

```
;; Appendix B, template 4: a problem for the PDDL 3 domain.
(define (problem courier-pddl3-p01)
  (:domain courier-pddl3)
  (:requirements :negative-preconditions :universal-preconditions)
  (:objects r1 - robot
            lobby pharmacy ward-a icu - location
            meds1 meal1 - item)
  (:init
    (connected dock lobby) (connected lobby dock)
    (connected lobby pharmacy) (connected pharmacy lobby)
    (connected lobby ward-a) (connected ward-a lobby)
    (connected ward-a icu) (connected icu ward-a)
    (quiet icu)
    (= (location-of r1) dock) (free r1)
    (item-at meds1 pharmacy) (item-at meal1 lobby))
  (:goal (and
    (item-at meds1 ward-a)                          ; required
    (preference meal-served (item-at meal1 icu))    ; wished
    (preference back-home (= (location-of r1) dock))
    (forall (?i - item)
      (preference hands-off (not (carrying r1 ?i))))))
  (:constraints (and                          ; for this problem
    (always (not (item-at meds1 lobby)))
    (sometime (= (location-of r1) pharmacy))
    (sometime-before (item-at meal1 icu) (item-at meds1 ward-a))
    (preference no-icu
      (always (not (= (location-of r1) icu))))))
  (:metric minimize (+ (* 10 (is-violated meal-served))
                       (* 2 (is-violated back-home))
                       (* 5 (is-violated hands-off))
                       (* 3 (is-violated no-icu))
                       (is-violated quiet-moves))))
```

Listado B.7: PDDL 3: el dominio.

Los operadores de las restricciones (Capítulo 15) son:

Listado B.8: PDDL 3: un problema.

- sin tiempos: `at end`, `always`, `sometime`, `at-most-once`, `sometime-before` y `sometime-after`;
- con tiempos: `within`, `always-within`, `hold-during` y `hold-after`.

Híbrido: PDDL+

Acciones, procesos y eventos. Un proceso actúa mientras se cumple su condición y tiene efectos continuos; un evento se dispara en el instante en que su condición se hace verdadera. En el plan encontrado, el robot saca la muestra, se enchufa durante 2,2 unidades de tiempo, se desenchufa y va al laboratorio.

```
;; Appendix B, template 5: PDDL+ (chapters 17 and 18). Actions are
;; what the agent decides; processes and events are what the world
;; does by itself.
(define (domain courier-plus)
  (:requirements :typing :negative-preconditions
                 :numeric-fluents :time)

  (:types robot location item)

  (:predicates
    (connected ?from ?to - location)
    (robot-at ?r - robot ?l - location)
    (powered ?r - robot)
    (has-charger ?l - location)
    (charging ?r - robot)
    (in-fridge ?i - item) (spoiled ?i - item))
```

```

  (:functions
    (battery ?r - robot) (battery-capacity ?r - robot)
    (drain-rate ?r - robot) (charge-rate ?r - robot)
    (temperature ?i - item))

  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (powered ?r) (robot-at ?r ?from)
                       (connected ?from ?to) (not (charging ?r))
                       (>= (battery ?r) 5))
    :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
                 (decrease (battery ?r) 5)))

  (:action plug
    :parameters (?r - robot ?l - location)
    :precondition (and (robot-at ?r ?l) (has-charger ?l)
                       (not (charging ?r)))
    :effect (charging ?r))

  (:action unplug
    :parameters (?r - robot)
    :precondition (charging ?r)
    :effect (not (charging ?r)))

  (:action take-out
    :parameters (?i - item)
    :precondition (in-fridge ?i)
    :effect (not (in-fridge ?i)))

  ;; Runs for as long as its condition holds.
  (:process drain
    :parameters (?r - robot)
    :precondition (and (powered ?r) (not (charging ?r)))
    :effect (decrease (battery ?r) (* #t (drain-rate ?r))))

  (:process recharge
    :parameters (?r - robot)
    :precondition (and (charging ?r)
                       (< (battery ?r) (battery-capacity ?r)))
    :effect (increase (battery ?r) (* #t (charge-rate ?r))))

  (:process warm-up
    :parameters (?i - item)
    :precondition (and (not (in-fridge ?i))
                       (< (temperature ?i) 20))
    :effect (increase (temperature ?i) (* #t 0.5)))

  ;; Fires at the instant its condition becomes true, and must
  ;; make it false.
  (:event shut-down
    :parameters (?r - robot)
    :precondition (and (powered ?r) (<= (battery ?r) 0))
    :effect (not (powered ?r)))

  (:event spoil
    :parameters (?i - item)
    :precondition (and (not (spoiled ?i)) (>= (temperature ?i) 8))
    :effect (spoiled ?i)))
```

```
;; Appendix B, template 5: a problem for the PDDL+ domain.
(define (problem courier-plus-p01)
  (:domain courier-plus)
  (:requirements :negative-preconditions)
  (:objects r1 - robot
            dock lobby lab - location
            sample1 - item)
  (:init
    (connected dock lobby) (connected lobby dock)
    (connected lobby lab) (connected lab lobby)
    (has-charger dock)
```

Listado B.9: PDDL+: el dominio.

```
    (robot-at r1 dock) (powered r1)
    (in-fridge sample1)
    (= (battery r1) 8) (= (battery-capacity r1) 100)
    (= (drain-rate r1) 1) (= (charge-rate r1) 10)
    (= (temperature sample1) 4))
  (:goal (and (robot-at r1 lab) (powered r1)
              (>= (battery r1) 20)
              (not (in-fridge sample1))
              (not (spoiled sample1)))))
```

Listado B.10: PDDL+: un problema.

B.3. Qué admite cada planificador

La Tabla B.3 procede de la documentación de Kairos, el fichero docs/pddl-support.md. Se obtuvo ejecutando cada planificador, no leyendo su manual: un dominio y un problema pequeños y resolubles por característica, diez segundos por planificador, y cada plan comprobado con `kairos validate`. Una marca de verificación significa que el planificador devolvió un plan válido. Los planificadores son Kairos, OPTIC (Benton et al., 2012), TFD, POPF (Coles et al., 2010), LPG-td, ENHSP (Scala et al., 2016), Metric-FF (Hoffmann, 2003) y LAMA (Richter & Westphal, 2010); los siete últimos en las versiones que distribuye planutils 0.10.25, probadas en octubre de 2026.

Característica	*Kairos*	*OPTIC*	*TFD*	*POPF*	*LPG-td*	*ENHSP*	*Metric-FF*	*LAMA*
STRIPS y tipos	✓	✓	·	✓	✓	✓	✓	✓
Tipos `either`	✓	✓	·	✓	✓	·	✓	·
Constantes del dominio	✓	✓	·	✓	✓	✓	✓	✓
Precondiciones negativas	✓	·	·	·	✓	✓	✓	✓
Igualdad	✓	✓	·	✓	✓	✓	✓	✓
Precondiciones disyuntivas	✓	·	·	·	✓	✓	✓	✓
Precondiciones cuantificadas	✓	✓	·	✓	·	✓	·	✓
Efectos condicionales	✓	·	·	·	·	✓	·	✓
Objetivos negativos	✓	·	·	·	✓	✓	✓	✓
Predicados derivados (recursivos)	✓	·	·	·	✓	·	·	✓
Fluentes numéricos	✓	✓	·	✓	✓	!	✓	·
Costes de acción (`total-cost`)	✓	✓	·	✓	·	·	·	✓
Métrica sobre fluentes numéricos	✓	✓	·	✓	·	✓	·	·
Acciones durativas	✓	✓	✓	✓	✓	·	·	·
Concurrencia requerida	✓	✓	✓	✓	·	·	·	·
Desigualdades de duración	✓	✓	·	✓	·	·	·	·
Efectos continuos (`#t`)	✓	·	·	✓	·	·	·	·
Literales iniciales temporizados	✓	✓	·	✓	·	·	·	·
Preferencias (PDDL 3)	✓	✓	·	✓	·	✓	·	·
Restricciones de trayectoria (PDDL 3)	✓	✓	·	!	·	·	·	·
Fluentes de objeto (PDDL 3.1)	✓	·	·	·	·	·	·	·
Procesos (PDDL+)	✓	·	·	·	·	·	·	·

Tabla B.3.: Características de PDDL por planificador. ✓: un plan válido. ·: un error, una característica no admitida o ningún plan en diez segundos. !: un plan que no supera la validación. Las desigualdades de duración se probaron con `?duration` en los efectos.

La tabla prueba un problema pequeño por característica, así que una marca de verificación dice que el planificador lee la construcción y planifica con ella, no lo bien que le va con problemas grandes. Las notas del mismo documento explican algunas celdas:

- **TFD** solo planifica dominios con acciones durativas, así que falla en todas las filas clásicas.
- **OPTIC y POPF** rechazan un efecto o una condición numérica que no esté dentro de `(and ...)`: `:effect (increase (fuel) 4)` es

para ellos un error de sintaxis, y `:effect (and (increase (fuel) 4))` es correcto. No tienen precondiciones negativas, disyunciones ni efectos condicionales.
- **ENHSP** devolvió un plan inválido en la prueba numérica, y no acepta `either`.
- **Metric-FF** se ejecutó con su búsqueda clásica; su búsqueda por defecto necesita una métrica.
- **LAMA**, una configuración de Fast Downward, rechaza `either` en los parámetros de las acciones y no tiene fluentes numéricos.
- **LPG-td** no tiene precondiciones cuantificadas ni efectos condicionales.

Para el modelador, la tabla es el criterio de Portabilidad del método MODEL en una sola imagen: STRIPS sin más, con tipos, constantes e igualdad, funciona en todos los planificadores de la tabla que aceptan acciones sin duración, y casi cada característica que se le añade quita planificadores de la lista.

Soluciones a los ejercicios

C.

Este apéndice recoge, explicadas, las soluciones de los ejercicios propuestos al final de cada capítulo. Cada solución lleva el número de su ejercicio (capítulo, sección y número): la solución del Ejercicio 5.2.1 está en la sección del Capítulo 5, bajo el mismo número. El PDDL de las soluciones está en la carpeta `solutions` de los ejemplos de cada capítulo, probado como cualquier otro fichero del libro.

Muchos ejercicios piden un modelo, y un modelo se puede escribir de más de una manera. Una solución distinta de la que se da aquí no es incorrecta por eso: compruébala con la batería de pruebas del paso E del método MODEL y compárala con la impresa usando los cinco criterios del paso L.

C.1 Soluciones del Capítulo 1 529
C.2 Soluciones del Capítulo 2 533
C.3 Soluciones del Capítulo 3 540
C.4 Soluciones del Capítulo 4 547
C.5 Soluciones del Capítulo 5 553
C.6 Soluciones del Capítulo 6 558
C.7 Soluciones del Capítulo 7 564
C.8 Soluciones del Capítulo 8 569
C.9 Soluciones del Capítulo 9 574
C.10 Soluciones del Capítulo 10 577
C.11 Soluciones del Capítulo 11 583
C.12 Soluciones del Capítulo 12 588
C.13 Soluciones del Capítulo 13 594
C.14 Soluciones del Capítulo 14 598
C.15 Soluciones del Capítulo 15 602
C.16 Soluciones del Capítulo 16 605
C.17 Soluciones del Capítulo 17 609
C.18 Soluciones del Capítulo 18 615
C.19 Soluciones del Capítulo 19 622
C.20 Soluciones del Capítulo 20 628

C.1. Soluciones del Capítulo 1

Sección 1.1 — Decidir antes de actuar

Solución 1.1.1. (a) No: el termostato reacciona a la temperatura actual y no interviene ninguna secuencia. (b) Sí: los coches se bloquean unos a otros, así que el orden de los movimientos importa y puede ser necesario un movimiento que parece inútil (alejar un coche de la salida). (c) No, en el sentido del capítulo: ordenar se resuelve con un algoritmo fijo que funciona para cualquier lista; no hay que buscar nada para cada entrada nueva. (d) Sí, en pequeño: lo que entra primero acaba en el fondo, así que las acciones interactúan y el orden hay que decidirlo de antemano. (e) No: es una reacción. La prueba es siempre la misma: ¿depende la acción correcta de ahora de las acciones que vendrán después?

Solución 1.1.2. Entregando primero los medicamentos: de la base al vestíbulo, del vestíbulo a la farmacia, coger `meds1`, de la farmacia al vestíbulo, del vestíbulo al pasillo, del pasillo a la sala B, dejar `meds1`, de la sala B a la UCI, coger `sample1`, de la UCI a la sala B, de la sala B al pasillo, del pasillo al laboratorio, dejar `sample1`. Son 13 pasos: 9 movimientos, 2 acciones de coger y 2 de dejar. En el otro orden, el robot va a la UCI (4 movimientos), coge, va al laboratorio (3 movimientos), deja, va a la farmacia (3 movimientos), coge, va a la sala B (3 movimientos) y deja: 17 pasos. El primer orden es mejor porque la sala B está de camino a la UCI: la primera entrega termina a una habitación de donde empieza la segunda. Las dos peticiones son independientes como objetivos, pero no como planes, y un planificador tiene que considerarlas juntas para verlo.

Sección 1.2 — Un resolutor general más un modelo

Solución 1.2.1. Una respuesta posible para cada caso. (a) Té. Estado: si el hervidor tiene agua, si el agua está caliente, dónde está la bolsita de té, si la taza tiene agua o té. Acciones: llenar el hervidor (exige: hervidor vacío; cambio: el hervidor tiene agua); hervir (exige: el hervidor tiene agua; cambio: el agua está caliente); verter (exige: agua caliente, taza con la bolsita dentro; cambio: la taza tiene té).

Objetivo: la taza tiene té. (b) Ascensor. Estado: el piso en el que está la cabina, si las puertas están abiertas, quién va dentro, en qué piso espera cada persona. Acciones: subir o bajar un piso (exige: puertas cerradas); abrir y cerrar las puertas; una persona sube o se baja (exige: puertas abiertas, cabina y persona en el mismo piso). Objetivo: cada persona está en el piso que pidió. (c) 8-puzle. Estado: qué ficha hay en cada una de las nueve posiciones, una de las cuales está vacía. Acción: deslizar una ficha a la posición vacía (exige: la ficha está al lado; cambio: la ficha y la posición vacía se intercambian). Objetivo: cada ficha en su sitio. En los tres casos, fíjate en lo que el estado deja fuera (la marca del té, el peso de las personas): solo lo necesario para decidir.

Solución 1.2.2. (a) La situación: un lugar nuevo y sus conexiones. (b) La situación: un objetivo nuevo. (c) La descripción de lo que puede hacer el robot: la regla «un artículo cada vez» forma parte de las acciones. (d) La situación: existe un robot más; lo que puede hacer cualquier robot no cambia. (e) El planificador, o el ordenador en el que se ejecuta; el modelo se queda como está (aunque el Capítulo 3 mostrará que un modelo mejor también puede acelerar la planificación). En ningún caso escribe nadie una nueva secuencia de pasos.

Solución 1.2.3. Ocho. El artículo tiene que estar en la sala A, así que el robot no lo lleva, y el robot puede estar en cualquiera de los ocho lugares. El problema completo tiene 72 estados (el artículo en uno de 8 lugares con el robot en uno de 8, más el robot en uno de 8 lugares llevando el artículo), y el objetivo selecciona 8 de ellos. Un objetivo es un conjunto de estados, normalmente grande.

Sección 1.3 — Los supuestos clásicos

Solución 1.3.1. (a) Acciones instantáneas; Parte IV. (b) Estático: el mundo cambia en un momento conocido haga lo que haga el robot; Parte IV, Capítulo 13. (c) Determinista; no se levanta en PDDL, se trata ejecutando, observando y volviendo a planificar (Capítulo 20). (d) Finito: un número no es un hecho verdadero o falso; Parte III. (e) Totalmente conocido; no se levanta en PDDL (Capítulo 20). (f) Otra vez acciones instantáneas: si las acciones van una tras otra, nunca ocurre nada a la vez; Parte IV.

Solución 1.3.2. Lo bastante bueno: decidir qué habitaciones debe visitar el robot y en qué orden coger y dejar, cuando la planta es pequeña y cualquier ruta es aceptable. En el plan de `courier-p01.pddl`, el orden de los pasos es correcto sea cual sea la longitud de cada pasillo. Claramente malo: elegir entre dos robots, uno en la habitación de al lado y el otro en el extremo opuesto de un edificio largo. Para un modelo sin distancias, los dos están «a unos cuantos movimientos», y puede mandar al lejano si resulta que cruza menos habitaciones. El mismo modelo acierta en una decisión y falla en otra: un modelo se juzga frente a las decisiones que se le piden.

Sección 1.4 — La planificación y sus alternativas

Solución 1.4.1. Un programa para una petición:

```
serve(i, a, b):
  for p in route(position, a): move_to(p)
  pick(i)
  for p in route(a, b): move_to(p)
```

```
    drop(i)
```

(a) Dos artículos y dos bandejas: el programa necesita un caso nuevo que visite los dos orígenes antes del destino, una decisión sobre qué origen visitar primero y una regla para qué hacer cuando una petición nombra tres artículos. Cada caso es código nuevo, y los casos se combinan. (b) Conexiones cerradas: `route` tiene que buscar ahora en el mapa del día; si no existe ninguna ruta, el programa necesita una forma de decirlo, y si el robot se queda atrapado a medio camino con un artículo, otra regla. El programa se va convirtiendo, caso a caso, en un planificador escrito para un solo problema. En un modelo, (a) cambia una regla del dominio (cuántos artículos puede llevar un robot) y (b) quita algunas conexiones del problema; los planes, incluida la respuesta «no hay plan», salen sin más trabajo.

Solución 1.4.2. (a) Comportamiento escrito a mano: pocas situaciones, todas conocidas, y la máquina tiene que ser predecible. (b) Aprendizaje: nadie sabe escribir la regla que distingue en una imagen un tomate maduro de uno verde. (c) Planificación: las reglas son claras, las situaciones son incontables y los objetivos cambian durante todo el día. (d) Búsqueda programada a mano (o una solución calculada una vez y guardada): un único problema fijo no compensa el esfuerzo de una herramienta general, aunque un planificador también lo resolvería. En (d) caben opiniones razonables distintas; lo que importa es el motivo que se da.

Sección 1.6 — Dominio, problema y PDDL

Solución 1.6.1. Dominio: (b) y (e), que son reglas sobre lo que puede hacer cualquier robot. Problema: (a), (c), (d) y (f), que describen una planta, un día y una petición. El mapa (a) es el caso que sorprende: parece una regla permanente, pero es un hecho de esta planta, y mantenerlo en el problema permite que el mismo dominio sirva para cualquier edificio.

Solución 1.6.2. (a) PDDL 2.1: números. (b) PDDL 2.2: hechos que cambian en momentos conocidos (y PDDL 2.1 para el propio reloj: las acciones necesitan duraciones para que «8:00» signifique algo). (c) PDDL 3.0: una preferencia. (d) PDDL 2.1: acciones con duración que se solapan. (e) PDDL+: el calentamiento es un proceso que nadie inicia, y el deterioro, un evento. (f) PDDL 3.0: una restricción sobre todo el plan. El requisito (f) también se puede escribir en PDDL 1.2 como una condición de la acción de moverse; el Capítulo 15 compara las dos formas.

Sección 1.7 — Primer contacto con Kairos

Solución 1.7.1. El objetivo es la última línea del problema, y basta con cambiar allí `ward-a` por `icu`. El resultado, en la carpeta `solutions` de `examples/ch01`, es el fichero `courier-s01.pddl`. La predicción a partir del mapa: los mismos tres pasos para conseguir los medicamentos (dos movimientos y coger), después farmacia, vestíbulo, pasillo, sala B, UCI (cuatro movimientos) y la entrega: 8 pasos.

```
$ kairos courier-domain.pddl solutions/courier-s01.pddl
; cost = 8
(move r1 dock lobby)
(move r1 lobby pharmacy)
```

```
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds1 icu)
```

Solución 1.7.2. El fichero es `solutions/courier-s02.pddl`. El planificador responde que el problema no tiene plan:

```
$ kairos courier-domain.pddl solutions/courier-s02.pddl
note: the goal needs atoms that no action can reach
(item-at meds1 ward-a) can never be reached:
  drop adds it, but needs (carrying ?r - object meds1), which
     never holds
    pick adds it, but needs (free ?r - object), which never
     holds
      drop adds it, but needs (carrying ?r - object ?i -
     object), which never holds
         pick adds it, but needs (free ?r - object), which
     never holds [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

Leído de arriba abajo: los medicamentos solo llegan a la sala A si alguien los deja allí; dejarlos exige un robot que los lleve; llevarlos solo viene de cogerlos; y coger exige un robot que esté *libre*. La única forma de quedar libre es dejar algo, lo que exige llevarlo: un círculo sin entrada. El texto borrado era la afirmación «el robot `r1` no lleva nada al principio». Sin ella, el planificador no supone que la bandeja del robot está vacía: solo sabe lo que dice el problema, y para él un robot que no se declara libre nunca puede coger nada. El Capítulo 2 le pone nombre a esta regla: la hipótesis del mundo cerrado.

Solución 1.7.3. Hay un plan válido en `solutions/courier-p02.plan`: es el plan que imprimió Kairos en el capítulo, un paso por línea. Se acepta cualquier plan que respete las reglas, también uno más largo que dé vueltas. El fichero `courier-p02-wrong.plan` es el mismo plan con sus pasos tercero y cuarto intercambiados:

```
$ kairos validate courier-domain.pddl courier-p02.pddl \
    solutions/courier-p02-wrong.plan
solutions/courier-p02-wrong.plan:5: error: step 3 (pick r1
     sample1 icu) at 2.000: precondition false: (robot-at r1
     icu) is false [validate.precondition_failed]
 5 | (pick r1 sample1 icu)
plan invalid
```

El mensaje da el fichero y la línea (la 5, porque antes van dos líneas de comentario), el paso del plan (el tercero) y su instante (los pasos se numeran desde el instante 0, una unidad cada uno), y el motivo: la acción de coger tiene una precondición, que el robot esté en la UCI, y en ese momento es falsa. El validador se detiene en el primer paso que no se puede hacer; lo que viene después no se comprueba.

Sección 1.8 — El método MODEL: cinco pasos y cinco criterios

Solución 1.8.1. Una ficha posible. *Historia*: un robot aspirador limpia las habitaciones de un piso y vuelve a su base. *Agentes*: el aspirador, solo. *Éxito*: todas las habitaciones están limpias y el robot está en su base.

Alcance, dentro: las habitaciones, cuáles están conectadas, cuáles están limpias, dónde está el robot. Fuera: la batería, el depósito de polvo, los muebles y el recorrido exacto dentro de una habitación, las personas y los animales, las puertas cerradas, lo sucia que está cada habitación. *Supuestos*: los clásicos; una habitación limpia sigue limpia. Un mal plan por dejar fuera la batería: en un piso grande, el plan limpia todas las habitaciones en un solo recorrido, y el robot real se para en la cuarta habitación con la batería agotada, lejos de su base. Si eso puede pasar, la batería tiene que entrar en el alcance.

Solución 1.8.2. (a) Paso M: es una línea del Alcance. (b) Paso E: es una de las cuatro pruebas. (c) Paso D. (d) Paso O. (e) Paso L: una codificación alternativa, probada y comparada. (f) Paso E: cada plan de la batería de pruebas se lee.

Solución 1.8.3. *Fidelidad*: pobre para la mayoría de las peticiones. Un plan «moverse al lado de las salas, dejar» no dice qué habitaciones cruzar ni en qué parte del lado de las salas acaba el artículo; una petición para la sala A y una para la UCI se convierten en la misma petición. También admite planes imposibles: coge un artículo en la farmacia mientras el robot está en la base, porque los dos son el mismo lugar. *Tamaño*: diminuto, 2 lugares y 2 movimientos. *Guía*: trivial, todos los planes tienen tres o cuatro pasos. *Claridad*: los dos nombres son fáciles, pero nadie puede leer una entrega en el plan. *Portabilidad*: clásica. Veredicto: solo es lo bastante buena para una pregunta como «¿cuántas veces tiene que cruzar hoy el robot el vestíbulo?», en la que los dos lados son lo único que importa. Para las entregas pierde fidelidad enseguida. Fallan tanto lo demasiado grueso como lo demasiado fino; el nivel adecuado lo fijan las decisiones que se necesitan, como decía el veredicto del capítulo.

C.2. Soluciones del Capítulo 2

Los ficheros de estas soluciones están en `examples/ch02/solutions`; las órdenes se ejecutan desde `examples/ch02`.

Sección 2.1 — Listas dentro de listas: la forma de PDDL

Solución 2.1.1. (a) Falta un paréntesis de cierre: `(and (on desk) (off hall))`. (b) La coma no es un separador; el espacio, sí: `(on desk)`. (c) El `and` y sus partes deben formar una lista propia: `(:goal (and (on desk) (off hall)))`. (d) El punto y coma empieza un comentario, así que el planificador lee `(on desk)` y nunca ve el resto de la línea; lo que se quería escribir era `((on desk) (off hall))` o, más probablemente, `(and (on desk) (off hall))`. Este último error es el peor de los cuatro, porque puede no dar ningún error: un hecho ha desaparecido sin que nadie avise. El árbol de (a) tiene la raíz `and` con dos hijos, `on` y `off`; `desk` cuelga del primero y `hall` del segundo. Es la Figura 2.1 sin su nodo superior.

Sección 2.2 — Predicados, objetos y hechos

Solución 2.2.1. (a) Cada uno de los seis hechos está en el conjunto o no está: $2^6 = 64$ conjuntos. (b) Un estado con sentido tiene, para cada lámpara, exactamente uno de sus dos hechos: $2^3 = 8$ estados. (c) {`(on hall)`, `(off hall)`, `(off desk)`}: la lámpara del recibidor

está encendida y apagada, y la del porche no está ni una cosa ni otra. El `:init` de un problema lo produce si enumera los dos hechos de la lámpara del recibidor y se olvida de la del porche. El lenguaje acepta los 64 conjuntos; mantener el modelo dentro de los 8 que tienen sentido es tarea del estado inicial y de las acciones.

Solución 2.2.2. (a) Falso: no está en el estado (y no debería estar, porque el robot lleva la muestra). (b) Falso: un artículo que va encima del robot no está en ninguna habitación. (c) Cierto: es uno de los hechos estáticos del mapa. (d) Falso: esa conexión no figura en la lista. (e) Falso. Ninguno es desconocido: con la hipótesis del mundo cerrado, todo hecho es cierto (está en el estado) o falso (no está).

Sección 2.3 — Acciones: precondiciones y efectos

Solución 2.3.1. (a) `(move r1 lobby lab)`: no es aplicable, `(connected lobby lab)` es falso. `(pick r1 meds1 lobby)`: aplicable. `(pick r1 sample1 lab)`: no es aplicable, `(robot-at r1 lab)` es falso. `(drop r1 meds1 lobby)`: no es aplicable, `(carrying r1 meds1)` es falso. `(move r1 lobby dock)`: aplicable. (b) Tras la recogida: `(robot-at r1 lobby)`, `(carrying r1 meds1)`, `(item-at sample1 lab)`; la recogida borró `(free r1)` y `(item-at meds1 lobby)`. Tras el movimiento: `(robot-at r1 corridor)`, `(carrying r1 meds1)`, `(item-at sample1 lab)`.

Solución 2.3.2. Al efecto le falta `(not (item-at ?i ?l))`. Tras la recogida, el estado tiene `(carrying r1 meds1)` y sigue teniendo `(item-at meds1 pharmacy)`: la caja está a la vez en el robot y en la estantería. Tras el movimiento y la descarga tiene `(item-at meds1 pharmacy)` y `(item-at meds1 lobby)`: la caja se ha duplicado. Ninguna de las cuatro pruebas del capítulo lo nota por su resultado, porque los planes conservan su longitud; es un borrado olvidado que no acorta nada en ellos. Lo cazan dos cosas. Leer el plan sobre el estado, como en la Tabla 2.3, muestra enseguida el hecho que sobra. Y una prueba nueva del tipo «no debe tener plan» dirigida a la regla «un artículo está en un solo sitio»: un objetivo que pida `(and (item-at meds1 lobby) (item-at meds1 pharmacy))` tiene que ser irresoluble, y con este dominio tiene un plan de tres pasos. La batería de pruebas crece con cada regla de la ficha del modelo.

Sección 2.4 — El problema: objetos, estado inicial y objetivo

Solución 2.4.1. El fichero es `switch-s01.pddl`: el objetivo pasa a ser `(and (on hall) (on desk) (on porch))`. La lámpara del recibidor ya está encendida, así que bastan dos pasos, en cualquier orden:

```
$ kairos switch-domain.pddl solutions/switch-s01.pddl
; cost = 2
(switch-on porch)
(switch-on desk)
```

Un hecho del objetivo que ya es cierto no cuesta nada, siempre que ninguna acción del plan lo borre.

Solución 2.4.2. El fichero es `switch-s02.pddl`. No hay plan:

```
$ kairos switch-domain.pddl solutions/switch-s02.pddl
note: the goal needs atoms that no action can reach
(on porch) can never be reached:
```

```
  switch-on adds it, but needs (off porch), which never holds
    switch-off adds it, but needs (on porch), which never
     holds [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

Solo `switch-on` añade `(on porch)`, y necesita `(off porch)`. Ese hecho no está en el estado inicial, así que es falso, y la única acción que lo añade, `switch-off`, necesita `(on porch)`: un círculo vicioso. Para nosotros, una lámpara que no está encendida está apagada. Para el planificador, `on` y `off` no tienen relación, y una lámpara sin ninguno de los dos hechos queda fuera del alcance de todas las acciones. El estado inicial tiene que ser completo.

Solución 2.4.3. La corriente es una cosa más que puede estar dada o cortada, y no es una propiedad de ninguna lámpara: dos predicados sin parámetros y una acción sin parámetros.

```
  (:predicates
    (on ?l) (off ?l)
    (power-on) (power-off))   ; predicates without parameters

  (:action power-up
    :parameters ()
    :precondition (power-off)
    :effect (and (power-on) (not (power-off))))

  (:action switch-on
    :parameters (?l)
    :precondition (and (off ?l) (power-on))
    :effect (and (on ?l) (not (off ?l))))
```

`switch-off` no cambia. El problema `switch-power-p01.pddl` empieza con `(power-off)` y tres lámparas apagadas:

```
$ kairos solutions/switch-power-domain.pddl \
    solutions/switch-power-p01.pddl
; cost = 3
(power-up)
(switch-on porch)
(switch-on desk)
```

`power-up` está en el plan aunque ningún objetivo menciona la corriente: el planificador ha descubierto que las lámparas la necesitan. `(power-on)` es una precondición de `switch-on` y ningún efecto la borra, así que un solo `power-up` sirve para las dos lámparas.

Sección 2.5 — Modelar el mundo del mensajero

Solución 2.5.1. El fichero es `courier-s01.pddl`. El mejor orden recoge primero la comida: del pasillo al vestíbulo, coger, del vestíbulo al pasillo y a la sala B, dejar (5 pasos); después, de la sala B al pasillo y al laboratorio, coger, del laboratorio al pasillo, a la sala B y a la UCI, dejar (7 pasos). Doce pasos. Empezar por la muestra cuesta 13, porque el robot cruza la sala B con la bandeja vacía y tiene que volver a ella. Kairos imprime los dos, primero el plan de 13 pasos y después el mejor:

```
$ kairos courier-domain.pddl solutions/courier-s01.pddl
; cost = 13
(move r1 corridor lab)
...
; better plan 2
; cost = 12
```

```
(move r1 corridor lobby)
(pick r1 meal1 lobby)
(move r1 lobby corridor)
(move r1 corridor ward-b)
(drop r1 meal1 ward-b)
(move r1 ward-b corridor)
(move r1 corridor lab)
(pick r1 sample1 lab)
(move r1 lab corridor)
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 sample1 icu)
```

Solución 2.5.2. Solo cambia el problema: el mapa son datos. Se quita el hecho `(connected ward-a corridor)` y se mantiene `(connected corridor ward-a)` (`courier-s02.pddl`). El dominio no necesita nada, porque `connected` siempre ha tenido dirección. Con el objetivo `(and (item-at meds1 ward-a) (robot-at r1 dock))`:

```
$ kairos courier-domain.pddl solutions/courier-s02.pddl
note: no plan: every state reachable from the initial one was
     searched (106 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Cada mitad del objetivo se puede alcanzar, y la nota no nombra ningún hecho inalcanzable. Pero los medicamentos solo entran en la sala A si entra el robot, y un robot que ha entrado nunca sale: las dos mitades no pueden cumplirse a la vez. La sala A es un *callejón sin salida*, un estado desde el que ya no se puede alcanzar el objetivo. Con el objetivo `(item-at meds1 ward-a)` solo, el problema tiene su plan habitual de 7 pasos, y el robot se queda en la sala para siempre.

Solución 2.5.3. La fila de la tabla de acciones. Parámetros: el robot que da, el que recibe, el artículo y el lugar. Se aplica cuando: los dos robots están en el lugar, el que da lleva el artículo y el que recibe está libre. Cambia, propiedad por propiedad: dónde están los robots, no cambia; dónde está el artículo, no cambia (no estaba en ninguna parte y sigue sin estarlo); qué se lleva, el que recibe lleva el artículo y el que da ya no; libre, el que da pasa a estar libre y el que recibe deja de estarlo.

```
37  (:action hand-over
38    :parameters (?giver ?taker ?i ?l)
39    :precondition (and (robot ?giver) (robot ?taker) (item ?i)
40                       (location ?l)
41                       (robot-at ?giver ?l) (robot-at ?taker ?l)
42                       (carrying ?giver ?i) (free ?taker))
43    :effect (and (not (carrying ?giver ?i)) (free ?giver)
44                 (carrying ?taker ?i) (not (free ?taker)))))
```

```
$ kairos solutions/courier-handover-domain.pddl \
    solutions/courier-handover-p01.pddl
; cost = 2
(move r2 lobby pharmacy)
(hand-over r1 r2 meds1 pharmacy)
```

Sin la acción nueva, el plan necesitaría que `r1` dejara el artículo y que `r2` lo cogiera: tres pasos. El planificador usa `hand-over` porque da un plan más corto, y por ninguna otra razón. Observa también que un robot no puede pasarse un artículo a sí mismo: la acción necesitaría `(carrying r1 meds1)` y `(free r1)` a la vez, y las acciones

nunca hacen ciertos los dos.

Sección 2.6 — Ensayar el modelo

Solución 2.6.1. `courier-p02-long.plan` da un paseo hasta la base de carga y vuelve antes de hacer el trabajo: siete pasos, todos aplicables, y el objetivo se cumple al final.

```
$ kairos validate courier-domain.pddl courier-p02.pddl \
    solutions/courier-p02-long.plan
plan valid, makespan 7
$ kairos validate courier-domain.pddl courier-p02.pddl \
    solutions/courier-p02-nopick.plan
solutions/courier-p02-nopick.plan:3: error: step 2 (drop r1
     meds1 lobby) at 1.000: precondition false: (carrying r1
     meds1) is false [validate.precondition_failed]
 3 | (drop r1 meds1 lobby)
plan invalid
```

Un plan válido no tiene por qué ser bueno: la validez dice que el modelo permite el plan y que este alcanza el objetivo, y nada sobre su longitud. En el segundo plan el movimiento es correcto (un robot siempre puede desplazarse) y el fallo llega al dejar el artículo, en el segundo paso, cuya precondición `(carrying r1 meds1)` no ha hecho cierta ninguna acción.

Solución 2.6.2. *Trivial*: una lámpara apagada que hay que encender; esperado, un paso. *Típica*: `switch-p01.pddl`; esperado, dos pasos. *Sin plan*: una lámpara que el estado inicial olvida, como en la Solución 2.4.2; esperado, irresoluble. *Trampa*: la regla de este mundo es que una lámpara está encendida o apagada, y nunca las dos cosas, así que pedimos las dos (`switch-s03.pddl`, objetivo `(and (on desk) (off desk))`); esperado, irresoluble.

```
$ kairos switch-domain.pddl solutions/switch-s03.pddl
note: no plan: every state reachable from the initial one was
     searched (2 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Si `switch-on` olvidara su efecto de borrado, esta última prueba encontraría un plan de un paso, y es la única de las cuatro que lo notaría.

Sección 2.7 — Dos clásicos: el mundo de bloques y gripper

Solución 2.7.1. El fichero es `blocks-s01.pddl`. Todos los bloques salvo el último tienen que salir de la torre antigua y ponerse en la nueva. `d` va a la mesa (unstack, put-down), `c` y `b` van de torre a torre (unstack, stack, dos veces), y `a` ya está sobre la mesa (pick-up, stack): 8 pasos.

```
$ kairos blocks-domain.pddl solutions/blocks-s01.pddl
; cost = 8
(unstack d c)
(put-down d)
(unstack c b)
(stack c d)
(unstack b a)
(stack b c)
(pick-up a)
```

```
(stack a b)
```

Solución 2.7.2. (a) Si apilar sobre `?y` no borra `(clear ?y)`, se puede apilar un segundo bloque sobre el mismo: dos bloques directamente encima de un tercero. (b) El objetivo `(and (on a c) (on b c))` con tres bloques sobre la mesa. Con el dominio erróneo (`blocks-noclear-domain.pddl`) tiene plan; con el correcto (`blocks-s02.pddl`) no tiene ninguno.

```
$ kairos solutions/blocks-noclear-domain.pddl \
    solutions/blocks-noclear-p01.pddl
; cost = 4
(pick-up b)
(stack b c)
(pick-up a)
(stack a c)
$ kairos blocks-domain.pddl solutions/blocks-s02.pddl
note: no plan: every state reachable from the initial one was
     searched (44 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

(c) Una prueba de la trampa: pide un estado que rompe una regla del mundo (un solo bloque encima de cada bloque) y espera que no haya plan. El problema típico `blocks-p01.pddl` da los mismos seis pasos con los dos dominios y no nota nada.

Solución 2.7.3. *Ficha del modelo.* Historia: un ferri lleva coches entre puertos. Éxito: cada coche espera en su puerto de destino. Alcance y supuestos: un ferri, como mucho un coche a bordo, todos los puertos accesibles desde todos los demás, sin tiempos ni distancias. *Tabla de estado.* Puerto y coche, predicados de tipo, estáticos. Dónde está el ferri, dinámico, `(ferry-at ?p)`. Dónde espera un coche, dinámico, `(car-at ?c ?p)`. Qué coche va a bordo, dinámico, `(on-board ?c)`. El ferri no lleva ningún coche, dinámico, `(empty-ferry)`. Hay un solo ferri, así que no es un objeto, igual que el robot de gripper. *Tabla de acciones.* `sail`: el ferri está en el origen; pasa a estar en el destino y no en el origen. `board`: coche y ferri en el mismo puerto, ferri vacío; el coche pasa a ir a bordo y ya no está en el puerto, y el ferri ya no está vacío. `debark`: el coche va a bordo y el ferri está en el puerto; la imagen especular.

```
13  (:action sail
14    :parameters (?from ?to)
15    :precondition (and (port ?from) (port ?to) (ferry-at ?from))
16    :effect (and (ferry-at ?to) (not (ferry-at ?from))))
17
18  (:action board
19    :parameters (?c ?p)
20    :precondition (and (car ?c) (port ?p) (car-at ?c ?p)
21                       (ferry-at ?p) (empty-ferry))
22    :effect (and (on-board ?c)
23                 (not (car-at ?c ?p)) (not (empty-ferry))))
24
25  (:action debark
26    :parameters (?c ?p)
27    :precondition (and (car ?c) (port ?p) (on-board ?c)
28                       (ferry-at ?p))
29    :effect (and (car-at ?c ?p) (empty-ferry)
30                 (not (on-board ?c)))))
```

```
$ kairos solutions/ferry-domain.pddl solutions/ferry-p01.pddl
; cost = 6
(board car1 north)
(sail north south)
(debark car1 south)
(board car2 south)
(sail south north)
(debark car2 north)
```

El ferri nunca navega vacío: cada travesía lleva un coche. Para el paso E, la batería de pruebas añadiría una prueba de la trampa, dos coches en el mismo puerto con el mismo destino, que debe necesitar tres travesías y no una.

Sección 2.8 — El método MODEL: la primera pasada completa

Solución 2.8.1. Bloques: los cinco predicados (`on`, `on-table`, `clear`, `holding`, `hand-empty`) aparecen en efectos; todos son dinámicos y no hay nada estático. Gripper: `room`, `ball` y `gripper` son estáticos; `at-robby`, `at`, `empty` y `holds` son dinámicos. `(unstack ?x ?y)`, propiedad por propiedad: `on`, falso para `?x` sobre `?y`; `on-table`, sin tocar; `clear`, cierto para `?y` y falso para `?x`; `holding`, cierto para `?x`; `hand-empty`, falso. Cambian cuatro de las cinco propiedades, en cinco efectos: el listado tiene exactamente esos cinco.

Solución 2.8.2. El dominio sin mano:

```
  (:action move                ; from block ?from to block ?to
    :parameters (?x ?from ?to)
    :precondition (and (on ?x ?from) (clear ?x) (clear ?to))
    :effect (and (on ?x ?to) (clear ?from)
                 (not (on ?x ?from)) (not (clear ?to))))

  (:action move-to-table       ; from block ?from to the table
    :parameters (?x ?from)
    :precondition (and (on ?x ?from) (clear ?x))
    :effect (and (on-table ?x) (clear ?from)
                 (not (on ?x ?from))))

  (:action move-from-table     ; from the table to block ?to
    :parameters (?x ?to)
    :precondition (and (on-table ?x) (clear ?x) (clear ?to))
    :effect (and (on ?x ?to)
                 (not (on-table ?x)) (not (clear ?to)))))
```

```
$ kairos solutions/blocks-move-domain.pddl \
    solutions/blocks-move-p01.pddl
; cost = 3
(move-to-table c a)
(move-from-table b c)
(move-from-table a b)
```

Fidelidad: los dos producen solo disposiciones reales de bloques. El modelo sin mano no puede hablar de la mano: ningún objetivo ni condición puede mencionar un bloque sostenido, y si el mundo tuviera dos manos el modelo no podría decirlo. *Tamaño*: para los tres bloques del problema, `kairos -v` da 24 acciones instanciadas con cuatro acciones y 45 sin mano. La acción `move` tiene tres parámetros, así que ella sola da $3^3 = 27$ acciones instanciadas, y con n bloques, n^3, frente a n^2 para `stack` y `unstack`. (Las 27 incluyen sinsentidos como `(move a a a)`, que nunca es aplicable; STRIPS no puede decir

que dos parámetros deben ser distintos, y el Capítulo 4 sí puede.) *Guía*: los planes son la mitad de largos, 3 pasos frente a 6, y los estados en los que la mano sostiene un bloque no existen. *Claridad*: los dos se leen bien; el que no tiene mano es más corto. *Portabilidad*: los dos son `:strips`. Veredicto: sin mano cuando solo importa la disposición y el número de bloques es pequeño; cuatro acciones cuando la mano forma parte de la historia, o cuando hay muchos bloques, porque el cubo de su número crece deprisa. Es el mismo compromiso que el de las codificaciones A y B del mundo del mensajero: fusionar dos acciones en una acorta los planes y añade parámetros.

Sección 2.9 — Errores comunes

Solución 2.9.1. El efecto de `drop` no añade `(free ?r)`. Tras dejar el primer artículo, el robot no está libre y nada volverá a dejarlo libre, así que nunca puede coger nada por segunda vez. (a) `courier-p01`, `courier-p02` y `courier-p04` siguen dando las respuestas correctas: los dos primeros solo necesitan una recogida, y el tercero tiene que ser irresoluble y lo es. `courier-p03` y `courier-p05` necesitan dos recogidas y fallan. (b) El planificador responde que no tienen plan (los ficheros son `courier-nofreeadd-domain.pddl` y `courier-nofreeadd-p05.pddl`):

```
$ kairos solutions/courier-nofreeadd-domain.pddl \
    solutions/courier-nofreeadd-p05.pddl
note: no plan: every state reachable from the initial one was
     searched (11 states) and none satisfies the goal. [
     orchestrator.why]
kairos: unsolvable (exhausted)
```

Once estados: el mundo del robot se acaba justo después de su primera entrega. (c) Una precondición que falta hace que una acción sea aplicable más veces que en el mundo: el modelo permite demasiado, y el planificador encuentra planes demasiado cortos. Un efecto de adición que falta quita algo que el mundo sí devuelve: el modelo permite demasiado poco, y los planes desaparecen. El primer tipo de error lo encuentran los problemas que no deben tener plan o que deben tenerlo largo; el segundo, los problemas típicos que deben tener plan. Por eso la batería de pruebas necesita los dos.

C.3. Soluciones del Capítulo 3

Los ficheros de estas soluciones están en `examples/ch03/solutions`. Las cifras se obtuvieron con `--threads 1`.

Sección 3.1 — Qué significa un modelo

Solución 3.1.1. Sin los hechos estáticos, el estado s_4 tiene dos hechos: `(robot-at r1 lobby)` y `(carrying r1 meds1)`. El hecho `(free r1)` no está en él, ni tampoco ningún hecho `item-at`: un artículo que se lleva encima no está en ninguna habitación. Hay tres acciones instanciadas aplicables:

- `(move r1 lobby pharmacy)` sustituye `(robot-at r1 lobby)` por `(robot-at r1 pharmacy)`; el robot sigue llevando los medicamentos. Es s_3.
- `(move r1 lobby corridor)` lo sustituye, en cambio, por `(robot-at r1 corridor)`. Es s_5.
- `(drop r1 meds1 lobby)` borra `(carrying r1 meds1)` y añade `(item-at meds1 lobby)` y `(free r1)`; el robot se queda en el vestíbulo. Es s_6, un estado con tres hechos.

No es aplicable ninguna recogida (el robot no está libre) ni ninguna otra descarga (el robot está en el vestíbulo).

Solución 3.1.2. Cualquier rodeo sirve. El fichero `courier-p01-long.plan` empieza yendo al pasillo y volviendo:

```
(move r1 lobby corridor)
(move r1 corridor lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(drop r1 meds1 corridor)
```

`kairos validate` responde `plan valid, makespan 7`. Un plan más largo es un plan: la definición pide acciones aplicables y un estado final que satisfaga el objetivo, y no dice nada de la longitud. En la Figura 3.2 es un camino que sale de s_0, vuelve a él y después sigue el camino corto. Preferir planes cortos es otra cuestión, la de la Sección 3.6.

Solución 3.1.3. La primera acción es aplicable en s_0 y lleva a s_1. La segunda, la descarga, necesita `(carrying r1 meds1)`, que no está en s_1: la secuencia se rechaza en el paso 2, y la tercera acción ni siquiera se mira. Incluso con la descarga y la recogida en el orden correcto, la secuencia no sería un plan: sería aplicable hasta el final y fallaría en el objetivo. El validador dice (el 3 tras el nombre del fichero es la línea del fichero, que empieza con un comentario):

```
$ kairos validate courier-domain.pddl courier-p01.pddl solutions/
     courier-p01-early.plan
solutions/courier-p01-early.plan:3: error: step 2 (drop r1 meds1
     pharmacy) at 1.000: precondition false: (carrying r1 meds1) is
     false [validate.precondition_failed]
 3 | (drop r1 meds1 pharmacy)
plan invalid
```

Sección 3.2 — Instanciación: de variables a objetos

Solución 3.2.1. Átomos: `robot-at` para 1 robot y 8 habitaciones (8), `item-at` para 2 artículos y 8 habitaciones (16), `carrying` para 1 robot y 2 artículos (2), `free` (1): 27. Acciones instanciadas: 14 movimientos (uno por cada hecho `connected`), $1 \times 2 \times 8 = 16$ recogidas y 16 descargas: 46. La línea `[ground]` de `kairos -v` dice `27 atoms` y `46 actions`.

Solución 3.2.2. Numeremos los discos desde el más pequeño, 1, hasta el más grande, n. El disco i puede descansar sobre los $n - i$ discos mayores y sobre las 3 varillas: $n - i + 3$ cosas. Tanto `?from` como `?to` recorren esas cosas, así que el disco i tiene $(n - i + 3)^2$ instancias, y

el total es

$$\sum_{i=1}^{n}(n-i+3)^2 = 3^2 + 4^2 + \cdots + (n+2)^2.$$

Para 4 discos: $9+16+25+36 = 86$. Para 6 discos: $86+49+64 = 199$. Ambas cifras son las que indica Kairos. Para 2 discos la fórmula da $9+16 = 25$. Las instancias con `?from` igual a `?to` no pueden aplicarse nunca (el disco tendría que estar sobre una cosa que está despejada): hay 4 para `d1` y 3 para `d2`, así que 7 de las 25. Cada una de las otras 18 es aplicable en algún estado alcanzable.

Solución 3.2.3. El dominio es `gripper-domain.pddl`:

```
;; Exercise 3.2.3: gripper. A robot with two grippers carries
;; balls from one room to another.
(define (domain gripper)
  (:requirements :strips)

  (:predicates
    (room ?x) (ball ?x) (gripper ?x)
    (at-robot ?r)           ; the robot is in room ?r
    (at ?b ?r)              ; ball ?b is in room ?r
    (free ?g)               ; gripper ?g holds nothing
    (holding ?g ?b))        ; gripper ?g holds ball ?b

  (:action move
    :parameters (?from ?to)
    :precondition (and (room ?from) (room ?to) (at-robot ?from))
    :effect (and (not (at-robot ?from)) (at-robot ?to)))

  (:action pick
    :parameters (?b ?r ?g)
    :precondition (and (ball ?b) (room ?r) (gripper ?g)
                       (at ?b ?r) (at-robot ?r) (free ?g))
    :effect (and (not (at ?b ?r)) (not (free ?g))
                 (holding ?g ?b)))

  (:action drop
    :parameters (?b ?r ?g)
    :precondition (and (ball ?b) (room ?r) (gripper ?g)
                       (holding ?g ?b) (at-robot ?r))
    :effect (and (not (holding ?g ?b))
                 (at ?b ?r) (free ?g))))
```

Listado C.1: Gripper en STRIPS sin tipos.

Estimación para `gripper-p01.pddl` (2 habitaciones, 2 pinzas, 4 bolas). `move`: 2 posibilidades para cada habitación, 4 instancias; nada en STRIPS, tal como lo conocemos, dice que las dos habitaciones son distintas, así que `(move rooma rooma)` está entre ellas, e incluso es aplicable (no hace nada). `pick`: $4 \times 2 \times 2 = 16$. `drop`: 16. Total: 36. Átomos: `at-robot` 2, `at` 8, `free` 2, `holding` 8: 20. Kairos informa de `20 atoms` y `36 actions`, un primer plan de 13 acciones y un mejor plan de 11: cinco acciones por cada par de bolas (dos recogidas, un movimiento, dos descargas) y un movimiento de vuelta entre los dos pares.

Sección 3.3 — Por qué explota el espacio

Solución 3.3.1. Cada uno de los dos robots está en una de 8 habitaciones: $8^2 = 64$ formas. Para los cuatro artículos hay tres casos. Nadie lleva nada: $8^4 = 4096$ formas. Un robot lleva un artículo: 2 robots por 4 artículos, y los otros tres artículos en habitaciones: $8 \cdot 8^3 = 4096$. Los dos llevan algo: 4 artículos para el primer robot, 3 para el segundo, los otros dos en habitaciones: $12 \cdot 8^2 = 768$. Total: $64 \cdot (4096 + 4096 + 768) = 573\,440$.

Con tres robots: $8^3 = 512$ formas de situarlos, y $8^4 + (3 \cdot 4) \cdot 8^3 + (3 \cdot 4 \cdot 3) \cdot 8^2 + (4 \cdot 3 \cdot 2) \cdot 8 = 4096 + 6144 + 2304 + 192 = 12\,736$ formas de situar los artículos (cero, uno, dos o tres robots cargados). Total: 6 520 832 estados. Un robot más, once veces más estados.

Solución 3.3.2. El tablero de 4×4 tiene 24 pares de casillas vecinas (4×3 en las filas y otros tantos en las columnas), así que 48 hechos `adjacent` y $15 \times 48 = 720$ acciones instanciadas. Átomos: `tile-at` para 15 fichas y 16 casillas (240) más 16 `empty`: 256. Estados alcanzables: $16!/2 = 10\,461\,394\,944\,000$, unos 10^{13}. A 52 bytes cada uno ocuparían unos $5{,}4 \times 10^{14}$ bytes: más de 500 terabytes. El modelo es cuatro veces más grande que el del 8-puzle; el espacio de estados es 58 millones de veces mayor. Ningún planificador demuestra nada sobre el 15-puzle agotándolo; un primer plan es otra cosa, y lo proporciona la heurística.

Sección 3.4 — Búsqueda hacia delante

Solución 3.4.1. Nombremos un estado por las varillas de los dos discos: (a, b) significa `d1` en la varilla a y `d2` en la varilla b (cuando $a = b$, `d1` está encima de `d2`). Hay $3 \times 3 = 9$ estados. El disco pequeño siempre puede ir a cualquiera de las otras dos varillas; el grande solo puede moverse cuando el pequeño está en otra varilla, y solo a la varilla en la que no está el pequeño. Así que cada estado tiene dos o tres sucesores. El inicio es (1,1) y el objetivo (3,3).

Búsqueda en anchura, con el bucle de la Sección 3.4: expande (1,1) y encuentra (2,1) y (3,1). Expande (2,1) y encuentra (2,3); expande (3,1) y encuentra (3,2). Expande (2,3) y encuentra (1,3) y (3,3); expande (3,2) y encuentra (1,2) y (2,2). Expande (1,3), nada nuevo. El siguiente estado que sale de la lista es (3,3): el objetivo. Se expandieron seis estados y se encontraron los nueve. El plan es el camino (1,1), (2,1), (2,3), (3,3), tres movimientos, que es lo que imprime Kairos para `hanoi-p01`:

```
(move d1 d2 peg2)
(move d2 peg1 peg3)
(move d1 peg2 d2)
```

Sección 3.5 — Heurísticas: plantear un problema más sencillo

Solución 3.5.1. Hacia delante: al principio el robot está en el pasillo y libre, y los medicamentos están en la farmacia. Un movimiento añade el robot en el vestíbulo; otro lo añade en la farmacia; la recogida añade `carrying`; una descarga añade el hecho del objetivo. Hacia atrás, el plan relajado es

```
(move r1 corridor lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(drop r1 meds1 corridor)     ; the robot never left the corridor
```

Cuatro acciones; la distancia real es 6. La relajación olvida todo el camino de vuelta, dos movimientos, porque `(robot-at r1 corridor )` era verdadero al principio y nada lo borra. Es el estado en el que la heurística más se equivoca en la Figura 3.6, y aquel en el que el robot está más lejos de donde tiene que ir.

Solución 3.5.2. En el estado inicial solo están despejados `d1`, `peg2` y `peg3`. Los hechos del objetivo `(on d1 d2)`, `(on d2 d3)` y así sucesivamente,

hasta el penúltimo, son verdaderos desde el principio, y en la relajación siguen siéndolo. El único hecho del objetivo que falta es el disco más grande sobre `peg3`. Para mover el disco n, este tiene que estar despejado; queda despejado, y sigue así, cuando se aparta el disco $n-1$, lo que exige que el disco $n-1$ esté despejado, y así hasta `d1`. Así que el plan relajado mueve cada disco exactamente una vez, del más pequeño al más grande: n acciones. Nunca hay que volver a poner nada, ya que nunca se quitó nada de verdad. Con `--debug=search`, `hanoi-p04` (6 discos) imprime `initial h: ff 6.000000`; su plan más corto tiene 63 movimientos.

Solución 3.5.3. Con los borrados ignorados, una ficha que se desliza está después en las dos casillas, y la casilla que dejó está vacía *y* sigue conteniendo la ficha. Las casillas vacías no hacen más que acumularse. Así que, tras unos pocos deslizamientos alrededor del hueco, todas las casillas están vacías, y cada ficha puede ir a su casilla objetivo por el camino más corto, como si las demás fichas no estuvieran. Un plan relajado es entonces poco más que la suma de las distancias de las fichas a sus sitios: las fichas nunca se bloquean entre sí. La estimación se queda corta, ya que en el puzle real las fichas gastan la mayoría de los movimientos en quitarse de en medio unas a otras. Para `puzzle-p02`, Kairos imprime `initial h: ff 22.000000, lmcount 13.000000`; el mejor plan tiene 31 deslizamientos.

Sección 3.6 — Un plan, un plan mejor, el mejor plan

Solución 3.6.1. Con `--no-anytime`: un plan de 357 movimientos, y el estado `solved (no improvement)`: se le dijo a la búsqueda que no mejorase. Con `--target-cost 300`: el primer plan (357) no es lo bastante bueno, la búsqueda sigue, y el siguiente plan que encuentra tiene 255 movimientos; el estado es `solved (target reached)`. El objetivo pedía 300 o menos, y el planificador no se detiene *en* 300 sino en el primer plan que es lo bastante bueno. Sin ninguna opción: los mismos dos planes, y después `solved (proven optimal)`: 255 es $2^8 - 1$, el mínimo para 8 discos.

Solución 3.6.2. El problema es `puzzle-p04.pddl`; su `:init` termina con

```
(tile-at t1 c11) (tile-at t3 c12) (tile-at t6 c13)
(tile-at t5 c21) (empty c22) (tile-at t2 c23)
(tile-at t4 c31) (tile-at t7 c32) (tile-at t8 c33))
```

Kairos encuentra un primer plan de 10 deslizamientos y después uno de 8, que demuestra que es el mejor tras encontrar 520 estados: muchos menos que los 181 440 del tablero difícil. Una demostración de optimalidad solo tiene que mirar donde aún podría esconderse un plan más corto que el encontrado, y alrededor de un plan de 8 deslizamientos esa región es pequeña.

Sección 3.7 — No hay plan, o no se ha encontrado

Solución 3.7.1. El fichero es `courier-busy.pddl`. La única acción que añade `(free r1)` es `drop`, que necesita que el robot lleve algo; la única que hace que lleve algo es `pick`, que necesita que esté libre. Ninguno de los dos hechos puede hacerse verdadero primero, ni siquiera con los borrados ignorados, así que la instanciación demuestra que no hay plan y no hace falta buscar. La nota de Kairos, impresa abajo, recorre el círculo una vez (drop, pick, drop, pick) y

se detiene. Con `-v`, la línea `[ground]` informa de 14 acciones: los movimientos son todo lo que queda. Olvidar `(free r1)` en el estado inicial es un clásico de la hipótesis del mundo cerrado: lo que no se dice es falso, así que el robot empieza con las manos llenas de nada.

```
$ kairos courier-domain.pddl solutions/courier-busy.pddl --threads 1
note: the goal needs atoms that no action can reach
(item-at meds1 ward-a) can never be reached:
  drop adds it, but needs (carrying ?r - object meds1), which never
     holds
    pick adds it, but needs (free ?r - object), which never holds
      drop adds it, but needs (carrying ?r - object ?i - object),
     which never holds
        pick adds it, but needs (free ?r - object), which never
     holds [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

Solución 3.7.2. Primero, la línea de estado y el código de salida: tras 30 segundos sin plan, casi con seguridad son `timeout (timeout)` y 11, que no demuestran nada; «imposible» exige `unsolvable` y 10. Segundo, la nota que precede a la línea de estado y la línea `[ground]` de `-v`: un número de acciones instanciadas muy alejado de la estimación apunta a un error en el modelo más que a un problema difícil. Después: dar más tiempo; probar una versión más pequeña del mismo problema (menos artículos, un solo robot) para ver si el modelo resuelve algo en absoluto; y pedir menos (`--no-anytime`), por si el tiempo se fue en mejorar. Si las versiones pequeñas se resuelven al instante y la grande nunca, puede que el modelo sea correcto y esté mal guiado: los callejones sin salida y las simetrías son los sospechosos de la Sección 3.8.

Sección 3.8 — Qué significa esto para el modelador

Solución 3.8.1. Como máximo $2 \times 2 \times 4 \times 8 = 128$ instancias, una vez que los predicados de tipo han hecho su trabajo. La mitad, 64, tienen el mismo robot como `?from` y `?to`: un robot que se da un artículo a sí mismo. Son inútiles, y peor que inútiles si los efectos se escriben sin cuidado (un efecto que añade y borra el mismo hecho). Con el STRIPS del Capítulo 2, nada en el dominio puede decir que dos parámetros son objetos distintos. Hay dos salidas: el requisito `:equality`, en el Capítulo 4, o, sin salir del STRIPS puro, un predicado estático `(different ?a ?b)` enumerado en el problema para cada par de robots distintos, y una precondición sobre él. La segunda es el mismo truco que `connected`: un hecho estático que poda durante la instanciación.

Solución 3.8.2. Las ejecuciones dan las cifras de la tabla: 33, 464, 4670 y 43 595 estados para la demostración con una a cuatro cajas. El problema `courier-p04`, con tres entregas distintas, necesita 5054: más o menos lo mismo que tres cajas idénticas. La conclusión es que el planificador no gana nada con que las cajas sean idénticas. Para una persona, tres cajas iguales son un problema más sencillo que tres recados distintos; para un planificador que nombra cada objeto, son la misma clase de problema, con el agravante de que la mayoría de los estados que distingue son la misma situación. La simetría es un conocimiento sobre el problema que esta codificación no le da al planificador.

Solución 3.8.3. El robot no puede hacer nada en absoluto: ningún movimiento empieza en la UCI. La instanciación conserva solo las acciones que podrían llegar a ser aplicables, así que no conserva ninguna, y el hecho del objetivo es inalcanzable. La ejecución de abajo muestra cero acciones instanciadas y tres átomos: los tres hechos del estado inicial que no son estáticos. Todo el espacio de estados es un único estado. La última línea de la nota es menos precisa que las demás: la recogida nunca es aplicable porque el robot nunca está en la farmacia.

```
$ kairos courier-domain.pddl solutions/courier-trapped.pddl --
     threads 1 -v
...
[   0.001s] [ground] grounded in 0.1 ms: 3 atoms, 0 numeric fluents,
      0 actions, 0 durative actions, 0 axioms (31 Datalog facts, 17
     rules, 0 instances pruned)
[   0.001s] [orchestrator] unsolvable (exhausted) in 0.3 ms
note: the goal needs atoms that no action can reach
(item-at meds1 icu) can never be reached:
  drop adds it, but needs (carrying ?r - object meds1), which never
     holds
    pick adds it, but is never applicable (types, negative or
     numeric conditions) [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
...
```

Sección 3.9 — El método MODEL: tamaño y guía en cifras

Solución 3.9.1. Estimación. Átomos: `robot-at` $2{\times}8 = 16$, `item-at` $4{\times}8 = 32$, `carrying` $2 \times 4 = 8$, `free` 2: 58. Acciones instanciadas: `go` $2 \times 56 = 112$, `pick` $2 \times 4 \times 8 = 64$, `drop` 64: 240. Kairos informa de `58 atoms` y `240 actions`: la estimación se cumple.
La ejecución da planes de 18, 17 y 16 acciones, y la demostración para 16 tras encontrar 509 848 estados, en unos 3 segundos. La codificación A en `courier-p06` tenía 156 acciones, un mejor plan de 27 y 544 039 estados para la demostración, en más o menos el mismo tiempo. El número de estados es casi el mismo, como debe ser: las dos codificaciones describen las mismas situaciones (robots en habitaciones, artículos en habitaciones o en las manos), y solo difieren en las flechas entre ellas. C tiene 84 acciones instanciadas más y planes once acciones más cortos. La demostración cuesta lo mismo.

Solución 3.9.2. El objetivo tiene que decir también que el robot termina con las manos vacías. La solución, `courier-loaded-p02.pddl`, es el problema del capítulo con un hecho más en el objetivo:

```
(:goal (and (item-at meds1 ward-a)
            (item-at sample1 lab)
            (item-at meal1 icu)
            (free r1))))
```

Ahora el plan tiene 19 acciones, como el de la codificación A, y termina con una descarga. La nueva columna B: Tamaño, el mismo (36 átomos, 104 acciones); Guía, un primer plan de 19 acciones tras 35 expansiones y una demostración tras 5054 estados, la misma cifra que A. El veredicto no cambia, empeora para B: con la corrección no le queda ninguna ventaja, y la corrección está en el problema, donde todos los problemas futuros tendrán que acordarse de ella. Con varios robots, `(free r1)` ni siquiera bastaría: todos los robots

tendrían que estar libres. Un modelo en el que el objetivo natural es erróneo es un modelo que hay que evitar.

Solución 3.9.3. **Fidelidad**: D sigue teniendo `move`, así que las rutas están en el plan para los trayectos con las manos vacías; pero `deliver` salta de una habitación a otra con el artículo, así que la ruta *con* el artículo no está, y nunca se ve al robot llevando nada: ningún estado tiene `carrying` verdadero. Correcta solo donde C es correcta, y confusa donde hace falta A. **Tamaño**: `deliver` tiene cuatro parámetros; con R robots, I artículos y 56 pares alcanzables, añade $56 \cdot R \cdot I$ acciones instanciadas (168 para un robot y tres artículos) a las de A. Es la más grande de las cuatro. **Guía**: los planes son más cortos que en A, ya que una entrega es una acción más el trayecto hasta el artículo; pero `pick` y `drop` tienen que seguir en el dominio si alguna vez se van a dejar artículos por el camino, y entonces el planificador tiene dos maneras de hacerlo todo, lo que es una simetría de nuestra propia cosecha. **Claridad**: peor; `deliver` repite lo que ya dicen `pick`, `move` y `drop`, y las dos descripciones pueden acabar divergiendo. **Portabilidad**: la misma, STRIPS puro. D mezcla dos niveles de detalle en un mismo modelo. O la ruta importa, y entonces A; o no importa, y entonces C.

C.4. Soluciones del Capítulo 4

Los ficheros que se nombran en estas soluciones se encuentran en la carpeta `examples/ch04/solutions/`.

Sección 4.1 — Lo que cuestan los predicados de tipo

Solución 4.1.1. `kairos check` responde `valid, plannable`: sin tipos, `robot-at` acepta dos objetos cualesquiera, y `(robot-at dock r1)` es un hecho legal. Dice que el «robot» `dock` está en la «habitación» `r1`. Después, el planificador demuestra que no hay plan (`courier-untyped-p01-robot.pddl`):

```
$ kairos ../../ch02/courier-domain.pddl courier-untyped-p01-
    robot.pddl
note: the goal needs atoms that no action can reach
(item-at meds1 ward-a) can never be reached:
  drop adds it, but needs (robot-at ?r - object ward-a), which
     never holds
    move adds it, but is never applicable (types, negative or
     numeric conditions) [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

La nota recorre la cadena hacia atrás desde el objetivo. Para que el artículo esté en la sala A, alguien tiene que dejarlo allí, lo que exige un robot en la sala A; solo `move` pone un robot en una habitación, y `move` nunca es aplicable. ¿Por qué? Sus precondiciones piden `(robot ?r)` y `(robot-at ?r ?from)` a la vez. El único robot es `r1`, y ningún hecho `robot-at` tiene `r1` como primer argumento. La nota lleva hasta `robot-at`, y a partir de ahí el hecho intercambiado se encuentra leyendo; un modelo con tipos lo señala directamente.

Solución 4.1.2. Sin tipos: 20 objetos, tres acciones de tres parámetros cada una, así que $3 \times 20^3 = 24\,000$ candidatas. Aplicables en algún estado alcanzable: un robot puede moverse por cada conexión en cada sentido, $3 \times 30 = 90$ movimientos; cualquier robot puede coger o dejar cualquier artículo en cualquier habitación (todas

las habitaciones son alcanzables, así que cualquier artículo puede acabar en cualquier sitio), $3 \times 5 \times 12 = 180$ acciones de coger y 180 de dejar. Total, 450. Con tipos: `move` tiene $3 \times 12 \times 12 = 432$ candidatas, y `pick` y `drop`, 180 cada una: 792 en total, de las que son aplicables las mismas 450. Los tipos eliminan 23 208 candidatas sin mirar ningún hecho; el predicado estático `connected` elimina las 342 restantes.

Sección 4.2 — Tipos

Solución 4.2.1. El dominio es `gripper-domain.pddl`, con tres tipos:

```
(:types room ball gripper)

(:predicates
  (at-robby ?r - room)
  (at ?b - ball ?r - room)
  (empty ?g - gripper)
  (holding ?g - gripper ?b - ball))

(:action move
  :parameters (?from ?to - room)
  :precondition (at-robby ?from)
  :effect (and (not (at-robby ?from)) (at-robby ?to)))

(:action pick
  :parameters (?b - ball ?r - room ?g - gripper)
  :precondition (and (at ?b ?r) (at-robby ?r) (empty ?g))
  :effect (and (not (at ?b ?r)) (not (empty ?g))
               (holding ?g ?b)))
```

(`drop` es simétrica de `pick`.) Hay un solo robot, así que no necesita tipo ni parámetro: su posición es el predicado `at-robby`, de un argumento. Las pinzas, en cambio, sí son objetos, porque `pick` tiene que elegir una. `gripper-p01.pddl` se resuelve con 9 acciones: un viaje con una bola y otro con dos, o al revés. Con 3 bolas, 2 habitaciones y 2 pinzas, `pick` tiene $3 \times 2 \times 2 = 12$ candidatas, todas ellas aplicables en algún estado. Sin tipos, el problema tiene 7 objetos y `pick`, $7^3 = 343$ candidatas. Kairos instancia el problema con tipos en 28 acciones: 12 acciones de coger, 12 de dejar y 4 movimientos. Cuatro movimientos con dos habitaciones significa que `(move rooma rooma)` está entre ellos: nada separa `?from` de `?to`, un caso para la desigualdad de la Sección 4.6.

Solución 4.2.2. `icu` va después del último tipo de la lista, así que no tiene tipo: es un simple `object`. Un `object` no encaja en un parámetro de tipo `location`, así que se rechaza todo hecho que use `icu` como habitación. En el mapa habitual son los dos hechos de la conexión con la sala B (`courier-p01-notype.pddl`):

```
$ kairos check ../courier-domain.pddl courier-p01-notype.pddl
courier-p01-notype.pddl:17:23: error: in the initial state:
     argument 2 of 'connected' must be location, 'icu' is
     object [semantic.type_mismatch]
 17 |     (connected ward-b icu) (connected icu ward-b)
    |                       ^
courier-p01-notype.pddl:17:39: error: in the initial state:
     argument 1 of 'connected' must be location, 'icu' is
     object [semantic.type_mismatch]
 17 |     (connected ward-b icu) (connected icu ward-b)
    |                                       ^
invalid
```

El mensaje dice «`'icu' is object`»: esa palabra es la pista de que

lo que está mal es la declaración, no el hecho.

Solución 4.2.3. Es legal. `?a` y `?b` son robots: el tipo se aplica a todas las variables sin tipo que lo preceden. `?c` y `?d` van después del último tipo y no tienen ninguno, así que son de tipo `object` y lo admiten todo. La reescritura da a cada una de las dos primeras su propio tipo: `(?a - robot ?b - item ?c ?d)`.

Sección 4.3 — Jerarquías de tipos

Solución 4.3.1. (a) Legal: `porter1` es un celador, y por tanto un portador; `meds1` es un medicamento, y por tanto un artículo. No es aplicable en el estado inicial (el celador está en la sala B y los medicamentos en la farmacia), pero eso es cuestión de hechos, no de tipos. (b) No es legal: el segundo parámetro de `serve` es un `meal` y `meds1` es un `medicine`; son hermanos bajo `item`, y ninguno está por encima del otro. (c) No es legal: los argumentos están intercambiados, un artículo donde se espera un portador y un portador donde se espera un artículo. (d) Legal: la base de carga es un lugar como cualquier otro. Los tipos no pueden decir que nadie come en la base de carga; si eso importara, haría falta un predicado. (e) No es legal: una comida no es un portador.

Solución 4.3.2. La jerarquía tiene dos árboles. Personal: `keeper` y `vet` son subtipos de `staff` (los dos caminan), y `senior` es subtipo de `keeper`, que es exactamente «un cuidador sénior puede hacer todo lo que hace un cuidador». Animales: `bigcat` y `herbivore` bajo `animal`, y las especies bajo ellos.

```
(:types
  enclosure
  keeper vet - staff
  senior - keeper
  bigcat herbivore - animal
  lion tiger - bigcat
  zebra giraffe - herbivore)
(:action feed
  :parameters (?k - keeper ?a - herbivore ?e - enclosure)
  :precondition (and (staff-at ?k ?e) (lives-in ?a ?e))
  :effect (fed ?a))

(:action feed-bigcat
  :parameters (?k - senior ?a - bigcat ?e - enclosure)
  :precondition (and (staff-at ?k ?e) (lives-in ?a ?e))
  :effect (fed ?a))

(:action treat
  :parameters (?v - vet ?a - animal ?e - enclosure)
  :precondition (and (staff-at ?v ?e) (lives-in ?a ?e))
  :effect (treated ?a)))
```

Cada parámetro se sitúa en el nivel de su regla: `feed` admite cualquier `keeper` (sénior incluidos) y un `herbivore`; `feed-bigcat` admite un `senior` y un `bigcat`; `treat` admite un `vet` y cualquier `animal`. Las especies (`lion`, `zebra`...) pasan la prueba de tipo solo a medias: ninguna acción distingue un león de un tigre. Son inofensivas y hacen legibles los problemas, pero bastaría con `leo - bigcat`. Kairos resuelve `zoo-p01.pddl` con 7 acciones, en las que el cuidador sénior alimenta a todos (es el único que puede alimentar al león y, ya puesto a caminar, sale más barato que alimente también al resto) y el veterinario trata a la jirafa. El plan con el cuidador normal se rechaza:

```
$ kairos validate zoo-domain.pddl zoo-p01.pddl zoo-p01-junior.
    plan
zoo-p01-junior.plan:3: error: step 2 (feed-bigcat kim leo cats-
    house) at 1.000: argument 1 must be senior, 'kim' is
    keeper [validate.type_mismatch]
 3 | (feed-bigcat kim leo cats-house)
plan invalid
```

Solución 4.3.3. Con `?from ?to - place` en `fly`, el avión aterriza en cualquier sitio. El plan de `logistics-p01` se reduce de 12 pasos a 4: el avión vuela al almacén de Madrid, carga el paquete, vuela al almacén de París y lo descarga. Sale perjudicada la fidelidad: el modelo tiene ahora planes que no se pueden llevar a cabo. Con `?t - vehicle` en `drive`, los aviones circulan por carretera: el planificador carga el paquete en el avión en el almacén, «conduce» el avión hasta el aeropuerto, vuela y lo conduce hasta el otro almacén. Otra vez una pérdida de fidelidad, de las que solo se descubren leyendo el plan, porque cada paso parece razonable por su nombre. Dar a un parámetro un tipo demasiado alto es la versión con tipos de una precondición que falta.

Sección 4.4 — Un parámetro de varios tipos: `either`

Solución 4.4.1. Con `either` (`courier-at-domain.pddl`), el predicado se declara así

```
(at ?x - (either carrier item) ?l - location)
```

y con un supertipo (`courier-at-hier-domain.pddl`), los tipos ganan una línea y el predicado lo nombra:

```
(:types
  location
  carrier item - locatable
  robot porter - carrier
  medicine sample meal - item)
...
(at ?x - locatable ?l - location)
```

Las acciones son las mismas en los dos ficheros. Ambos resuelven el problema `courier-at-p01.pddl` con un plan de 13 pasos. Para el compañero, la versión con `locatable`: todos los planificadores que leen tipos la leen, mientras que algunos rechazan `either`. Fíjate en lo que cuesta un único `at` en cualquiera de las dos versiones: `(at meds1 lobby)` y `(at r1 lobby)` se parecen ahora, y una acción que solo debería aplicarse a portadores queda protegida por el tipo de su parámetro y por nada más.

Solución 4.4.2. Un tipo tiene un solo supertipo. «Se puede tomar prestado» agrupa libros, DVD y audiolibros; «necesita un reproductor» agrupa DVD y audiolibros. El segundo grupo está dentro del primero, así que aquí todavía es posible un árbol (`playable` bajo `loanable`), pero el ejercicio pide el caso general, y este aparece en cuanto algo que no es prestable necesita también un reproductor. La codificación de `library-player-domain.pddl` conserva la jerarquía para la agrupación principal y usa `either` para la que la atraviesa:

```
  (:action borrow
    :parameters (?m - member ?x - book)
    :precondition (and (on-shelf ?x) (can-borrow ?m))
    :effect (and (not (on-shelf ?x)) (not (can-borrow ?m))
```

```
                (borrowed ?x ?m)))

  (:action borrow-to-play
    :parameters (?m - member ?x - (either dvd audiobook))
    :precondition (and (on-shelf ?x) (can-borrow ?m)
                       (has-player ?m))
    :effect (and (not (on-shelf ?x)) (not (can-borrow ?m))
                 (borrowed ?x ?m)))
```

Hacen falta dos acciones porque, con lo que sabemos hasta ahora, una precondición no puede decir «si necesita un reproductor, entonces el socio tiene uno»; el Capítulo 5 da los medios. En `library-player-p01.pddl` Ana quiere un audiolibro, y el plan consiste en comprar un reproductor y después llevarse el audiolibro en préstamo.

Sección 4.5 — Constantes

Solución 4.5.1. La primera versión (`courier-park-cheat-domain.pddl`) declara `(:constants dock - location)` y

```
(:action park
  :parameters (?r - robot)
  :precondition (and (robot-at ?r dock) (free ?r))
  :effect (parked ?r))
```

El problema `courier-park-p01.pddl` no incluye `dock` entre sus objetos. El plan:

```
$ kairos courier-park-cheat-domain.pddl courier-park-p01.pddl
; cost = 8
(park r1)
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

El robot aparca primero y después hace su trabajo. El objetivo `(parked r1)` se cumple al final, porque nada lo borra nunca, pero el robot está en la sala A. El modelo ha perdido el significado de «aparcado»: un robot aparcado no se mueve. La reparación natural es una precondición «no aparcado» en `move`, que necesita el Capítulo 5. Con STRIPS a secas, la reparación consiste en que `park` borre lo que `move` necesita (`courier-park-domain.pddl`):

```
  :effect (and (not (robot-at ?r dock)) (parked ?r))
```

Un robot aparcado ya no está `robot-at` en ninguna habitación, así que no se le aplica ningún `move`, `pick` ni `drop`. Ahora el plan tiene 11 pasos y termina con la vuelta a la base de carga y `(park r1)`, y `kairos validate` rechaza el plan tramposo (`courier-park-p01-early.plan`) en su segundo paso. El precio se paga en Claridad: el lector tiene que saber que «aparcado» implica «en la base de carga», porque el estado ya no dice dónde está el robot.

Solución 4.5.2. (a) Si la mesa es un objeto, sí: todos los problemas de bloques tienen exactamente una, y las acciones que ponen un bloque sobre la mesa la nombran. La codificación clásica evita el objeto con un predicado `(ontable ?b)`, que es la misma decisión

llevada un paso más allá. (b) No. Ninguna regla de logistics trata de Madrid; forma parte de una situación. (c) No, aunque estén en todos los problemas: `pick` y `drop` tratan igual a las dos pinzas y nunca nombran una. Podrían declararse como constantes para ahorrar dos palabras en cada problema, y no se rompería nada, pero la prueba (¿la nombra una regla?) dice que son objetos de la situación: un robot con tres pinzas es otro problema, no otro dominio. (d) Sí, si todos los hospitales que interesan tienen un laboratorio y `analyze` solo ocurre allí: `(item-at ?s lab)` en su precondición. Si algunos hospitales tienen dos laboratorios, un predicado estático y un parámetro, como en la Tabla 4.3.

Sección 4.6 — Igualdad

Solución 4.6.1. La acción, en `courier-swap-domain.pddl`:

```
(:action swap
  :parameters (?a ?b - robot ?i ?j - item ?l - location)
  :precondition (and (not (= ?a ?b))
                     (robot-at ?a ?l) (robot-at ?b ?l)
                     (carrying ?a ?i) (carrying ?b ?j))
  :effect (and (not (carrying ?a ?i)) (not (carrying ?b ?j))
               (carrying ?a ?j) (carrying ?b ?i))))
```

Sin la primera precondición (`courier-swap-noeq-domain.pddl`), `(swap r1 r1 meds1 meds1 lab)` es aplicable: todas las precondiciones tratan de que `r1` lleve `meds1` en el laboratorio, dos veces. Sus efectos borran `(carrying r1 meds1)` y añaden ese mismo hecho. Cuando una acción borra y añade un mismo hecho, el hecho es cierto después, así que el paso no cambia nada; `kairos validate` acepta `courier-swap-p01-self.plan`, que empieza con él, en el dominio sin la desigualdad, y lo rechaza en el otro. Los autointercambios con dos artículos *distintos* necesitarían un robot que llevara dos artículos, cosa que nunca ocurre. Así que aquí la desigualdad que falta no crea planes imposibles; crea acciones instanciadas inútiles. En `courier-swap-p01.pddl`, Kairos instancia 220 acciones sin la desigualdad y 156 con ella: las 64 que desaparecen son los autointercambios, 2 robots por 2 artículos por 2 artículos por 8 habitaciones. Los dos dominios encuentran el mismo plan óptimo de 9 pasos, en el que `r2` va hasta el laboratorio, los robots intercambian sus artículos y `r2` vuelve con los medicamentos.

Solución 4.6.2. El fichero es `logistics-eq-domain.pddl`. Desaparecen seis acciones y quedan 18: cada uno de los dos camiones podía «conducir» desde cada uno de los dos lugares de su ciudad hasta ese mismo lugar (4 acciones), y el avión podía «volar» de cada aeropuerto a sí mismo (2 acciones). Ninguna de ellas cambia nada, así que el plan es el mismo, de 12 pasos.

Solución 4.6.3. No cambian ni los planes ni las acciones instanciadas: en `courier-p02`, Kairos instancia 124 acciones con ella o sin ella. Un movimiento de una habitación a sí misma necesita `(connected l l)`, que ningún problema declara, así que esas instancias ya se descartaban. Importaría si el mapa tuviera bucles (un hecho `(connected lobby lobby)` escrito por error o por un generador de mapas), o en un mundo sin mapa, donde cualquier habitación es alcanzable desde cualquier otra y `move` no tiene precondición `connected`: entonces, como en el dominio gripper del Ejercicio 4.2.1, solo la desigualdad separa las dos habitaciones.

Sección 4.7 — El método MODEL: clases de cosas

Solución 4.7.1. Las clases, con la prueba «algún predicado o alguna acción las trata de forma distinta»:

Clase	Es un	Por qué
`place`		en él están vehículos y palés
`shelf`, `station`	`place`	un palé solo se empaqueta en un puesto
`mover`		se mueve y lleva un palé
`forklift`, `cart`	`mover`	solo una carretilla elevadora llega a una estantería alta
`pallet`		se transporta

«Alta» y «frío» son propiedades de los lugares, no tipos: `(high ?s - shelf)` y `(cold ?s - station)` como predicados estáticos. Podrían ser subtipos (`high-shelf`), pero dos propiedades independientes de los lugares pronto pedirían dos supertipos. «Congelado» es una propiedad estática de los palés, `(frozen ?p - pallet)`; un subtipo `frozen-pallet` también es defendible, ya que en esta historia un palé nunca se descongela. La regla «productos congelados solo en un puesto frío» necesita, con lo que tenemos, dos acciones de descarga (una para los palés congelados con `(cold ?s)` en su precondición y otra para los demás, distinguidas por el tipo) o la implicación del Capítulo 5. La tabla de estado: dónde está cada vehículo, dónde está cada palé o quién lo lleva, si un vehículo está libre; estáticos: los pasillos, `high`, `cold`, `frozen`. Ninguna constante: ninguna regla nombra una estantería o un puesto concretos.

Solución 4.7.2. *Fidelidad*: los dos dejan a `r2` fuera de todo plan, si ninguna acción admite `broken-robot`; pero entonces el tipo tiene que quedar fuera de `robot` en la jerarquía, así que `r2` no puede aparecer siquiera en `(robot-at r2 dock)`, y el modelo no puede decir dónde está el robot averiado (puede estar bloqueando una habitación). Y un robot que se avería o se repara *durante* el plan no se puede modelar en absoluto: los tipos no cambian. *Tamaño*: la versión con el tipo no instancia nada para `r2`; la versión con el predicado instancia sus acciones y las poda, porque `(working r2)` es estático y falso. Tras la instanciación son iguales. *Guía*: igual. *Claridad*: el predicado dice lo que significa, en el estado inicial, donde el lector busca la situación; el tipo esconde un hecho de hoy en la declaración de los objetos. *Portabilidad*: los dos son STRIPS con tipos, sin más. Veredicto: el predicado. «Averiado» es estado, aunque en este problema dé la casualidad de que no cambia.

C.5. Soluciones del Capítulo 5

Salvo los que ya se usaron en el capítulo, los ficheros que se citan en estas soluciones están en `examples/ch05/solutions/`.

Sección 5.1 — Condiciones sobre la ausencia: precondiciones negativas

Solución 5.1.1. Ningún hecho: cerrada. Solo `(open d)`: abierta. Solo `(locked d)`: cerrada con llave. Los dos: una puerta abierta con el cerrojo echado, que no significa nada en este modelo. No puede

alcanzarse. Solo `open-door` añade `open`, y exige `(not (locked ?d))`; ninguna acción añade `locked`. Así que, si una puerta no tiene los dos hechos en el estado inicial, no los tendrá nunca. El argumento es un *invariante*: una propiedad que se cumple al principio y que toda acción conserva. El dominio `courier-doors-v0` del capítulo lo rompía, al dejar que `open-door` se aplicara a una puerta cerrada con llave.

Solución 5.1.2. `courier-doors-hands-domain.pddl` añade `(free ?r)` a las precondiciones de `open-door` y `close-door`. Abrir no da problemas: el robot abre la puerta de la farmacia al entrar, con la pinza vacía. Cerrar, sí. Al salir lleva los medicamentos. El plan de Kairos tiene 11 pasos:

```
$ kairos courier-doors-hands-domain.pddl ../courier-doors-p02.
    pddl
; cost = 11
...
(move-through r1 door-pharmacy pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lobby)
(close-door r1 door-pharmacy lobby pharmacy)
```

El robot entrega primero y vuelve a cerrar la puerta con la pinza libre. El plan que predice la mayoría de la gente (dejar los medicamentos en el vestíbulo, cerrar la puerta y volver a cogerlos) también tiene 11 pasos: tres acciones de más en los dos casos. Al planificador le da igual; al farmacéutico, cuya puerta se queda abierta un rato, quizá no. Expresarlo exigiría una restricción sobre el plan completo (Capítulo 15).

Solución 5.1.3. En `courier-doors-lock-domain.pddl`:

```
(:action lock
  :parameters (?r - robot ?d - door ?a ?b - location)
  :precondition (and (robot-at ?r ?a) (joins ?d ?a ?b)
                     (has-keycard ?r)
                     (not (open ?d)) (not (locked ?d)))
  :effect (locked ?d))
```

`(not (open ?d))` es imprescindible: sin ella el robot podría cerrar con llave una puerta abierta, y sería alcanzable la cuarta combinación del Ejercicio 5.1.1, la que no tiene sentido, porque ahora hay una acción que añade `locked`. `(not (locked ?d))` solo elimina una acción inútil (cerrar con llave lo que ya lo está). En `courier-doors-lock-p01.pddl` el plan tiene 10 pasos: tras salir de la farmacia con los medicamentos, el robot hace `close-door` y después `lock`.

Sección 5.2 — Objetivos negativos

Solución 5.2.1. El objetivo de `courier-doors-p05.pddl` es `(and (item-at meds1 ward-a) (not (robot-at r1 ward-a)))`. El plan gana un paso, `(move r1 ward-a corridor)`, y el robot acaba en el pasillo, estorbando a todo el mundo. Lo que la supervisora quería decir seguramente era «vuelve a la base de carga». Un objetivo negativo dice dónde no debe estar el robot y deja que el planificador elija el más barato de todos los demás sitios. Cuando hay un sitio en el que el robot *debe* estar, un objetivo positivo lo dice mejor.

Solución 5.2.2. `courier-doors-p06-init.pddl`:

```
$ kairos check ../courier-doors-domain.pddl courier-doors-p06-
    init.pddl
courier-doors-p06-init.pddl:21:5: warning: negative initial
    facts have no effect [semantic.negative_init]
 21 |     (not (open door-pharmacy))
    |     ^
valid, plannable
...
```

El plan es el de `p01`. La puerta ya estaba cerrada antes de escribir esa línea, porque ningún hecho decía que estuviera abierta.

Sección 5.3 — Alternativas en una condición: `or` e `imply`

Solución 5.3.1. `library-imply-domain.pddl`:

```
(:action borrow
  :parameters (?m - member ?x - loanable)
  :precondition (and (on-shelf ?x) (can-borrow ?m)
                     (imply (needs-player ?x)
                            (has-player ?m)))
  :effect (and (not (on-shelf ?x)) (not (can-borrow ?m))
               (borrowed ?x ?m)))
```

Un libro no necesita reproductor, así que para un libro la implicación es cierta tenga lo que tenga el socio. En `library-imply-p01.pddl` el plan es `(buy-player ana)`, `(borrow bob book1)`, `(borrow ana audio1)`. Frente a la Solución 4.4.2, las dos acciones se han convertido en una, y «necesita un reproductor» ha pasado a ser un hecho de cada problema y ha dejado de ser una unión de tipos. Una nueva clase de cosa prestable ya no obliga a tocar el dominio.

Solución 5.3.2. Con las cuatro combinaciones de `a` y `b`:

a	b	(imply a b)	(or (not a) b)	(not (and a (not b)))
cierto	cierto	cierto	cierto	cierto
cierto	falso	falso	falso	falso
falso	cierto	cierto	cierto	cierto
falso	falso	cierto	cierto	cierto

Las tres son falsas solo cuando `a` se cumple y `b` no. La tercera forma de la regla está en `courier-access-nand-domain.pddl`: «no es cierto que la sala esté restringida y el robot no tenga tarjeta».

```
                    (not (and (restricted ?to)
                              (not (has-keycard ?r)))))
```

Resuelve `courier-access-p01` con el mismo plan y las mismas 64 acciones instanciadas: el planificador empuja la negación hacia dentro y llega al `or`. Para un lector, el `imply` es la más clara de las tres.

Sección 5.4 — Algún objeto: `exists`

Solución 5.4.1. «Lleva una comida» habla de alguna comida, sea cual sea, y la regla lo prohíbe: es un `exists` negado. En el fichero `courier-nomeal-domain.pddl` las dos exigencias sobre una sala restringida quedan bajo un mismo `imply`:

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition
    (and (robot-at ?r ?from) (connected ?from ?to)
```

```
          (imply (restricted ?to)
                 (and (has-keycard ?r)
                      (not (exists (?m - meal)
                                   (carrying ?r ?m))))))
   :effect (and (not (robot-at ?r ?from))
                (robot-at ?r ?to)))
```

El tipo de la variable cuantificada hace la selección: a un robot que lleva un medicamento no le afecta. `courier-nomeal-p01.pddl` (medicamentos a la UCI, una comida a la sala B) tiene un plan de 13 pasos. `courier-nomeal-p02.pddl` pide la comida en la UCI y no tiene ninguno, aunque el robot tiene tarjeta: Kairos recorre los 480 estados alcanzables y responde «unsolvable».

Solución 5.4.2. Solo cambia la sala de la escolta. El fichero es `courier-escort-inside-domain.pddl`:

```
              (exists (?e - robot)          ; inside
                (and (has-keycard ?e) (robot-at ?e ?to)))))
```

El plan antiguo ya no funciona: `r2` esperaba en la sala B, y ahora tiene que estar en la UCI. El plan nuevo tiene 9 pasos, uno más, y en él `r2` sigue hasta la UCI antes de que llegue `r1`. No es más corto que dejar que `r2` lleve él mismo los medicamentos, así que con esta regla la escolta solo compensa cuando el robot con tarjeta está ocupado o ya está dentro.

Solución 5.4.3. La llave de hierro se rompe, y el cofre de pino da la llave de plata, que solo encaja en el cofre de pino, que ya está abierto. No se puede hacer nada más: es un callejón sin salida, y el tesoro se pierde. El planificador tiene que probar primero el cofre de roble, o descubrir el callejón sin salida y volver atrás. Con `exists` y llaves que no se gastan, abrir un cofre nunca quita nada, así que ningún orden de las acciones puede estropear el rompecabezas. Una variable tiene que ser un parámetro cuando un efecto debe mencionar el objeto (aquí, para borrar `(have ?k)`) o cuando el plan debe nombrarlo.

Sección 5.5 — Todos los objetos: `forall`

Solución 5.5.1. En `courier-night-domain.pddl`:

```
(:action close-floor
  :parameters ()
  :precondition
    (and (forall (?r - robot) (robot-at ?r dock))
         (forall (?d - door) (not (open ?d))))
  :effect (floor-closed))
```

y `move` gana la precondición `(not (floor-closed))`. Esto último es lo que pide «después de ella, ningún robot se mueve», y es importante: sin ello, el planificador cierra la planta en cuanto los dos robots coinciden en la base de carga y luego sigue repartiendo, la misma trampa que la del robot aparcado de la Solución 4.5.1. Ahora que hay precondiciones negativas, el arreglo cabe en una línea. En `courier-night-p01.pddl` Kairos encuentra un plan de 15 pasos en el que `r2`, que empieza en el laboratorio, cierra la puerta del laboratorio al salir, hace la entrega, cierra la puerta de la farmacia y vuelve a la base de carga, donde `r1` ha estado todo el tiempo; `(close-floor)` es el último paso.

Solución 5.5.2. El `forall` es cierto para un tipo vacío, así que el objetivo se cumple en el estado inicial y el plan es vacío. El `exists` es falso para un tipo vacío y ninguna acción crea objetos, así que no hay

plan (`courier-ward-p05-none.pddl`):

```
$ kairos ../courier-ward-domain.pddl courier-ward-p05-none.
    pddl
...
  kairos-aux-1 adds it, but there is no object of type
    medicine [orchestrator.unreachable_goal]
kairos: unsolvable (exhausted)
```

(`kairos-aux-1` es el nombre que Kairos da internamente al objetivo cuantificado.)

Solución 5.5.3. El objetivo de `blocks-p02.pddl` es `(forall (?x - block) (on-table ?x))`, y el plan desmonta la torre desde arriba: `(unstack a b)`, `(put-down a)`, `(unstack b c)`, `(put-down b)`. El objetivo obliga también a que la mano acabe vacía, porque un bloque sostenido no está sobre la mesa. «`?x` está despejado» con un cuantificador universal: `(forall (?z - block) (not (on ?z ?x)))`.

Solución 5.5.4. La negación de «todo `d` cumple `c`» es «algún `d` no cumple `c`», y una implicación solo falla cuando su primera parte se cumple y la segunda no:

```
(exists (?d - door)
  (and (joins ?d ?from ?to) (not (open ?d))))
```

Con palabras: alguna puerta de este paso no está abierta. Fíjate en el `and`: la regla práctica del capítulo (`imply` bajo `forall`, `and` bajo `exists`) es precisamente esta equivalencia.

Sección 5.6 — Qué hace un planificador con las condiciones ricas

Solución 5.6.1. Una acción por cada manera de elegir una parte de cada `or`: $2 \times 2 \times 2 = 8$. Para la escolta con los robots `r1` y `r2`, el `exists` se convierte en un `or` sobre los dos robots, y las alternativas son cuatro: (1) la sala no está restringida; (2) `?r` tiene tarjeta; (3) `r1` tiene tarjeta y está en `?from`; (4) `r2` tiene tarjeta y está en `?from`. Cada `move` instanciado puede aparecer en hasta cuatro copias. Con tarjeta solo para `r2`, los hechos estáticos eliminan la alternativa (3) en todas partes y la alternativa (2) para `r1`, y así es como la cuenta se queda en 98.

Sección 5.7 — El método MODEL: ausencia, alguno, todos

Solución 5.7.1. Medido con `blocks-p01` y `blocks-clear-p01`:

Criterio	Sin `clear`	Con `clear`
Fidelidad	El mismo plan de 6 pasos. Necesita la desigualdad en `stack`.	El mismo plan. Si falta un hecho `clear` en el estado inicial, un bloque queda inservible.
Tamaño	21 hechos, 32 acciones instanciadas.	29 hechos, 40 acciones instanciadas.
Guía	93 estados visitados.	93 estados visitados.
Claridad	Efectos cortos; el estado son solo las torres. La precondición cuesta más de leer.	Precondiciones sencillas; cuatro efectos por acción para mantener `clear` al día.
Portabilidad	Negación, igualdad y un cuantificador.	STRIPS con tipos, sin más.

Para un curso, la versión sin `clear`, una vez vistos los cuantificadores: dice qué significa estar despejado. Para un planificador temporal como POPF u OPTIC, que no admiten precondiciones negativas, la clásica es la única opción.

Solución 5.7.2. *Paso O.* Clases: `cup` y `drink` (las recetas). La máquina es única, así que no es un objeto; sus propiedades son predicados sin argumentos.

Propiedad	De qué	¿Cambia?
`under-nozzle`	una taza	sí
`full`, `holds`	una taza (y una bebida)	sí
`with-milk`	una bebida	estática
`has-water`, `has-beans`, `has-milk`, `clean`	la máquina	sí

Paso D. El dominio es `coffee-domain.pddl`; tres de sus reglas tienen una condición rica. «Solo cabe una taza» trata de la ausencia de cualquier taza, y `place` exige `(not (exists (?o - cup) (under-nozzle ?o)))`. «Un capuchino necesita además leche» es una implicación sobre una propiedad estática de la receta:

```
(:action make
  :parameters (?d - drink ?c - cup)
  :precondition (and (under-nozzle ?c) (not (full ?c))
                     (has-water) (has-beans) (clean)
                     (imply (with-milk ?d) (has-milk)))
  :effect (and (full ?c) (holds ?c ?d) (not (clean))))
```

«Solo se puede limpiar cuando no hay ninguna taza bajo la boquilla» vuelve a ser un `exists` negado, en `clean-machine`. El plan para `coffee-p01.pddl` tiene 9 pasos: `(add-milk)`; después, para cada taza, `place`, `make`, `remove` y `(clean-machine)`. La taza tiene que retirarse antes de limpiar, y la máquina tiene que limpiarse antes de la segunda bebida: los dos órdenes salen de las condiciones negativas. Un predicado complementario `(nozzle-free)` evitaría el cuantificador, a cambio de dos efectos más que mantener.

C.6. Soluciones del Capítulo 6

En `examples/ch06/solutions` están los ficheros PDDL de estas soluciones.

Sección 6.1 — Efectos que dependen de la situación

Solución 6.1.1. Cada acción de empujar tiene los parámetros de un movimiento (un robot, un carro, 14 pares ordenados de habitaciones conectadas) y k parámetros de artículo, así que tiene $14 \cdot 5^k$ candidatas: $14 + 70 + 350 + 1750 = 2184$ de `push-cart-0` a `push-cart-3`. Un `push-cart-4` añadiría $14 \cdot 5^4 = 8750$, cuatro veces todas las demás juntas.

Para `courier-cart-cases-p02` (3 artículos) la misma cuenta da $14\,(1+3+9+27) = 560$ candidatas, y Kairos informa de 343 acciones instanciadas para el dominio *entero*, de las que 224 son empujones. La instanciación descarta las candidatas cuya precondición no se puede cumplir nunca. Aquí son las que repiten un artículo, por `(not (= ?i ?j))`: las versiones de dos y de tres artículos conservan $3 \cdot 2 = 6$ elecciones ordenadas cada una, y $14\,(1 + 3 + 6 + 6) = 224$. Quedan los seis órdenes de los mismos tres artículos: seis acciones instanciadas que hacen lo mismo.

Solución 6.1.2. (a) Basta una lista fija: la acción nombra la válvula y la tubería. (b) Necesita un cuantificador: «todas las tuberías» es un efecto universal. (c) Necesita las dos cosas: toda persona, si está dentro del ascensor, cambia de piso. (d) Necesita una condición en el efecto: el botón siempre se puede pulsar, y que se encienda depende de dónde esté el ascensor. (Si el botón no se pudiera pulsar con el ascensor en el piso, sería una precondición.) (e) Basta una lista fija, siempre que el artículo sea un parámetro de la acción, como en `drop`.

Sección 6.2 — Efectos condicionales: `when`

Solución 6.2.1. Todas las condiciones se leen en el estado anterior a cada aplicación.

1. En $\{p\}$ solo se cumple la primera condición: se añade q. Resultado $\{p, q\}$.
2. En $\{p, q\}$ se cumplen las dos primeras: se añaden q (que ya estaba) y r. Resultado $\{p, q, r\}$.
3. En $\{p, q, r\}$ se cumplen las tres: se vuelven a añadir q y r y se borra p. Resultado $\{q, r\}$.
4. En $\{q, r\}$ se cumplen las dos últimas; p se borra otra vez, lo que no cambia nada. Resultado $\{q, r\}$, y así para siempre.

Ejecutada en orden, como un programa, la primera aplicación añadiría q, luego encontraría q y añadiría r, y luego encontraría r y borraría p: llegaría a $\{q, r\}$ en un paso en lugar de en tres.

Solución 6.2.2. Los tres niveles se codifican con dos predicados; apagada es la ausencia de los dos. Cada caso del botón es un `when`, y las tres condiciones se excluyen entre sí:

```
(:action press
  :parameters (?l - lamp)
  :effect (and
    (when (and (not (low ?l)) (not (high ?l))) (low ?l))
    (when (low ?l) (and (not (low ?l)) (high ?l)))
    (when (high ?l) (not (high ?l))))))
```

Para el problema `dimmer-p01.pddl` el plan tiene cuatro pasos, dos pulsaciones por lámpara: de baja a alta y a apagada para `l1`, de apagada a baja y a alta para `l2`. No se puede saltar un nivel, porque el botón no tiene otra acción. Fíjate en el segundo `when`: borra `low` y añade `high` en el mismo efecto. Si hubiéramos olvidado el borrado,

una lámpara estaría baja y alta a la vez, un estado que no significa nada.

Solución 6.2.3. Ningún plan cambia, y el número de acciones instanciadas se queda en 31 para los tres problemas con muestra: `(sealr1 sample1)` existe en las dos versiones, ya que la instanciación no puede saber de antemano que una muestra nunca se sellará dos veces. Lo que cambia es dónde es aplicable la acción. Sin la precondición, sellar una muestra sellada está permitido y no hace nada: cada estado con una muestra sellada y transportada gana una acción que lo deja tal como estaba.
A favor de conservar la precondición: describe el mundo (una bolsa sellada no se vuelve a sellar) y elimina elecciones inútiles de la búsqueda. A favor de quitarla: la acción se vuelve más sencilla, y añadir `(sealed ?i)` dos veces es inofensivo. El primer argumento es el más fuerte. Una precondición que prohíbe una acción sin efecto no cuesta nada y le ahorra al planificador tenerla en cuenta.

Sección 6.3 — Efectos universales: `forall`

Solución 6.3.1. La alarma cambia todas las puertas, sean cuales sean las puertas del problema:

```
27  ; One effect for all the doors of the problem, however many.
28  (:action raise-alarm
29    :parameters (?r - robot ?l - location)
30    :precondition (and (robot-at ?r ?l) (alarm-button ?l))
31    :effect (forall (?d - door)
32              (and (not (locked ?d)) (open ?d)))))
```

En `alarm-p01.pddl` el robot está en el vestíbulo, la puerta de la farmacia está bloqueada y el único pulsador de alarma está en el pasillo. El plan tiene cinco pasos: abrir la puerta del pasillo, ir allí, dar la alarma, volver y entrar en la farmacia por la puerta que la alarma ha desbloqueado y abierto. El efecto no necesita ningún `when`: desbloquear una puerta que no estaba bloqueada y abrir una que estaba abierta no cambian nada.

Solución 6.3.2. El objetivo es una *condición*: es cierto en un estado si en él no hay ninguna lámpara encendida, y su requisito es `:universal-preconditions`. El efecto es un *cambio*: después de la acción no hay ninguna lámpara encendida, fuera lo que fuera cierto antes, y necesita `:conditional-effects`. Con seis lámparas, el planificador expande los dos cuando instancia la tarea. El objetivo se convierte en una conjunción de seis literales negativos que la búsqueda comprueba en cada estado; el efecto se convierte en seis efectos de borrado de una acción instanciada. Un objetivo nunca cambia el estado, y un efecto nunca lo comprueba.

Sección 6.4 — Los dos juntos: para todo objeto tal que

Solución 6.4.1. Con las cinco lámparas encendidas, pulsar `b1` (lámparas 1 y 2) y `b4` (lámparas 3, 4 y 5) las apaga todas; `b2` y `b5` también. Kairos encuentra primero un plan de seis pulsaciones y después el de dos, `(press b4) (press b1)`. El orden no importa: cada pulsación cambia sus lámparas sea cual sea su estado, así que el estado final de una lámpara depende solo de cuántas veces se ha cambiado. Por la misma razón, pulsar un botón dos veces nunca sirve de nada: la segunda pulsación deshace la primera. Una solución es un *conjunto* de botones, y un planificador que busca secuencias explora muchos

órdenes del mismo conjunto. Esta simetría es la clase de cosa que examina el criterio de Guía.

Solución 6.4.2. El `move` único tiene dos efectos condicionales universales, uno para el carro que sujeta el robot y otro para los artículos que hay en ese carro:

```
; One action for walking and for pushing: the cart that the
; robot holds, if any, comes along, with what is on it.
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (forall (?c - cart)
                 (when (pushing ?r ?c)
                   (and (not (cart-at ?c ?from))
                        (cart-at ?c ?to))))
               (forall (?c - cart ?i - item)
                 (when (and (pushing ?r ?c) (on-cart ?i ?c))
                   (and (not (item-at ?i ?from))
                        (item-at ?i ?to))))))
```

El segundo `forall` cuantifica a la vez sobre un carro y un artículo, con las dos condiciones unidas por `and`; un `when` dentro del `when` del primero no sería PDDL estándar. El plan para `p02` tiene los mismos 25 pasos, con `move` allí donde el capítulo tenía `push-cart`. El dominio tiene 95 acciones instanciadas en lugar de 109 (desaparecen los 14 `push-cart`) y un requisito menos, ya que el `exists` negado del antiguo `move` deja de hacer falta. El coste está en la lectura de los planes: `(move r1 pharmacy lobby)` ya no dice si también se movió un carro, y su carga.

Solución 6.4.3. El problema es `courier-cart-twice-p01.pddl`, y Kairos encuentra para él un plan de nueve pasos: los dos robots van al pasillo y *los dos cogen el carro*; después `r2` lo empuja al laboratorio y `r1` lo empuja a la UCI. El carro acaba en dos habitaciones. Al dominio le faltan dos cosas. `grab-cart` no pregunta si otro robot sujeta ya el carro, y `push-cart` da por hecho que un carro empujado está donde está su robot, lo que solo es cierto si lo sujeta un único robot. La reparación es una precondición:

```
(:action grab-cart
  :parameters (?r - robot ?c - cart ?l - location)
  :precondition (and (robot-at ?r ?l) (cart-at ?c ?l) (free ?r)
                     (not (exists (?o - robot)
                                  (pushing ?o ?c))))
  :effect (and (not (free ?r)) (pushing ?r ?c)))
```

Con ella, el mismo problema (`courier-cart-two-p01.pddl`) es irresoluble, como debe ser. La lección es para el paso E: un modelo probado con un robot está probado para un robot. Cada nueva clase de objeto en el problema merece su propia prueba de la trampa.

Sección 6.5 — Efectos que hay que evitar: el laboratorio estéril

Solución 6.5.1. La esclusa rocía lo que entra, en una mano o en un carro. Se conserva el `move` del laboratorio estéril, y `push-cart` gana un segundo efecto condicional universal, para las muestras que van en el carro:

```
(:action push-cart
  :parameters (?r - robot ?c - cart ?from ?to - location)
  :precondition (and (pushing ?r ?c) (robot-at ?r ?from)
```

```
                    (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (cart-at ?c ?from)) (cart-at ?c ?to)
               (forall (?i - item)
                 (when (on-cart ?i ?c)
                   (and (not (item-at ?i ?from))
                        (item-at ?i ?to))))
               ; the airlock sprays what is on the cart too
               (forall (?i - sample)
                 (when (and (sterile ?to) (on-cart ?i ?c)
                            (not (sealed ?i)))
                   (contaminated ?i)))))
```

Los dos efectos de `push-cart` cuantifican sobre tipos distintos: todo *artículo* del carro se mueve, toda *muestra* sin sellar que va en él se contamina. Para `courier-cart-sterile-p01.pddl` el mejor plan tiene nueve pasos: coger, sellar y cargar cada muestra en la sala A, coger el carro y empujarlo dos veces. El robot sella cada muestra mientras la sujeta, antes de ponerla en el carro, porque `seal` exige `carrying`. Una muestra cargada sin sellar habría que volver a cogerla.

Solución 6.5.2. En `courier-sterile-refuse-domain.pddl`, el `move` no tiene efecto condicional y tiene una precondición más, universal:

```
; The move into a sterile room is not always possible.
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (forall (?i - sample)
                       (imply (and (sterile ?to)
                                   (carrying ?r ?i))
                              (sealed ?i))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

El problema «sin plan» sigue siendo irresoluble: una muestra sin sellar no puede entrar de ninguna manera. El problema «trampa» cambia: su plan tiene ahora ocho pasos y sella la muestra, aunque el objetivo no pide una muestra utilizable, porque sellarla es la única forma de cruzar la puerta. En este modelo `contaminated` nunca puede hacerse cierto, y se podría eliminar.

El primer modelo es fiel a una esclusa que rocía a quien entra; al planificador hay que decirle, mediante el objetivo, que la contaminación importa. El segundo es fiel a una esclusa con un sensor que mantiene la puerta cerrada. Si la puerta real no tiene sensor, el segundo modelo es inseguro de una forma sutil: solo produce planes buenos, pero no puede *representar* lo que ocurre si algo sale mal, por ejemplo para comprobar un plan escrito por otra persona.

Sección 6.6 — El requisito `:adl`

Solución 6.6.1. El fichero `lights-adl-domain.pddl` es válido, y `kairos check` indica las mismas tres características que antes: tipos, precondiciones negativas y efectos condicionales. El informe enumera lo que el fichero *usa*, y `:adl` anuncia cuatro cosas más que no usa: la igualdad, las precondiciones disyuntivas y las precondiciones existenciales y universales.

Un planificador decide si puede tratar un dominio a partir de los requisitos, antes de leer las acciones. Uno sin precondiciones disyuntivas puede aceptar la lista original, donde no aparecen, y tiene derecho a rechazar `:adl`, que las incluye. La lista precisa deja la puerta abierta a más planificadores.

Sección 6.7 — Qué hacen los planificadores con los efectos condicionales

Solución 6.7.1. Tras la instanciación, `(press b1)` tiene efectos condicionales para las dos lámparas conectadas a él, `l1` y `l2`. Para cada lámpara se cumple exactamente una de sus dos condiciones (encendida o apagada), así que las combinaciones son los $2^2 = 4$ estados de las dos lámparas. Una de las cuatro acciones STRIPS, por ejemplo, tiene la precondición `(and (lit l1) (not (lit l2)))` y el efecto `(and (not (lit l1)) (lit l2))`. El botón `b3` tiene tres lámparas y necesita $2^3 = 8$ acciones. Los cinco botones juntos necesitan $4 + 8 + 8 + 8 + 4 = 32$, frente a cinco acciones instanciadas con efectos condicionales. En un tablero donde cada botón cambiara cinco lámparas, cada uno necesitaría 32.

Solución 6.7.2. El dominio del capítulo tiene 31 acciones instanciadas y el de casos tiene 32: el movimiento del pasillo al laboratorio existe dos veces, como `enter-clean` y como `enter-exposed`, y los otros 13 movimientos son los de siempre. Las dos búsquedas visitan los mismos 55 estados. El segundo dominio es la primera compilación, una acción por combinación de condiciones, hecha a mano y con algo de cuidado: las tres condiciones de la esclusa darían 2^3 combinaciones, pero siete de ellas tienen el mismo efecto, y tres acciones con precondiciones más amplias cubren las ocho. Un modelador que divide los casos a mano ve esos ahorros; un compilador que lo hace a ciegas puede no verlos.

Sección 6.8 — El método MODEL: efectos que dependen de la situación

Solución 6.8.1. Tabla de estado: dónde está el pintor (`painter-in`, dinámico), las puertas (`door`, estático), dónde está cada mueble (`in`, dinámico) y tres propiedades que solo se vuelven ciertas: `covered`, `painted` y `stained`. Tabla de acciones:

Acción	Siempre cambia	También cambia
`walk`	el pintor cambia de habitación	
`cover`	el mueble queda cubierto	
`carry-out`	el pintor y el mueble cambian de habitación	
`spray`	la habitación queda pintada	*todo* mueble, *si* está en la habitación y no está cubierto, queda manchado

Pintar siempre es posible, así que los muebles van en su efecto, no en su precondición:

```
35  ; Always possible; the stains depend on what is in the room.
36  (:action spray
37    :parameters (?r - room)
38    :precondition (painter-in ?r)
39    :effect (and (painted ?r)
40                 (forall (?f - furniture)
41                   (when (and (in ?f ?r) (not (covered ?f)))
42                     (stained ?f))))))
```

Para `paint-p01.pddl` el plan es `walk`, `cover` la lámpara, `cover` la cama, `spray`: cuatro pasos. Sacar los dos muebles llevaría más tiempo. El objetivo debe contener `(not (stained bed))` y `(not (stained lamp))`; con `(painted bedroom)` solo, el plan es `walk`, `spray`, y los muebles quedan estropeados. Ese problema es la prueba de la trampa de este modelo.

Solución 6.8.2. En `briefcase-implied-domain.pddl`, `put-in` borra el hecho `at` de la cosa, `carry` mueve solo el maletín y `take-out` gana un parámetro de lugar y añade `at` para el lugar donde está el maletín. El dominio es STRIPS puro con tipos. El plan para la misma historia tiene cuatro pasos en lugar de tres: meter el informe, sacar las llaves, llevar el maletín y sacar el informe en la oficina, porque solo entonces es cierto `(at report office)`.

	Efecto condicional	Sin `at` para lo que va dentro
Fidelidad	una cosa dentro está en el lugar del maletín	una cosa dentro no está en ninguna parte hasta que se saca; los objetivos sobre lugares obligan a un `take-out`
Tamaño	10 acciones instanciadas, 7 variables de estado	12 acciones instanciadas, 3 variables de estado
Guía	3 pasos	4 pasos; estados más sencillos
Claridad	una regla que explicar: el contenido sigue al maletín	tres acciones simples
Portabilidad	`:conditional-effects`, `:negative-preconditions`	cualquier planificador

El efecto condicional ha costado dos requisitos. Ha comprado el derecho a preguntar dónde está una cosa mientras viaja, y un objetivo que se cumple en cuanto llega el maletín. Si las únicas preguntas que se hacen son sobre cosas ya sacadas, la versión STRIPS es la mejor elección.

C.7. Soluciones del Capítulo 7

En `examples/ch07/solutions` están los ficheros PDDL de estas soluciones.

Sección 7.1 — Un problema a partir de una historia: el camarero

Solución 7.1.1. Una ficha posible (otras son igual de buenas, si dicen lo que deciden):

Historia. Una cafetera atiende a una cola de personas, una bebida para cada una, y hay que rellenar sus depósitos cuando se vacían.

Agentes. Uno: quien maneja la máquina (coloca las tazas, prepara las bebidas y rellena). Las personas de la cola no son agentes: solo traen pedidos.

Éxito. Cada persona de la cola tiene la bebida que ha pedido.

Alcance. *Dentro*: el nivel de agua, de café y de leche, medido en tazas; si hay una taza bajo la boquilla; qué pedidos están servidos. *Fuera*: la temperatura y el tiempo que tarda una bebida (de eso se encarga la máquina); de dónde salen las recargas (un armario que nunca se vacía); el pago; la limpieza; la gente que cambia de opinión.

Supuestos. El orden de la cola es fijo. Una recarga llena el depósito por completo. La máquina nunca falla.

Cada cosa que queda «fuera» tiene un responsable: la electrónica de la máquina, el material de oficina, nadie (aceptamos la simplificación). Los niveles son cantidades pequeñas y acotadas de las que solo importa cuánto valen: una cadena de sucesores por depósito.

Solución 7.1.2. El problema es `waiter-dessert-p01.pddl`: tres platos pedidos por `t1` y dos hechos para el orden, `(before soup1 fish1)` y `(before fish1 cake1)`. El dominio no cambia, porque el orden de los platos es una tabla estática. El mejor plan tiene nueve pasos y dos viajes: la sopa y el pescado juntos, y después la tarta. Un solo viaje es imposible con dos huecos y tres platos. Observa que `(before soup1 cake1)` no hace falta: la tarta espera al pescado, que espera a la sopa.

Solución 7.1.3. Con el objetivo `(served soup1)` el plan es cargar, moverse y servir: los otros tres platos se quedan en el pase, y el mundo lo acepta (la noche no es un éxito, pero el objetivo no pedía tanto). No hay trampa en el sentido de un plan imposible. Hay un objetivo débil: un plan también podría cargar un plato y dejarlo en la bandeja para siempre. Si eso importa, el objetivo tiene que decirlo, por ejemplo con `(empty s1)` y `(empty s2)`.

Sección 7.2 — Patrones para el estado

Solución 7.2.1. El problema `river-p03.pddl` tiene cuatro cosas y los mismos dos hechos `eats`, con los nombres cambiados. Kairos encuentra un plan de nueve travesías: las siete clásicas y un viaje más de ida y vuelta para el perro, al que se puede dejar con cualquiera. Con una acción por animal, un animal nuevo supone una acción nueva, y una enemistad nueva obliga a editar la precondición de cada acción que pudiera dejar juntos a los dos. Con la tabla estática, ambas cosas son líneas del problema.

Solución 7.2.2. Los hechos espejo lo expresan omitiendo un hecho: el problema conserva `(connected lobby corridor)` y quita `(connected corridor lobby)`. La codificación con `or` no puede: lee cada hecho en los dos sentidos. Con la conexión de un solo sentido, `courier-p03` no tiene plan (Kairos explora 16 estados y se detiene): el robot puede entregar los medicamentos en la sala A, pero después el único camino de vuelta a la base de carga va del pasillo al vestíbulo. Una conexión de un solo sentido es una acción sin vuelta atrás, y acaba de convertir toda la parte derecha del mapa en un callejón sin salida.

Solución 7.2.3. En las dos codificaciones el dominio no cambia. El problema con huecos gana un objeto y un hecho (`s3`, `(empty s3)`); el problema con la cadena gana un objeto y un hecho (`n3`, `(next n2 n3)`). Los ficheros son `waiter-p06.pddl` y `waiter-count-p04.pddl`. El mejor plan tiene 11 pasos en los dos, igual que con dos sitios: con cuatro platos para dos mesas, un tercer sitio no ahorra ningún viaje.

Estados explorados para `p02`	Huecos	Cadena
Bandeja para dos	220	123
Bandeja para tres	566	143

La cadena crece un poco, porque hay más cuentas posibles. Los huecos crecen mucho más deprisa: k platos pueden repartirse en tres huecos de muchas más maneras que en dos, y cada manera es un estado distinto para el planificador y el mismo estado para el camarero. La simetría entre huecos crece con el número de disposiciones, y por eso los huecos son una mala codificación de una capacidad grande.

Solución 7.2.4. (a) Lo mantiene: se sabe que el valor antiguo es verdadero, y se borra. (b) No lo mantiene: si el robot no está en `?from`, el borrado no hace nada y el robot gana una segunda posición. La precondición es lo que ata el borrado al valor antiguo real. (c) Lo mantiene, por

la regla de que las adiciones se aplican después de los borrados: el testigo se borra y se vuelve a añadir. La acción no hace nada, lo que es una razón para prohibir que `?a` sea igual a `?b`, pero no rompe nada. (d) Lo rompe: `(free ?r)` sigue siendo verdadero, así que la acción se puede aplicar otra vez y el robot lleva dos artículos, y luego tres. Falta el borrado del *gemelo*.

Sección 7.3 — Patrones para el control

Solución 7.3.1. El plan corta el asiento y una pata, avanza a la fase de ensamblado y puede seguir con los dos ensamblados, el segundo avance y dos pinturas. Nunca puede cortar `leg2`: `cut` necesita `(now cutting)`, y ninguna acción devuelve el testigo. El objetivo queda fuera de alcance desde el tercer paso, y nada parecía ir mal cuando se dio ese paso. Un testigo de fase convierte el «demasiado pronto» en un callejón sin salida. El planificador tiene que descubrir que antes hay que cortar todas las piezas, y la relajación de borrados no se lo dice, porque sin borrados todas las fases siguen activas.

Solución 7.3.2. Sin `wait` el problema es irresoluble, y Kairos lo dice tras un solo estado (`patrol-nowait-domain.pddl`). En el primer turno, el único movimiento del robot es entrar en `h1`, donde está la máquina. Ninguna acción del robot es aplicable, el turno no pasa nunca y el mundo se queda congelado. `wait` modela la decisión más común de un agente en un mundo que se mueve: no hacer nada por ahora. Con turnos, no hacer nada tiene que ser una acción, porque solo las acciones pasan el testigo.

Solución 7.3.3. El dominio es `courier-narrow-domain.pddl`. El hecho de disponibilidad es la ausencia de `(taken ?l)`; `enter` adquiere la habitación y `leave` la libera:

```
  ; Acquire the room on the way in...
  (:action enter
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                       (narrow ?to) (not (taken ?to)))
    :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
                 (taken ?to)))

  ; ...and release it on the way out.
  (:action leave
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                       (narrow ?from) (not (narrow ?to)))
    :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
                 (not (taken ?from))))
```

El `move` normal se limita a las habitaciones que no son estrechas, así que las tres acciones se excluyen entre sí. Los dos robots de `courier-narrow-p01.pddl` entran en la farmacia uno detrás de otro, seis pasos en total; la prueba de la trampa `p02`, con los dos dentro, es irresoluble. Esto es una acción por caso (Capítulo 6): la alternativa es un único `move` con efectos condicionales sobre `taken`.

Sección 7.4 — Patrones para las acciones

Solución 7.4.1. `move` y `push-cart`: cada una se deshace con la misma acción intercambiando `?from` y `?to`, siempre que el mapa tenga hechos espejo. `grab-cart` y `release-cart` son un par inverso. `pick` y `drop` son un par inverso para un artículo que está en el suelo. Para un artículo que está en el carro, la inversa de `pick` es `load`,

que lo vuelve a poner en el carro, y la inversa de `load` es `pick`. Toda acción tiene inversa, así que con un solo robot el dominio no tiene callejones sin salida: desde cualquier estado alcanzable, el robot puede deshacer lo hecho y probar otra cosa. (El defecto con dos robots del Ejercicio 6.4.3 es otra cuestión: allí el estado deja de tener sentido.)

Solución 7.4.2. El dominio `blocks-coarse-domain.pddl` tiene tres acciones: `move-to-block`, `move-to-table` y `move-from-table`. Con dos no bastaría, porque la mesa no es un bloque: nunca está `clear` ni deja de estarlo, y en ella caben cuantos bloques se quiera. La inversión de la torre lleva tres pasos:

```
$ kairos blocks-coarse-domain.pddl blocks-coarse-p01.pddl
; cost = 3
(move-to-table c b)
(move-to-block b a c)
(move-from-table a b)
```

El modelo de cuatro acciones necesita seis pasos para lo mismo, el doble, porque cada movimiento es coger y dejar. Lo que ese modelo puede expresar y este no es todo lo relativo a la mano: un objetivo o un estado inicial en el que la mano sostiene un bloque, dos brazos que comparten los bloques, una mano que falla entre coger y colocar. Si nada de eso está en la historia, el modelo fusionado es mejor: planes de la mitad de longitud y ningún hecho `holding`.

Sección 7.5 — Antipatrones

Solución 7.5.1. Los tres antipatrones:

- **Borrado olvidado.** `pick` no borra `(ball-at ?b ?r)`: una bola está en la pinza y sigue en la habitación. El plan para `p01` no es más corto (las bolas tienen que viajar igualmente), así que falta el síntoma habitual. La línea `[sas]` lo delata: `13 variables (3 from 3 mutex groups, 10 binary)`, mientras que el dominio reparado tiene `7 variables (7 from 7 mutex groups)`. El planificador no ha podido demostrar que una bola está en un solo sitio. Una prueba de la trampa también lo muestra: `gripper-buggy-p02.pddl` quiere una bola en las dos habitaciones, y lo consigue en tres pasos.
- **Gemelos desacompasados.** `busy` es el gemelo de `free`; `pick` lo activa y `drop` nunca lo desactiva. Nadie lo lee, así que de momento no falla nada. El arreglo es eliminarlo: un gemelo que ninguna precondición usa no es más que un riesgo.
- **Parámetro ocioso.** `move` tiene un parámetro de pinza que no usa. Los planes lo muestran, `(move rooma roomb left)`, y también el recuento: 40 acciones instanciadas frente a 36.

El dominio reparado es `gripper-domain.pddl`. Con él, la prueba de la trampa es irresoluble. Este ejercicio deja una moraleja para el paso E: el problema típico pasaba con los tres fallos dentro.

Solución 7.5.2. Es un objetivo en el estado inicial. La muestra empieza en la sala A, así que no está en el pasillo, y empieza limpia, así que no está contaminada: los dos literales negativos son verdaderos antes de cualquier acción, y el plan vacío es correcto. Los objetivos negativos son propensos a esto por la hipótesis del mundo cerrado: lo que `:init` no enumera es falso, así que un objetivo hecho solo de negaciones es verdadero al principio salvo que el estado inicial

diga lo contrario. Un objetivo debería contener al menos una cosa que haya que *conseguir*; aquí, `(item-at sample1 lab)`.

Sección 7.6 — Depurar un modelo

Solución 7.6.1. Basta con el paso 1. La nota dice que `(box-at c12)` no se puede alcanzar nunca: `push` lo añade, pero nunca es aplicable. Para llevar la caja de `c22` a `c12`, el empujón tiene que ir hacia el norte, con el jugador en `c32`; la instancia necesita `(adjacent c32 c22 north)` y `(adjacent c22 c12 north)`. Si se busca `north` en el fichero, aparece la dirección declarada como objeto y no usada en ningún hecho: faltan los ocho hechos para ir hacia arriba. Quien tecleó la cuadrícula escribió las vecindades verticales en un solo sentido, que es el error de la mitad de los hechos espejo. Con la cuadrícula completa (`sokoban-p04.pddl`) el plan tiene cuatro pasos: tres desplazamientos para ponerse debajo de la caja y un empujón hacia el norte.

Solución 7.6.2. Trabaja hacia atrás desde el objetivo. `(served soup1)` solo lo añade `serve`, así que relaja primero `serve`: quítale las precondiciones de una en una, empezando por las más complejas o las que se escribieron en último lugar (aquí el `forall`, después `ordered`, después `on-tray` y por último la posición). Si al quitar una aparece un plan, esa precondición nombra el hecho que hay que perseguir, y se hace la misma pregunta un nivel más abajo: ¿qué acción debería añadirlo, y cuál de *sus* precondiciones falla? Si el obstáculo es `on-tray`, `load` es la siguiente acción que relajar. En general: primero la última acción del plan que esperabas, después la anterior. Cada ronda retrocede un paso a lo largo del plan que debería haber existido, hasta dar con una acción cuyo efecto ausente o cuya precondición imposible explica todo lo que viene después.

Sección 7.7 — El método MODEL: el método clásico, consolidado

Solución 7.7.1. Ficha del modelo. Historia: un transbordador lleva coches entre dos orillas, dos a la vez como mucho. Agentes: el transbordador. Éxito: cada coche en la orilla que quiere. Alcance, dentro: dónde están el transbordador y cada coche, cuántos coches hay a bordo; fuera: el tiempo, el orden de los coches en la cola, las tarifas. Supuestos: un coche a bordo sigue a bordo hasta que se le hace bajar; los coches no se mueven por sí solos.

Tabla de estado. `(ferry-at ?s)` y `(car-at ?c ?s)`, dinámicos, cada uno con el invariante de a lo sumo uno; `(aboard ?c)`, dinámico (un coche a bordo no tiene orilla: su posición es la del transbordador, el «no guardes lo que se puede deducir» del Capítulo 6); `(load ?n)`, dinámico, y `(next ?n ?m)`, estático: una cadena de sucesores.

Tabla de acciones. `sail` (el transbordador cambia de orilla), `board` (un coche que está en la orilla del transbordador sube a bordo y la cuenta aumenta) y `debark` (la inversa). Las tres forman pares inversos, así que no hay callejones sin salida. El dominio es `ferry-domain.pddl`:

```
(:action board
  :parameters (?c - car ?s - shore ?n ?m - count)
  :precondition (and (ferry-at ?s) (car-at ?c ?s)
                     (load ?n) (next ?n ?m))
  :effect (and (not (car-at ?c ?s)) (aboard ?c)
               (not (load ?n)) (load ?m)))
```

```
27
28  (:action debark
29    :parameters (?c - car ?s - shore ?n ?m - count)
30    :precondition (and (ferry-at ?s) (aboard ?c)
31                       (load ?m) (next ?n ?m))
32    :effect (and (not (aboard ?c)) (car-at ?c ?s)
33                 (not (load ?m)) (load ?n))))
```

Batería de pruebas. Trivial: un coche. Típica (`ferry-p01.pddl`): tres coches hacia el norte y uno hacia el sur, resuelta en 11 pasos con tres travesías; el transbordador trae el coche que va al sur en el viaje de vuelta. Sin plan y de la trampa a la vez (`ferry-p02.pddl`): tres coches a bordo al mismo tiempo, irresoluble.
Comparativa. El rival de la cadena son dos huecos, `(deck-free ?k)` y `(on-deck ?c ?k)`. Fidelidad: la misma, salvo que importen las posiciones en la cubierta (el primero que sube es el último que baja). Tamaño y Guía: un número parecido de acciones instanciadas; los huecos añaden la simetría de qué coche va en qué lado de la cubierta. Claridad: los huecos se leen mejor en los planes. Portabilidad: los dos son STRIPS con tipos e igualdad. Para la historia tal como se cuenta, la cadena; con un transbordador que descarga en orden, los huecos, o una cadena de posiciones.

Solución 7.7.2. No hay una única respuesta. Una buena respuesta nombra, para un dominio tuyo: las tablas estáticas y si queda alguna regla fijada dentro de una acción; cada propiedad de un solo valor y la acción que mantiene su invariante; cada capacidad y si es una cadena o huecos; cada acción y su inversa, o la razón por la que no la tiene. La medición debería dar las acciones instanciadas, las variables de estado y los estados explorados antes y después del cambio, en el mismo problema, y decir qué criterio del paso L se ha movido.

C.8. Soluciones del Capítulo 8

En `examples/ch08/solutions` están los ficheros PDDL de estas soluciones.

Sección 8.1 — Lo que un hecho no puede decir

Solución 8.1.1. (a) Hechos: seis lugares, y el ascensor se mueve entre vecinos. Bastan un predicado `(lift-at ?f)` y un hecho estático `(above ?f1 ?f2)`. (b) Una magnitud: se le restan precios de cualquier importe. (c) Un nombre: nadie suma números de cama. (d) Depende de cómo se caliente el horno. Si una acción «precalentar» lo lleva a la temperatura deseada, un hecho `(hot-enough ?o)` es todo lo que piden las recetas: solo importa un umbral. Si calentar suma grados poco a poco y recetas distintas piden temperaturas distintas, es una magnitud. (e) Un hecho. (f) Una magnitud: trabajos de longitudes distintas restan cantidades distintas.

Sección 8.2 — Funciones: números en el estado

Solución 8.2.1. Hay cinco funciones: `(altitude ?d - drone)`, en metros, dinámica; `(payload-limit ?d - drone)`, en gramos, estática; `(payload ?d - drone)`, los gramos que lleva ahora, dinámica; `(distance ?from ?to - site)`, estática; y `(weight ?p -`

`parcel)`, estática. La historia no menciona `payload`, pero el límite no sirve de nada sin la suma que se compara con él: es el fluente resumen de la Sección 8.4.

Solución 8.2.2. (1) `(battery r1 80)` escribe un valor como argumento: un error de sintaxis, el comprobador rechaza el fichero. Debería ser `(= (battery r1) 80)`. (2) La distancia se da en un solo sentido. El fichero es válido; el robot puede ir del vestíbulo a la base de carga y no volver nunca, porque `(distance dock lobby)` está indefinida. (3) `(= (consumption) 1)` olvida el robot: `consumption` tiene un parámetro, así que el comprobador informa de un número de argumentos incorrecto. Si la función se hubiera declarado sin parámetros, la línea sería correcta.

Sección 8.3 — Comparar y cambiar números: la batería

Solución 8.3.1. La primera entrega y la vuelta a la base de carga necesitan $64 \times 1{,}5 = 96$ unidades, y la segunda entrega 70,5, como antes. Con 50 unidades el robot ni siquiera puede hacer la primera (57 unidades para llegar a la sala A), así que tiene que cargar antes de salir, y otra vez entre las dos entregas: dos recargas, 20 acciones. El problema es `courier-battery-half.pddl`, y el mejor plan de Kairos empieza con `(charge r1 dock)` y después repite el plan del capítulo. Los primeros planes que imprime son más largos (22 acciones en nuestra ejecución, con movimientos de ida y vuelta): la heurística no ve que la batería se agota, como explica la Sección 8.7.

Solución 8.3.2. El dominio `courier-reserve-domain.pddl` añade una función estática `(reserve ?r)` y cambia la precondición de `move`:

```
    (>= (- (battery ?r)
           (* (distance ?from ?to) (consumption ?r)))
        (reserve ?r)))
```

El efecto no cambia. El plan del capítulo llegaba a la base de carga con 4 unidades, por debajo de la reserva de 10, así que su movimiento del vestíbulo a la base deja de ser aplicable. Solo la ida y vuelta a la sala A necesita ahora 96 unidades de las 90 que se pueden usar. El planificador encuentra un plan de 21 acciones que reparte el trabajo de otra manera: recoge primero la muestra y la deja en el pasillo, recoge los medicamentos, recarga cuando le quedan 16 unidades y entrega las dos cosas en la segunda ronda. Una reserva de 10 cuesta dos acciones y otro orden de trabajo.

Solución 8.3.3. El dominio no cambia: el deseo es una condición numérica en el objetivo (`courier-battery-end.pddl`):

```
(:goal (and (item-at meds1 ward-a) (item-at sample1 icu)
            (>= (battery r1) 50))))
```

El plan del capítulo termina con 29,5 unidades. El robot no puede recargar después de la última entrega y quedarse en la UCI, así que el planificador recarga más tarde: recoge la muestra *antes* de la segunda visita a la base de carga, recarga con la muestra a bordo y va directamente a la UCI, adonde llega con 53,5 unidades. El plan tiene 24 acciones y dos recargas.

Sección 8.4 — Números que nunca cambian

Solución 8.4.1. En `courier-battery-load-domain.pddl` el consumo de un movimiento es

```
(* (* (distance ?from ?to) (consumption ?r))
   (+ 1 (* 0.02 (load ?r)))))))
```

en la precondición y en el efecto. La expresión lee ahora un fluente dinámico, la carga transportada, así que el planificador no puede reducirla a una constante al instanciar la acción. En la prueba el robot gasta 2 unidades por metro. Va de la base de carga al vestíbulo vacío ($4 \times 2 = 8$, quedan 92), recoge los dos artículos (7 kg, un factor de 1,14) y recorre $10 + 12 + 5 = 27$ metros: $27 \times 2 \times 1{,}14 = 61{,}56$. Llega con 30,44 unidades, que es lo que muestra la traza de Kairos.

Solución 8.4.2. Dieciséis de los diecisiete valores pertenecen a funciones estáticas (`distance`, `battery-capacity`, `consumption`): el planificador las sustituye por sus valores al instanciar. Solo `(battery r1)` la cambia alguna acción. El problema de carga también indica uno, `(load r1)`: la capacidad y los tres pesos son estáticos.

Solución 8.4.3. En `truck-p04.pddl` el camión está en Cuenca con 40 litros, Valencia está a 200 km (50 litros) y la gasolinera está en Teruel, a 150 km de Cuenca y a 140 de Valencia. El camión va a Teruel (37,5 litros, quedan 2,5), reposta cuatro veces (42,5; con tres tendría 32,5, menos de los 35 que necesita el último tramo) y va a Valencia: seis acciones.

Sección 8.5 — Asignar, escalar y preguntar

Solución 8.5.1. El dominio `jugs-closed-domain.pddl` es el del Listado 8.6 sin `fill` ni `empty`; el problema es `jugs-p03.pddl`. El plan más corto tiene siete trasvases, y Kairos lo demuestra:

```
(pour-until-full j8 j5)
(pour-until-full j5 j3)
(pour-all j3 j8)
(pour-all j5 j3)
(pour-until-full j8 j5)
(pour-until-full j5 j3)
(pour-all j3 j8)
```

Los estados son (8,0,0), (3,5,0), (3,2,3), (6,2,0), (6,0,2), (1,5,2), (1,4,3), (4,4,0).

Solución 8.5.2. `jugs-when-domain.pddl` tiene un único `pour` cuyos dos efectos condicionales llevan las condiciones que distinguían las dos acciones:

```
(when (<= (+ (amount ?from) (amount ?to)) (volume ?to))
  (and (increase (amount ?to) (amount ?from))
       (assign (amount ?from) 0)))
```

y lo mismo para el otro caso. El plan tiene los mismos seis pasos. Fidelidad: igual. Tamaño: 6 acciones instanciadas en lugar de 8, cada una algo más pesada. Guía: más o menos igual aquí. La claridad es cuestión de gustos: una acción corresponde al gesto único de verter, dos acciones hacen visibles en el plan los dos resultados (`pour-all`, `pour-until-full`), lo que ayuda a leerlo. La portabilidad favorece a las dos acciones: las condiciones numéricas dentro de `when` las aceptan menos planificadores que las precondiciones numéricas simples.

Solución 8.5.3. El volumen podría reducirse a la mitad indefinidamente: 2; 1; 0,5; 0,25... Cada reducción a la mitad es un estado nuevo, así que los estados alcanzables no se acaban nunca y, para un problema sin plan, el planificador no podría demostrarlo (Sección 8.7). La

precondición también es fiel al laboratorio: una gota no se puede partir con una pipeta.

Solución 8.5.4. En `elevator-domain.pddl`, dónde están el ascensor y cada persona es cuestión de hechos (`lift-at`, `person-at`, `inside`); cuántas personas hay dentro es un número, `(passengers)`, que se compara con `(max-passengers)` en `board` y que actualizan `board` y `leave`. El recuento es un resumen de los hechos `inside`. El plan para tres personas y un ascensor para dos tiene 14 acciones: el ascensor baja, sube a dos personas, vuelve a bajar y sube a la tercera.

Sección 8.6 — Valores que nunca se dieron

Solución 8.6.1. `kairos check` dice `valid, plannable`: la sintaxis no tiene nada mal. El planificador responde `unsolvable`, con la nota citada en la Sección 8.9: `pick` nunca es aplicable a la comida, porque su precondición lee `(weight meal1)`. Con cero por defecto, la comida no pesaría nada, los tres artículos viajarían juntos en un solo viaje y el plan parecería mejor que el correcto. Nadie sospecharía.

Solución 8.6.2. Fidelidad: los mismos planes, siempre que cada conexión tenga su distancia en un sentido. Tamaño: el doble de movimientos candidatos, de los que la instanciación conserva el mismo número (un movimiento cuya distancia está indefinida nunca es aplicable). Guía: sin cambios. Claridad: peor en el dominio (dos acciones que se diferencian en el orden de dos argumentos, y planes con dos nombres para moverse) y mejor en el problema (la mitad de líneas). Portabilidad: sin cambios. Un veredicto justo: cambia un error frecuente en los problemas por un dominio más difícil de explicar, y solo desplaza la trampa, ya que olvidar una conexión en los dos sentidos es tan silencioso como antes. Generar el fichero del problema con un script es mejor remedio.

Sección 8.7 — Por qué los números dificultan la planificación

Solución 8.7.1. (a) Finito: la batería solo toma valores entre 0 y la capacidad que son la carga inicial menos una suma de consumos. (b) Finito: `packed` está limitado por el espacio de la caja, y las existencias solo crecen desde cero. (c) Infinito: las existencias se pueden aumentar indefinidamente. (d) Infinito, y es fácil no verlo: el volumen está acotado, pero alternar diluir y tirar la mitad devuelve el volumen a donde estaba y reduce la concentración a la mitad cada vez, sin fin. Un límite inferior para la concentración en `dilute` lo cerraría. (e) Doblemente infinito.

Solución 8.7.2. En nuestras ejecuciones el planificador exploró unos 307 000, 604 000 y 1 153 000 estados: el doble de tiempo, el doble de estados y la misma respuesta, `timeout`, con código de salida 11. Ninguna de las respuestas es «irresoluble». Un límite de tiempo agotado dice que no se encontró ningún plan en los estados visitados. Puede que el plan esté más lejos, que la heurística se equivoque o que no haya plan y el espacio de estados sea infinito; el planificador no puede saber cuál de las tres. Para saber más, acota las magnitudes, de modo que el planificador pueda agotar los estados, o relaja el objetivo como en el procedimiento de depuración del Capítulo 7.

Sección 8.8 — El método MODEL: magnitudes

Solución 8.8.1. `courier-slots-domain.pddl` da al robot objetos hueco; `pick` ocupa un hueco vacío del robot y `drop` lo libera. Con dos huecos, los tres artículos necesitan dos viajes y el plan tiene 12 acciones, como con pesos.

	Pesos (números)	**Huecos**
Fidelidad	Límite en kilos: dos artículos ligeros o uno pesado	Límite en piezas: dos artículos cualesquiera, pesen lo que pesen
Tamaño	62 acciones instanciadas	110: cada recogida y entrega, una vez por hueco
Guía	Un fluente numérico; la relajación ignora el límite	Clásica; pero los dos huecos son intercambiables, una simetría
Claridad	Cercana a la historia	Un tipo más, tres predicados y un parámetro más
Portabilidad	Necesita `:numeric-fluents`	STRIPS simple con tipos

Veredicto: huecos cuando el límite del robot real es un número de plazas (una bandeja con dos compartimentos) o el planificador es clásico; pesos cuando el límite es un peso o un volumen. No son dos codificaciones de lo mismo salvo que todos los artículos pesen lo mismo.

Solución 8.8.2. Da a `move` un hecho estático más y una acción más. Un hecho `(long ?from ?to)` marca las dos conexiones; `move` exige que no esté y baja un nivel, y una segunda acción, `move-long`, lo exige y baja dos niveles, con tres parámetros de nivel y dos precondiciones `next`, `(next ?mid ?before)` y `(next ?after ?mid)`. El tamaño crece poco, porque los hechos estáticos fijan el parámetro adicional. Pero cada cantidad nueva necesita su propia acción, y una cantidad de k niveles necesita k precondiciones `next`: llegados a ese punto, el número sale más barato.

Solución 8.8.3. *Ficha del modelo.* Historia: un robot riega bancales con un depósito que rellena en un grifo. Éxito: cada bancal ha recibido lo que necesita. Dentro: el agua del depósito, la necesidad de cada bancal. Fuera: las distancias y la batería, el tiempo de riego. Supuesto: el grifo nunca se seca.

Tabla de estado. Dónde está el robot: un hecho, dinámico. Agua en el depósito: una magnitud, dinámica, entre 0 y el tamaño del depósito, mantenida por la precondición de `water` y el `assign` de `refill`. Necesidad de un bancal: una magnitud, dinámica, desde su valor inicial hasta 0, mantenida por la precondición de `water`. Tamaño del depósito y dosis: estáticos.

Tabla de acciones. `go` (origen, destino); `refill` (en un grifo, depósito no lleno: el depósito queda lleno); `water` (en un bancal, una dosis en el depósito, el bancal necesita al menos una dosis: los dos bajan una dosis).

Los ficheros son `greenhouse-domain.pddl` y `greenhouse-p01.pddl`. La prueba típica da 10 acciones: rellenar, tres dosis a los tomates, volver al grifo, rellenar, dos dosis a las lechugas. Para las demás pruebas: un problema con un solo bancal que necesita 2 litros y el depósito lleno (trivial); un bancal que necesita 5 litros, que las dosis de 2 nunca dejan en cero (sin plan, y demostrado, porque las dos magnitudes están acotadas); y un objetivo `(> (tank) 6)` para la trampa. Observa que el objetivo usa igualdades, `(= (need tomatoes) 0)`: aquí no hay peligro, porque todas las cantidades son números enteros.

C.9. Soluciones del Capítulo 9

Los ficheros PDDL de las soluciones están en `examples/ch09/solutions`.

Sección 9.1 — Cuando no basta con un plan cualquiera

Solución 9.1.1. (a) El número de movimientos; la longitud del plan es exactamente eso. (b) El dinero: el combustible gastado, en euros, frente a unos ingresos fijos por entrega. La longitud del plan no es justa: un desplazamiento largo y uno corto cuentan lo mismo. (c) El tiempo, y como cada acción tarda más o menos lo mismo, la longitud del plan es un buen sustituto. (d) La comida que se tira, en gramos o en euros. La longitud del plan no dice nada de ella: dos planes de la misma longitud pueden usar ingredientes distintos.

Solución 9.1.2. Sin métrica, el número es la longitud del plan, 4. Con la métrica, es el valor de `(total-cost)` al final del plan, en metros: 20 y después 16. En la primera ejecución todos los planes tienen cuatro acciones, así que el primer plan no se puede mejorar y no se imprime nada más. En la segunda, el primer plan encontrado es el mismo, existe uno mejor, y la búsqueda anytime lo encuentra.

Sección 9.2 — Costes de acción

Solución 9.2.1. En `courier-cost-handling-domain.pddl` las acciones `pick` y `drop` ganan un incremento de 3 del coste total. El mejor plan de `courier-cost-handling-p02.pddl` cuesta 183: los mismos 159 metros y ocho manipulaciones, las mínimas posibles para cuatro artículos, a 3 cada una. El plan del capítulo tenía diez manipulaciones, porque dejaba la muestra en el pasillo y volvía a cogerla: ahora costaría 159 + 30 = 189. El planificador encuentra otro orden de las cuatro entregas, de nuevo solo con `r1`, que recorre los mismos metros sin la parada. Cuando manipular era gratis, los dos planes eran igual de buenos y el planificador no tenía ningún motivo para preferir el ordenado.

Solución 9.2.2. El coste de una acción tiene que ser un número no negativo o una función *estática*. `(load ?r)` cambia con las acciones, así que el coste de un desplazamiento dependería del estado, y el dominio tendría un segundo fluente que cambia. El modelo es PDDL legal si se usa `:numeric-fluents` en su lugar: mantén la carga como en el Capítulo 8, añade un contador como `(effort)`, increméntalo en `move` con una expresión que lea la carga y minimiza el contador. Un planificador limitado a costes de acción no lo leerá.

Solución 9.2.3. El plan tiene 18 desplazamientos: 4 + 6 + 6 + 10 + 12 para entregar los medicamentos (38), 12 + 10 de vuelta al vestíbulo con la muestra dejada en el pasillo (60), 10 + 12 para llevar la comida a la sala B (82), 12 + 8 para llevar la muestra al laboratorio (102), 8 + 10 + 6 hasta la farmacia (126) y 6 + 10 + 12 + 5 hasta la UCI: 159. El validador está de acuerdo: `plan valid, makespan 28, metric 159`.

Sección 9.3 — Costes a partir de datos: una cocina con precios

Solución 9.3.1. En `kitchen-p02.pddl` el precio de la mantequilla es 0. Uno de los dos platos lleva la mantequilla y el otro una botella de

aceite: las grasas cuestan 0 + 2 en lugar de 2 + 2, y la comida cuesta 8 en lugar de 10. Kairos pone la mantequilla en el bizcocho. Qué plato la lleve no importa para el coste, así que la tortilla habría sido igual de buena.

Solución 9.3.2. Con métrica, nunca en un plan óptimo: el segundo huevo cuesta su precio y no consigue nada nuevo, porque `(with-egg cake1)` ya era verdadero. Sin métrica, los planes más cortos tampoco lo hacen, porque es una acción más, pero nada impide que un planificador satisfactorio devuelva un plan así. Si el dominio debe prohibirlo es una cuestión de fidelidad: si el cocinero nunca lo haría, una precondición `(not (with-egg ?d))` elimina planes que no se quieren y reduce el espacio de búsqueda, al precio de `:negative-preconditions`. Una métrica hace que el desperdicio sea poco atractivo; no lo hace imposible.

Sección 9.4 — El planificador en acción: búsqueda anytime

Solución 9.4.1. En nuestra máquina:

Opción	Coste	Motivo de parada	Tiempo total
`--no-anytime`	188	`no improvement`	2 ms
`--target-cost 165`	164	`target reached`	0,4 s
ninguna	159	`proven optimal`	4,1 s

Un robot que tiene que arrancar en menos de una décima de segundo puede usar `--timeout 0.1`, que en nuestras ejecuciones devolvió un plan de coste 168: en los primeros 14 milisegundos se encuentran tres planes, y la búsqueda anytime entrega el mejor de ellos cuando se acaba el tiempo. `--no-anytime` desperdicia casi toda la décima de segundo.

Solución 9.4.2. (1) Cambiar lo que se mide: el tiempo en lugar de los metros. Con acciones durativas y el *makespan* como métrica (Capítulo 10), dos robots que trabajan a la vez terminan antes. Esto supone que el hospital quiere la ronda terminada pronto. (2) Mantener los metros y añadir un límite: un máximo de metros por robot y por ronda, con un contador para cada robot y una precondición en `move`, como el presupuesto de la Sección 9.7. Esto supone que lo que importa es el desgaste o la batería de cada robot. Una tercera opción, más débil, es una métrica que castigue el desequilibrio; PDDL no tiene operador de máximo, así que es incómoda de escribir, y el límite dice lo mismo de forma más sencilla.

Solución 9.4.3. Llega hasta el final y responde `solved (proven optimal), cost 159`. El objetivo de coste solo detiene antes la búsqueda cuando un plan lo cumple. No existe ningún plan de 100 metros, así que la búsqueda termina por sus propios motivos. Responder «irresoluble» sería un error: el problema tiene planes, y un objetivo de coste es un deseo sobre cuándo parar, no una parte del problema. (Para *exigir* un coste de como mucho 100, escribe un límite.)

Sección 9.5 — Métricas sobre cualquier fluente

Solución 9.5.1. `courier-handlings-domain.pddl` añade un contador, `(handlings)`, que `pick` y `drop` incrementan en 1, y el problema de prueba, `courier-handlings-p01.pddl`, tiene la métrica

```
(:metric minimize (+ (energy-used) (* 5 (handlings))))
```

El mejor plan cuesta 112: `r1` hace las dos entregas una detrás de otra, 92 unidades de energía y cuatro manipulaciones. El plan con la parada en el pasillo gasta 88 unidades y seis manipulaciones: 88 + 30 = 118. La parada ahorraba 4 unidades de energía y cuesta dos manipulaciones más, 10 unidades con este peso. Con un peso inferior a 2 por manipulación, la parada volvería.

Solución 9.5.2. Premia los planes que terminan con el robot tan cargado como sea posible: el robot coge todo lo que le cabe antes del final y no deja nada que el objetivo no le obligue a dejar. Hay un mejor plan, porque la carga está acotada por la capacidad, pero no es lo que nadie quiere. Probablemente el autor quería decir «llevar tanto como sea posible en cada viaje», es decir, pocos viajes. Eso es una suma que se acumula: un coste en cada desplazamiento, o un contador de desplazamientos, minimizado.

Sección 9.6 — Varios deseos en un solo número

Solución 9.6.1. «Los dos robots» es el punto (68, 128), y «`r1` solo», con la parada, es (88, 88). Sus costes son $68+128w$ y $88+88w$, iguales cuando $40w = 20$: $w = 0{,}5$. Por debajo gana el plan con los dos robots, y por encima, `r1` solo. Los ficheros `courier-quality-w04.pddl` y `courier-quality-w06.pddl` lo confirman: con $w = 0{,}4$ el óptimo es 119,2 (68 + 51,2), y con $w = 0{,}6$ es 140,8 (88 + 52,8). Exactamente en 0,5, los dos planes empatan y cualquiera de ellos es óptimo.

Solución 9.6.2. El coche pequeño quema 9,5 litros solo con el túnel y 8 con las dos autopistas, y paga 6 o 10 euros de peajes. Con el combustible a p euros, los costes son $6 + 9{,}5p$ y $10 + 8p$, iguales en $p = 8/3$, unos 2,67 euros. A 3 euros (`road-trip-p04.pddl`), el mejor plan es el de las dos autopistas, con un coste de 34 frente a 34,5.

Solución 9.6.3. Ninguna diferencia de energía entre dos planes de este problema puede pesar más que un solo metro, así que la energía nunca cambia qué cantidad de metros se elige. Solo decide entre planes con los mismos metros: es un criterio de desempate. Así se dice «ante todo pocos metros y, entre esos, poca energía», un orden de prioridades y no un compromiso entre medidas. El peso grande tiene que ser mayor que cualquier diferencia que pueda producir la segunda medida.

Sección 9.7 — Un límite o una métrica

Solución 9.7.1. `road-trip-tank-domain.pddl` añade una función estática `(tank ?c)` y, en las dos acciones de conducir, la precondición de que el combustible gastado después del tramo no lo supere:

```
(<= (+ (fuel-used)
       (/ (road-km ?from ?to) (km-per-liter ?c)))
    (tank ?c)))
```

Con 9 litros (`road-trip-tank-p01.pddl`) solo una ruta es lo bastante corta: las dos autopistas por Burgo, 160 km y 8 litros. El plan paga 10 euros de peajes, el más caro de todos, y es óptimo porque es el único plan. Con 7 litros (`road-trip-tank-p02.pddl`) no hay ningún plan, y el planificador lo demuestra tras 4 estados. La métrica quería no pagar peajes; decidió el límite.

Solución 9.7.2. (a) Un límite: una batería vacía es un fallo, no un resultado peor. Es la precondición de `move` del Capítulo 8. (b) Un límite si

5 kg es lo que soporta la bandeja; si es una recomendación, una métrica que penalice los viajes con sobrepeso. (c) Una métrica: menos tiempo es mejor, y ningún tiempo concreto es un fallo. Si la muestra se estropea al cabo de una hora, hay además un límite, y hacen falta los dos. (d) Una métrica, como dicen las palabras «cuando pueda»: un coste por entrar en la UCI. Un límite haría imposibles las entregas en la UCI.

Sección 9.8 — El método MODEL: qué hace mejor a un plan

Solución 9.8.1. Sin la métrica (`kitchen-p01-nometric.pddl`), Kairos devuelve el plan con el huevo de granja, la mantequilla y la miel, dice que su coste es 7 y que es óptimo demostrado: por longitud, lo es. Las dos codificaciones se diferencian en Fidelidad (la longitud no distingue una comida de 18 euros de una de 10) y un poco en Claridad (los precios y las líneas `increase`). El Tamaño es el mismo, porque los precios son estáticos; la Guía es más o menos la misma; y la Portabilidad apenas cambia, porque `:action-costs` lo aceptan los planificadores clásicos. Cuando un dominio tiene precios, no hay motivo para dejarlos fuera.

Solución 9.8.2. *Paso M.* Qué hace mejor a un plan: menos energía, en Wh, y menos riesgo, en la escala del operador de 0 a 5 por vuelo. No es un solo número hasta que el operador diga cuántos Wh vale un punto de riesgo; ese peso va en la ficha del modelo con el motivo que dé el operador. «No a cualquier riesgo» puede esconder también un límite (nunca un vuelo de riesgo 5), y hay que preguntarlo.
El dominio `drone-domain.pddl` tiene dos funciones estáticas por ruta, `flight-energy` y `flight-risk`, y dos contadores, `(energy)` y `(risk)`, que `fly` incrementa. Por encima de la plaza, el viaje cuesta 22 Wh y 8 puntos de riesgo; por el río, 55 Wh y 1 punto. Con un peso w para el riesgo, los costes son $22 + 8w$ y $55 + w$, iguales en $w = 33/7$, unos 4,7. En `drone-p01.pddl` el peso es 10 y el dron da el rodeo (coste 65 frente a 102); en `drone-p02.pddl` es 2 y el dron cruza la plaza (38 frente a 57).

C.10. Soluciones del Capítulo 10

Los ficheros de estas soluciones están en `examples/ch10/solutions`.

Sección 10.1 — Lo que una secuencia no puede decir

Solución 10.1.1. El robot `a` trabaja $5 + 3 + 8 + 2 = 18$ segundos y el robot `b`, $4 + 4 + 6 + 1 + 7 = 22$. Con una acción tras otra, el plan tarda $18 + 22 = 40$ segundos. Con cada robot por su cuenta, tarda lo que el robot más ocupado: 22 segundos. El cronograma tiene dos filas; la fila de `a` termina en 18 y está vacía desde ahí hasta 22.

Solución 10.1.2. (a) Basta con un coste: un solo agente, y lo que importa es una suma (kilómetros). (b) Tiempo: dos agentes trabajan a la vez y el éxito depende del instante de finalización, que una suma de duraciones no da. (c) Tiempo, y más de lo que ofrece este capítulo: los platos deben estar listos con poca diferencia entre unos y otros, lo que es coordinación entre acciones (Capítulo 11). (d) Un coste: la batería gastada es una suma sobre las acciones, sea cual sea su orden o su solapamiento.

Sección 10.2 — Una primera acción durativa

Solución 10.2.1. La predicción tentadora es tres segundos más, 55,006. El planificador devuelve 52,006, lo mismo que antes:

```
20.002: (pick r1 meds1 pharmacy) [5.000]
20.003: (move r1 pharmacy lobby) [10.000]
```

En este dominio, la recogida solo necesita al robot al inicio, así que el robot se va un milisegundo después y la recogida «se ejecuta» mientras anda. Termina en 25,002, mucho antes de que la descarga de 50,006 necesite el artículo. La recogida podría durar 29 segundos sin cambiar el makespan. Una duración que se puede cambiar sin ningún efecto sobre los planes es señal de que la acción no está ligada al agente que la realiza; el invariante de la Sección 10.4 la liga.

Solución 10.2.2. Cada aparato es un recurso propio, con su propio testigo (`toaster-ready`, `kettle-ready`), que se toma al inicio y se devuelve al final.

```
;; Breakfast: the toaster and the kettle work alone once started.
;; Times are in seconds.
(define (domain breakfast)
  (:requirements :strips :durative-actions)

  (:predicates
    (bread-in-toaster) (toaster-ready) (toast-done)
    (kettle-full) (kettle-ready) (water-hot))

  (:durative-action toast
    :parameters ()
    :duration (= ?duration 120)
    :condition (and (at start (bread-in-toaster))
                    (at start (toaster-ready)))
    :effect (and (at start (not (toaster-ready)))
                 (at start (not (bread-in-toaster)))
                 (at end (toast-done))
                 (at end (toaster-ready))))

  (:durative-action boil
    :parameters ()
    :duration (= ?duration 180)
    :condition (and (at start (kettle-full))
                    (at start (kettle-ready)))
    :effect (and (at start (not (kettle-ready)))
                 (at end (water-hot))
                 (at end (kettle-ready)))))
```

Listado C.2: Desayuno (`breakfast-domain.pddl`).

Las dos acciones no comparten nada, así que empiezan juntas y el makespan es la duración más larga:

```
$ kairos breakfast-domain.pddl breakfast-p01.pddl
; cost = 180
; makespan = 180.000
0.000: (toast) [120.000]
0.000: (boil) [180.000]
```

Solución 10.2.3. Los dos efectos no tienen especificador temporal. Las condiciones están bien. El comprobador informa de cada efecto; el fichero es `wrong-untimed-effect-domain.pddl`:

```
$ kairos check wrong-untimed-effect-domain.pddl
wrong-untimed-effect-domain.pddl:17:18: error: in durative
     action 'move': effects of durative actions must be under
     at start or at end (or be continuous) [semantic.
     untimed_effect]
```

```
 17 |     :effect (and (not (robot-at ?r ?from))
    |                  ^
...
invalid
```

La reparación no es mecánica: el borrado va `at start` y la adición `at end` (la Tabla 10.5 muestra qué hacen las otras opciones).

Sección 10.3 — Qué ocurre en cada instante

Solución 10.3.1. Seis acontecimientos:

1. 3,001: empieza `travel f1 f2` (borra `(lift-at e1 f1)`).
2. 6,001: termina `travel f1 f2` (añade `(lift-at e1 f2)`).
3. 6,002: empiezan las dos acciones `board`. Son un único acontecimiento: ninguna toca lo que la otra necesita (cada una borra la posición de su propio pasajero).
4. 8,002: termina `board ada` (añade `(boarded ada e1)`).
5. 10,002: termina `board bob` (añade `(boarded bob e1)`).
6. 10,003: empieza `travel f2 f3` (borra `(lift-at e1 f2)`).

Los acontecimientos 2 y 3 interfieren a través de `(lift-at e1 f2)`: el final del viaje lo añade y las subidas lo necesitan al inicio; de ahí 6,002 y no 6,001. Los acontecimientos 5 y 6 interfieren a través del mismo hecho: el final de la subida de Bob lo necesita (`at end`) y el inicio del viaje lo borra; de ahí 10,003. Los acontecimientos 3, 4 y 5 no interfieren entre sí: solo leen la posición del ascensor.

Solución 10.3.2. El makespan es 42,500: cinco separaciones, no seis.

```
$ kairos courier-time-domain.pddl courier-time-p01.pddl --
    epsilon 0.1
; cost = 42.50000000000001
; makespan = 42.500
...
```

Siete acciones tienen seis junturas, pero en una de ellas los dos acontecimientos no interfieren: el final de la recogida y el inicio del desplazamiento que la sigue comparten instante (Sección 10.4). En las otras cinco junturas, el final de un desplazamiento añade el hecho `robot-at` que la acción siguiente necesita al empezar.

Sección 10.4 — Condiciones en el tiempo

Solución 10.4.1. El robot no puede coger nada desde otra habitación: un instante después del inicio, el invariante lo necesita en la habitación. El plan de `courier-time-overall-domain.pddl` es el honrado, un milisegundo más corto, 42,004:

```
4.001: (move r1 lobby pharmacy) [6.000]
10.001: (pick r1 meds1 pharmacy) [2.000]
```

La recogida empieza en 10,001, el instante mismo en que llega el robot, cuando antes empezaba en 10,002. Sin condición al inicio, el inicio de la recogida no interfiere con el final del desplazamiento, y el intervalo de `over all` empieza justo después. No ocurre nada imposible, pero la recogida depende ahora de un hecho en el instante exacto en que se hace cierto, que es justo lo que la separación por ϵ existe para evitar. Mantén las dos líneas.

Solución 10.4.2. El dominio es `elevators-nowait-domain.pddl`. Si el ascensor solo se pide al inicio, se va mientras la gente todavía está entrando. El makespan baja de 28,008 a 22,008:

```
6.002: (board ada e1 f2) [2.000]
6.002: (board bob e1 f2) [4.000]
6.003: (travel e1 f2 f3) [3.000]
9.004: (board eve e1 f3) [2.000]
9.005: (travel e1 f3 f2) [3.000]
```

Bob empieza a subir en la segunda planta en 6,002. Un milisegundo después, el ascensor sale. Cuando termina su subida, en 10,002, el ascensor ya ha ido a la tercera planta y está bajando, y Bob está «dentro». A Eve le pasa lo mismo, y en la planta baja tampoco se pide al ascensor que espere a que salga nadie.

Sección 10.5 — Duraciones que dependen de la acción

Solución 10.5.1. Los cinco desplazamientos suman $4+6+6+10+12 = 38$ metros: 19 segundos a 2 metros por segundo. La recogida y la descarga siguen durando 2 segundos cada una, y cinco junturas necesitan ϵ: $19 + 4 + 0{,}005 = 23{,}005$. El planificador coincide (`courier-time-fast-p01.pddl`):

```
; makespan = 23.005
0.000: (move r1 dock lobby) [2.000]
2.001: (move r1 lobby pharmacy) [3.000]
```

Solución 10.5.2. Una acción para cada caso, distinguidas por una condición sobre la pinza. La condición es también un invariante: el robot no debe pasar de libre a cargado en mitad de un desplazamiento.

```
22  (:durative-action move
23    :parameters (?r - robot ?from ?to - location)
24    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
25    :condition (and (at start (robot-at ?r ?from))
26                    (at start (connected ?from ?to))
27                    (at start (free ?r))
28                    (over all (free ?r)))
29    :effect (and (at start (not (robot-at ?r ?from)))
30                 (at end (robot-at ?r ?to))))
31
32  (:durative-action move-loaded
33    :parameters (?r - robot ?i - item ?from ?to - location)
34    :duration (= ?duration (/ (distance ?from ?to)
35                              (loaded-speed ?r)))
36    :condition (and (at start (robot-at ?r ?from))
37                    (at start (connected ?from ?to))
38                    (at start (carrying ?r ?i))
39                    (over all (carrying ?r ?i)))
40    :effect (and (at start (not (robot-at ?r ?from)))
41                 (at end (robot-at ?r ?to))))
```

Listado C.3: Dos desplazamientos con velocidades distintas (de `courier-loaded-domain.pddl`).

`move-loaded` necesita el artículo como parámetro para nombrar lo que se lleva; con una sola pinza, no añade acciones instanciadas que importen. La entrega tarda 70,006 segundos: los tres desplazamientos con carga duran 12, 20 y 24.

```
10.002: (pick r1 meds1 pharmacy) [2.000]
12.003: (move-loaded r1 meds1 pharmacy lobby) [12.000]
```

Ahora hay seis separaciones. El desplazamiento con carga necesita `(carrying r1 meds1)`, que añade el final de la recogida, así que estos dos acontecimientos interfieren y el robot ya no sale en el instante en que termina la recogida.

Sección 10.6 — Trabajar en paralelo

Solución 10.6.1. No hay relevo (`courier-time-p06.pddl`): el mejor plan encontrado da una entrega a cada robot y tiene un makespan de 47,006. El relevo compensaba porque `r2` era lo bastante rápido para ir a por los medicamentos y entregarlos antes de que `r1` hubiera podido llevarlos; a la misma velocidad, el rodeo por la farmacia le cuesta a `r2` tanto como a `r1`. El makespan lo fija `r1`, cuya descarga en la UCI termina en 47,006; `r2` termina en 36,003.

Solución 10.6.2. `r2` está más lejos de la farmacia (16 metros frente a 10), pero es el doble de rápido, y hace toda la entrega en 26,005 segundos, frente a los 42,005 de `r1` (`courier-time-p07.pddl`):

```
; makespan = 26.005
0.000: (move r2 corridor lobby) [5.000]
5.001: (move r2 lobby pharmacy) [3.000]
8.002: (pick r2 meds1 pharmacy) [2.000]
```

`r1` no aparece en el plan. Un agente que no tiene nada útil que hacer no tiene líneas; no hay ninguna acción para esperar, ni hace falta.

Sección 10.7 — Una cosa cada vez

Solución 10.7.1. El plan honrado deja los medicamentos (2 segundos) y después desinfecta (20): 22 segundos, 22,001 con la separación. El dominio defectuoso planifica esto:

```
$ kairos courier-idle-bug-domain.pddl courier-idle-p02.pddl
; cost = 20
; makespan = 20.000
0.000: (drop r1 meds1 pharmacy) [2.000]
0.000: (disinfect r1 pharmacy) [20.000]
```

La descarga no pide el testigo, así que se ejecuta durante la desinfección. El dominio correcto da 22,001. La prueba ha encontrado el fallo porque su makespan honrado se conocía de antemano; un problema con un objetivo que solo necesitara una acción cada vez habría pasado.

Solución 10.7.2. El testigo (`counter-free ?l`) pertenece a la habitación. `pick` lo toma al inicio y lo devuelve al final, junto al testigo del robot:

```
(:durative-action pick
  :parameters (?r - robot ?i - item ?l - location)
  :duration (= ?duration (handling-time ?r))
  :condition (and (at start (idle ?r))
                  (at start (robot-at ?r ?l))
                  (over all (robot-at ?r ?l))
                  (at start (item-at ?i ?l))
                  (at start (free ?r))
                  (at start (counter-free ?l)))
  :effect (and (at start (not (idle ?r)))
               (at start (not (counter-free ?l)))
               (at start (not (item-at ?i ?l)))
               (at start (not (free ?r)))
               (at end (carrying ?r ?i))
               (at end (counter-free ?l))
               (at end (idle ?r))))
```

Listado C.4: Una recogida que ocupa el mostrador (de `courier-counter-domain.pddl`).

El problema incluye (`counter-free pharmacy`) en su estado inicial, y lo mismo para cada habitación con mostrador: una habitación sin ese hecho no permitiría ninguna recogida. Con el testigo, la

segunda recogida espera a la primera (makespan 12,003); sin él, los dos robots cogen en 0,000 (10,002):

```
0.000: (pick r2 meds2 pharmacy) [2.000]
2.001: (pick r1 meds1 pharmacy) [2.000]
2.001: (move r2 pharmacy lobby) [6.000]
```

Sección 10.8 — Otros dominios: una cocina y un ascensor

Solución 10.8.1. Con un fogón, la fila del fogón no tiene huecos y la de la cocinera tiene muchos: la cocinera no es el cuello de botella, y una segunda cocinera no cambia nada. El planificador lo confirma: `kitchen-p03.pddl` (dos cocineras) tiene un makespan de 40,004, como antes. Con dos fogones ninguna fila está llena, y el makespan, 28,003, es la cadena de la sopa detrás de la pasta: 1 + 6 + 20 + 1. Un tercer fogón (`kitchen-p04.pddl`) permite a la cocinera preparar primero la sopa y aun así cocinar lo demás en paralelo: 27,002, que es la sopa sola, 6 + 20 + 1. Ningún número de fogones bajará de ahí con una sola cocinera; la siguiente mejora necesita a alguien que prepare la pasta y la salsa mientras se prepara la sopa.

Solución 10.8.2. `e1` necesita 6 segundos para subir a la segunda planta, 4 para que suba Bob (Ada sube mientras tanto), 6 para bajar y 4 para que salga Bob: 20 segundos, más las separaciones. `e2` solo con Eve necesitaría 2 + 9 + 2 = 13. El plan terminaría hacia el 20, más tarde que los 19,005 de `e2` haciéndolo todo. `e2` pasa de todos modos por la segunda planta, y pararse allí solo le cuesta los 4 segundos de la subida de Bob.

Sección 10.9 — El método MODEL: tiempo

Solución 10.9.1. (a) Un coste. Un solo agente, y la magnitud que hay que minimizar es una suma (metros). La Fidelidad no pierde nada, y la Guía y la Portabilidad ganan: muchos planificadores pueden demostrar que el plan es óptimo. (b) Acciones durativas. Tres agentes trabajan en paralelo, el éxito es el instante de finalización, y el secado es una duración durante la cual nadie trabaja pero algo no debe ocurrir (el fogón de la cocina): un coste no puede expresar ninguna de las tres cosas. (c) Un coste. Varios agentes, pero la medida es el combustible total, una suma que no depende de qué se solape; el tiempo añadiría esfuerzo de búsqueda sin aportar nada. Si apareciera un plazo, la respuesta cambiaría.

Solución 10.9.2. El artículo está en la habitación dos segundos antes de que el robot lo haya soltado, así que otro puede cogerlo durante la descarga (el dominio es `courier-time-dropstart-domain.pddl`). En el relevo de `courier-time-p03`, el planificador lo aprovecha enseguida:

```
18.004: (drop r2 meds1 corridor) [2.000]
18.005: (pick r1 meds1 corridor) [2.000]
```

`r1` empieza a coger la caja un milisegundo después de que `r2` empiece a dejarla, y el makespan pasa de 41,007 a 40,007. Es el fallo de «ambos al inicio» de la Tabla 10.5: un producto disponible antes de haberse producido.

C.11. Soluciones del Capítulo 11

Los ficheros de estas soluciones están en `examples/ch11/solutions`.

Sección 11.1 — Planes que solo existen si las acciones se solapan

Solución 11.1.1. No hay plan. La reparación necesita `(lit ?m)` al inicio, y es el inicio de `light` el que añade ese hecho: los dos acontecimientos interfieren, así que la reparación empieza al menos ϵ después de encenderse la cerilla. Termina entonces al menos ϵ después de que la cerilla se apague, y su invariante se incumple durante ese último ϵ. Con `cellar-long-domain.pddl`:

```
$ kairos cellar-long-domain.pddl cellar-p01.pddl
note: engine M3 found no plan; as POPF, it keeps one partial
     plan per state, so this is strong evidence but not a
     proof [orchestrator.m3_exhausted]
...
kairos: no plan found (exhausted)
```

El código de salida es 14. Kairos solo responde 10 cuando tiene una demostración, y para esta tarea su demostración (olvidarse de las duraciones y de los tiempos) no sirve: sin duraciones, una reparación dentro de un encendido es posible. Lo que hace irresoluble el problema es la aritmética de los tiempos, que hemos hecho a mano.

Solución 11.1.2. (a) Inválido: en 0,000 la reparación empieza en el mismo instante que la luz, y su condición al inicio no ve un hecho que se está añadiendo en ese instante.

```
$ kairos validate cellar-domain.pddl cellar-p01.pddl cellar-
     p01-same.plan
cellar-p01-same.plan:4: error: step 2 (mend f1 m1) at 0.000:
     at start condition false: (lit m1) is false [validate.
     precondition_failed]
 4 | 0.000: (mend f1 m1) [2.000]
plan invalid
```

(b) Válido (`plan valid, makespan 5`): el invariante cubre el intervalo abierto de 3 a 5, y en el propio 5, cuando la luz se apaga, la reparación no pide nada. El final de la reparación y el final de la luz no interfieren. (c) Inválido: la reparación dura hasta 5,5, y el validador informa de `over-all condition broken after the happening at 5.000000`.

Solución 11.1.3. La lámpara es la cerilla y la foto es la reparación; el enfriamiento hace que la lámpara solo pueda usarse una vez, como una cerilla. La cámara es un testigo.

```
;; Exercise 11.1.3: a camera photographs plants, and each photo
;; needs a flash lamp on. A lamp stays on for 10 seconds and
;; then has to cool down for the rest of the session.
(define (domain greenhouse)
  (:requirements :strips :typing :durative-actions)

  (:types plant lamp)

  (:predicates
    (cold ?l - lamp)
    (on ?l - lamp)
    (camera-ready)
    (photographed ?p - plant))

```

```
  (:durative-action light-up
    :parameters (?l - lamp)
    :duration (= ?duration 10)
    :condition (at start (cold ?l))
    :effect (and (at start (not (cold ?l)))
                 (at start (on ?l))
                 (at end (not (on ?l)))))

  (:durative-action photograph
    :parameters (?p - plant ?l - lamp)
    :duration (= ?duration 4)
    :condition (and (at start (camera-ready))
                    (at start (on ?l))
                    (over all (on ?l)))
    :effect (and (at start (not (camera-ready)))
                 (at end (photographed ?p))
                 (at end (camera-ready)))))
```

Listado C.5: Un invernadero con lámparas de flash (`greenhouse-domain.pddl`).

Dos fotos caben en los 10 segundos de una lámpara (8 segundos, más las separaciones), y la tercera necesita la segunda lámpara:

```
$ kairos greenhouse-domain.pddl greenhouse-p01.pddl
; cost = 12.003
; makespan = 12.003
0.000: (light-up lamp2) [10.000]
0.001: (photograph pepper lamp2) [4.000]
2.003: (light-up lamp1) [10.000]
4.002: (photograph basil lamp2) [4.000]
8.003: (photograph tomato lamp1) [4.000]
```

La segunda lámpara se enciende en 2,003, no en 0 ni cuando se apaga la primera. La tercera foto puede empezar en 8,003, cuando la cámara queda libre, y termina en 12,003; 2,003 es el instante más temprano a partir del cual los 10 segundos de la lámpara la cubren. El makespan es el final de la última acción envolvente.

Sección 11.2 — La acción envolvente: una puerta sujeta

Solución 11.2.1. El plan que devuelve Kairos tiene un makespan de 62,006. `r1` sujeta la puerta desde el pasillo mientras `r2` entra y coge la muestra; después `r2` la sujeta desde dentro y `r1` *entra* en el laboratorio, para sujetarla a su vez desde dentro mientras `r2` sale:

```
10.001: (hold-door r1 lab-door corridor lab) [15.000]
12.001: (pass r2 lab-door corridor lab) [8.000]
20.002: (pick r2 sample1 lab) [2.000]
25.002: (hold-door r2 lab-door lab corridor) [15.000]
25.003: (pass r1 lab-door corridor lab) [8.000]
40.003: (hold-door r1 lab-door lab corridor) [15.000]
40.004: (pass r2 lab-door lab corridor) [8.000]
```

El plan es válido, y `r1` acaba encerrado en el laboratorio, cosa que el objetivo no prohíbe. Un plan más sencillo deja a `r1` en el pasillo y le hace sujetar la puerta dos veces seguidas; está en la carpeta del capítulo, en `courier-door-p03-hand.plan`:

```
10.001: (hold-door r1 lab-door corridor lab) [15.000]
12.001: (pass r2 lab-door corridor lab) [8.000]
20.002: (pick r2 sample1 lab) [2.000]
25.002: (hold-door r1 lab-door corridor lab) [15.000]
25.003: (pass r2 lab-door lab corridor) [8.000]
33.004: (move r2 corridor ward-a) [12.000]
45.005: (drop r2 sample1 ward-a) [2.000]
```

```
$ kairos validate courier-door-domain.pddl courier-door-p03.
     pddl \
    courier-door-p03-hand.plan
plan valid, makespan 47.005, metric 47.005
```

Quince segundos más corto. Un planificador temporal devuelve horarios válidos, no el más corto; en Kairos 1.0.6, los planes que repiten la misma acción envolvente con un contenido dentro de cada repetición están entre los que se le escapan a su búsqueda. Leer el plan y probar a mano uno mejor, con el validador como juez, forma parte del paso E.

Solución 11.2.2. Es resoluble, intercambiando los papeles. `r2` no puede pasar en 15 segundos, pero sí puede sujetar una puerta, y `r1` es lo bastante rápido para pasar (`courier-door-p05.pddl`):

```
$ kairos courier-door-domain.pddl courier-door-p05.pddl
; cost = 43.003
; makespan = 43.003
0.000: (move r1 lobby corridor) [10.000]
0.000: (pick r2 sample1 ward-a) [2.000]
2.001: (move r2 ward-a corridor) [24.000]
10.001: (hold-door r1 lab-door corridor lab) [15.000]
26.002: (drop r2 sample1 corridor) [2.000]
28.003: (pick r1 sample1 corridor) [2.000]
28.003: (hold-door r2 lab-door corridor lab) [15.000]
30.004: (pass r1 lab-door corridor lab) [8.000]
38.005: (drop r1 sample1 lab) [2.000]
```

La muestra cambia de manos en el pasillo y el robot lento se convierte en portero. (La sujeción de `r1` en 10,001 es inútil: nadie pasa durante ella. Un plan válido puede contener acciones que no aportan nada al objetivo.) El problema solo sería irresoluble si los dos robots fueran demasiado lentos.

Sección 11.3 — Temporalmente simple, temporalmente expresivo

Solución 11.3.1. `courier-time` y `kitchen`: no. Todo hecho se borra al inicio y se añade al final; ninguna acción proporciona algo solo mientras se ejecuta. Kairos dice que su motor M2 comprime las acciones durativas de ambos. `cellar`: sí, la acción envolvente es `light`. En `courier-door`, sí: es `hold-door`, y el registro nombra sus cuatro instancias (dos robots, dos lados). `courier-door-lock`: no; la puerta se abre al final de una acción y se cierra al inicio de otra, y el registro dice `engine M2: 66 durative actions compressed`. `courier-crate`: sí, `grip`, una vez por cada robot.

Solución 11.3.2. Con el dominio de la acción envolvente, los dos problemas dan el aviso `engine M2 forced on a task that may need actions to overlap` y terminan con `no plan found`: como un bloque, `hold-door` abre y cierra la puerta en un solo paso, y ningún `pass` la encuentra nunca abierta. Con el dominio del cerrojo, M2 resuelve los dos (mejor makespan 24,003, el mismo que con el motor por defecto): `open-door`, `pass` y `release-door` pueden ponerse en secuencia, y después el cálculo del horario deja que se solapen donde no dependen unas de otras. La codificación con cerrojo es temporalmente simple; esa es su portabilidad.

Sección 11.4 — Trabajar juntos: dos robots y un cajón

Solución 11.4.1. La elevación necesita los dos agarres al inicio y durante sus 4 segundos. Puede empezar ϵ después del segundo agarre, y debe terminar no más tarde que el primer agarre, en 8 (los dos finales pueden coincidir, porque el invariante cubre el intervalo abierto). Así que el segundo agarre empieza como muy tarde en $8 - 4 - 0{,}001 = 3{,}999$:

```
0.000: (grip r1 crate1 dock) [8.000]
3.999: (grip r2 crate1 dock) [8.000]
4.000: (raise crate1 r1 r2 dock) [4.000]
```

El validador responde `plan valid, makespan 11.999`. Con el agarre en 4,000 y la elevación en 4,001, el invariante de la elevación se incumple cuando termina el primer agarre, en 8,000, y el plan es inválido. La holgura del modelo son esos 3,999 segundos.

Solución 11.4.2. Tres parámetros necesitan tres desigualdades, una por cada pareja.

```
28  (:durative-action raise
29    :parameters (?c - crate ?r1 ?r2 ?r3 - robot ?l - location)
30    :duration (= ?duration 4)
31    :condition (and (at start (not (= ?r1 ?r2)))
32                    (at start (not (= ?r1 ?r3)))
33                    (at start (not (= ?r2 ?r3)))
34                    (at start (crate-at ?c ?l))
35                    (at start (gripping ?r1 ?c))
36                    (at start (gripping ?r2 ?c))
37                    (at start (gripping ?r3 ?c))
38                    (over all (gripping ?r1 ?c))
39                    (over all (gripping ?r2 ?c))
40                    (over all (gripping ?r3 ?c)))
41    :effect (and (at start (not (crate-at ?c ?l)))
42                 (at end (on-cart ?c)))))
```

Listado C.6: Una elevación que necesita tres robots (de `courier-crate3-domain.pddl`).

El plan tiene tres agarres en 0,000 y `(raise crate1 r1 r3 r2 dock)` en 0,001. Hay seis instancias de `raise`, una por cada orden de los tres robots, y todas significan lo mismo: con `-v`, Kairos cuenta 9 acciones durativas, 3 agarres y 6 elevaciones. La simetría crece como $n!$; para equipos mayores, cuenta los agarres con un fluente numérico (Capítulo 8) en lugar de nombrar a los robots.

Sección 11.5 — Duraciones que elige el planificador

Solución 11.5.1. De la base de carga a la sala A hay $4 + 10 + 12 = 26$ metros, 52 para el viaje de ida y vuelta, y 52 unidades de batería. El robot tiene 20 y le faltan 32: 16 segundos a 2 unidades por segundo. El plan dura $16 + 52 = 68$ segundos. Volver a cargar por el camino no serviría de nada: el único cargador está en la base, y el robot no pasa por ella hasta que el viaje ha terminado. Repartir los 16 segundos en varias cargas antes de salir tampoco cambia nada, salvo por las separaciones que añade.

Solución 11.5.2. El testigo de la bomba, pedido al inicio y durante todo el intervalo, mantiene el cambio de taza lejos de la bomba, y `pour` ya necesita su taza bajo la boquilla durante todo el intervalo.

```
34  ;; Not while the pump runs: pour needs the cup there over all.
35  (:durative-action swap
36    :parameters (?out ?in - cup)
37    :duration (= ?duration 3)
38    :condition (and (at start (under-spout ?out))
39                    (at start (waiting ?in))
```

```
                    (at start (pump-off))
                    (over all (pump-off)))
    :effect (and (at start (not (under-spout ?out)))
                 (at start (not (waiting ?in)))
                 (at end (under-spout ?in)))))
```

Listado C.7: Cambiar las tazas (de `coffee-tray-domain.pddl`).

```
$ kairos coffee-tray-domain.pddl coffee-p02.pddl
; cost = 8.602
; makespan = 8.602
0.000: (pour cup1) [1.600]
1.601: (swap cup1 cup2) [3.000]
4.602: (pour cup2) [4.000]
```

1,6 segundos para el expreso (40 ml) y 4 para el café largo (100 ml): en los dos casos, el instante en que se alcanza el límite inferior del intervalo. Ninguna es una cota de su acción.

Solución 11.5.3. El nuevo efecto es `(at end (increase (total-cost) (* ?duration (price ?l))))`, en `courier-charge-cost-domain.pddl`. Kairos devuelve el plan con la carga completa, con un coste de 120 céntimos, y dice por qué no es bueno:

```
$ kairos courier-charge-cost-domain.pddl courier-charge-cost-
    p01.pddl
; cost = 120
; makespan = 71.004
0.000: (charge r1 dock) [40.000]
...
note: the search minimizes the number of actions; the metric
    of the problem is reported for each plan but does not
    guide the search yet [orchestrator.metric]
```

Un coste que depende de una duración elegida no es un coste fijo de la acción, y la búsqueda no lo usa; encuentra el plan con menos acciones, que es el de una sola carga larga. El plan a mano es mucho más barato:

```
$ kairos validate courier-charge-cost-domain.pddl courier-
    charge-cost-p01.pddl \
    courier-charge-cost-p01-hand.plan
plan valid, makespan 36.504, metric 16.5
```

La lección es el último error típico del capítulo: un planificador no es un optimizador de números reales. Cuando el dinero depende de una duración, lee la nota, y ayuda con bloques o a mano.

Sección 11.6 — Cómo ordenan el tiempo los planificadores

Solución 11.6.1. Puntos temporales: el inicio y el final de cada agarre (g_1, g_1', g_2, g_2') y de la elevación (r, r'). Duraciones: $g_1' - g_1 = 8$, $g_2' - g_2 = 8$, $r' - r = 4$. Condiciones al inicio de la elevación, que necesita los dos agarres que añaden los inicios de los agarres: $r - g_1 \geq \epsilon$ y $r - g_2 \geq \epsilon$. Invariantes, que los finales de los agarres romperían: $g_1' - r' \geq 0$ y $g_2' - r' \geq 0$. Nada relaciona entre sí los dos agarres. Tiempos más tempranos: $g_1 = g_2 = 0$, $r = 0{,}001$, $r' = 4{,}001$, $g_1' = g_2' = 8$, como en el plan.

Solución 11.6.2. La flecha del inicio de `light` a su final pasa a ser $[1, 5]$. El camino que pasa por la reparación necesita $\epsilon + 2 + 0 = 2{,}001$ del inicio de la luz a su final, así que la cerilla debe arder al menos 2,001 segundos, y con ese valor el makespan es 2,001: el plan termina cuando el fusible queda reparado. La red encuentra este valor por

sí sola; una duración de la que solo depende el reloj es el caso fácil de una duración elegida.

Sección 11.7 — El método MODEL: coordinación

Solución 11.7.1. (a) Un cerrojo, o una acción envolvente con duración elegida: el operario decide cuándo bajar el puente y no hay una duración natural; con el cerrojo, el objetivo debe pedir el puente bajado. (b) Una acción envolvente de duración fija, 8 minutos: el paso termina por sí solo y nadie lo controla (en rigor, tampoco nadie lo empieza: el Capítulo 13 tiene la herramienta adecuada para ventanas que se abren en instantes conocidos). (c) Una acción envolvente para la sujeción, con una duración elegida hasta lo que aguante el brazo, y la soldadura dentro de ella: el cajón de los dos robots. (d) Ninguna coordinación; una duración fija, y no vale ni más corta ni más larga.

Solución 11.7.2. En `cellar-lock-domain.pddl`, `strike` añade `(lit ?m)` al final y `blow-out` lo borra. El problema que no tenía plan ahora lo tiene:

```
$ kairos cellar-lock-domain.pddl cellar-p03.pddl
; cost = 7.003
; makespan = 7.003
0.000: (strike m1) [1.000]
1.001: (mend f3 m1) [2.000]
3.002: (mend f2 m1) [2.000]
5.003: (mend f1 m1) [2.000]
```

Tres fusibles con una cerilla, que arde seis segundos y no se apaga nunca. El modelo ha perdido lo único que hacía del sótano un problema: la cerilla se acaba sola a los 5 segundos, le guste o no al plan. Un cerrojo da al agente el control sobre el final; cuando ese control lo tiene el mundo, solo la acción envolvente es fiel.

C.12. Soluciones del Capítulo 12

Los ficheros que se nombran aquí están en `examples/ch12/solutions`, y todos los ejecuta `tools/check_examples.sh ch12`.

Sección 12.1 — Una escalera donde el mundo tiene una rampa

Solución 12.1.1. (a) En el mundo, la batería en el segundo t del desplazamiento es $50 - t$: 40 cuando empieza la cámara y 30 cuando termina. Cruza el 35 en $t = 15$, así que la cámara tiene batería suficiente solo para la primera mitad de su trabajo. (b) El modelo que toma las 30 unidades al principio dice 20 durante todo el desplazamiento: la cámara no puede funcionar en absoluto. Se equivoca entre los segundos 10 y 15, pero del lado seguro. (c) El modelo que las toma al final dice 50 durante todo el desplazamiento: la cámara puede funcionar. Se equivoca entre los segundos 15 y 20 en la dirección peligrosa, porque acepta un plan que falla en el mundo.

Solución 12.1.2. Con una capacidad de 100 (`lamp-discrete-s01.pddl`), el modelo prudente tiene un plan:

```
$ kairos lamp-discrete-domain.pddl solutions/lamp-discrete-s01.
    pddl
; cost = 26.006
; makespan = 26.006
0.000: (charge r1 dock) [1.000]
1.001: (charge r1 dock) [1.000]
2.002: (charge r1 dock) [1.000]
3.003: (charge r1 dock) [1.000]
4.004: (charge r1 dock) [1.000]
5.005: (charge r1 dock) [1.000]
6.006: (sanitize r1) [20.000]
```

Seis cargas de un segundo llevan la batería de 10 a 40, y solo entonces puede empezar la lámpara. El *makespan* es de 26 segundos, donde el modelo continuo necesita 20. Esta vez el modelo prudente no ha perdido la existencia de un plan, sino su calidad: no puede solapar la carga y la lámpara, así que todos sus planes duran al menos 6 segundos más de lo necesario.

Sección 12.2 — Efectos continuos

Solución 12.2.1. El dominio es `tank-drain-domain.pddl`. La acción nueva es el reflejo de `fill`: su propio cerrojo (`valve-closed`), un `decrease` con `#t` y una cota superior que es el tiempo hasta vaciar el depósito:

```
(:durative-action drain
  :parameters (?t - tank)
  :duration (and
    (>= ?duration 1)
    (<= ?duration (/ (level ?t) (outflow ?t))))
  :condition (at start (valve-closed ?t))
  :effect (and
    (at start (not (valve-closed ?t)))
    (at end (valve-closed ?t))
    (decrease (level ?t) (* #t (outflow ?t))))))
```

Con 80 litros y 4 litros por segundo, la predicción es una acción de 20 segundos, y `kairos` devuelve `0.000: (drain t1) [20.000]` para `tank-drain-p01.pddl`. La cota calculada al principio es correcta aquí por la misma razón que en `fill`: el objetivo se puede alcanzar sin el grifo, así que nada más necesita cambiar el nivel mientras tanto. Si tuviera sentido tener abiertos a la vez el grifo y la válvula, la cota tendría que convertirse en una condición `over all`, como en el invernadero.

Solución 12.2.2. (a) Un efecto continuo correcto: el producto se puede escribir en cualquier orden. (b) Un error: `#t` bajo `at start`; la respuesta de `kairos check` es `#t can only appear in increase/decrease effects of durative actions and processes`. (c) Un error con el mismo mensaje: no existe un `assign` continuo. (d) Un efecto discreto correcto, el del Capítulo 11: toda la cantidad llega al final.

Solución 12.2.3. El planificador imprime dos planes. El primero es un llenado de 50 segundos, hasta tener el depósito lleno. El último tiene cuarenta llenados de un segundo, con un *makespan* de 40,039: cuarenta veces 2 litros son los 80 litros del objetivo, y cuarenta segundos son menos que cincuenta. Kairos prueba cada duración libre en sus cotas, aquí 1 segundo o el tiempo hasta llenar el depósito; la duración que elegiríamos nosotros, 40 segundos, no es ninguna de las dos, y ninguna condición de una acción la señala. La cadena de llenados mínimos es la forma que tiene el planificador de construirla. Si un grifo no se puede abrir y cerrar cada segundo, la

cota inferior de 1 segundo es una afirmación falsa sobre el mundo, y el arreglo corresponde al dominio: una cota inferior mayor (`(>= ?duration 10)`) o un grifo que solo se pueda usar una vez (borrar un hecho `(tap-unused ?t)` al principio y no restaurarlo nunca).

Sección 12.3 — Leer un número mientras cambia

Solución 12.3.1. Supongamos que el llenado empieza en s. Antes, el bancal bebe solo y el nivel es $5 - t$, así que el grifo debe abrirse antes de que el depósito se vacíe: $s \leq 5$. Durante el llenado, el nivel sube 1 litro por segundo. Un llenado de 30 segundos termina con $5 - s + 30$ litros, que caben en el depósito solo si $s \geq 5$: el único inicio válido es $s = 5$, el que encontró Kairos. Un llenado de 20 segundos termina con $25 - s$ litros, que siempre caben, y lo que queda dura hasta el final del riego: cualquier inicio de 0 a 5 es válido. Los planes `greenhouse-fill-at-0.plan` y `greenhouse-fill-at-5.plan` los acepta `kairos validate`; `greenhouse-fill-at-6.plan` se rechaza con `(>= (level ?t) 0): -1 >= 0 is false`.

Solución 12.3.2. El fichero es `greenhouse-reserve-domain.pddl`. Cambia un número en la acción `irrigate`:

```
     (over all (>= (level ?t) 2))
```

```
$ kairos solutions/greenhouse-reserve-domain.pddl solutions/
    greenhouse-reserve-p01.pddl
; cost = 40
; makespan = 40.000
0.000: (irrigate b1 t1) [40.000]
3.000: (fill t1) [28.000]
```

El grifo se abre en 3, cuando el nivel ha bajado de 5 a la reserva de 2, y sigue abierto 28 segundos, hasta que el depósito se llena en $t = 31$. Los últimos 9 segundos de riego dejan 21 litros. El plan sigue esperando al último momento; la reserva ha desplazado ese último momento.

Solución 12.3.3. Con la carga desde $t = 0$, la batería sube 3 unidades por segundo desde 10 y alcanza la capacidad de 30 a los 6,67 segundos. Una carga de 10 segundos seguiría hasta 40: no es un plan. El validador lo dice para `lamp-charge10.plan`: `(<= (battery ?r) (battery-capacity ?r)): 40 <= 30 is false`. Una carga de 6 segundos termina con 28 unidades, y la lámpara sola gasta exactamente 28 en sus 14 segundos restantes: un plan válido, el de `lamp-together.plan`, que termina con la batería vacía.

Sección 12.4 — El generador: cuándo empezar

Solución 12.4.1. Un vertido que empieza en s durante el funcionamiento termina con $90 - s + 10$ litros, como mucho 100 para todo $s \geq 0$. Solo, el generador vacía el depósito en $t = 90$. La ventana va de 0 a 90: con 90 litros, el bidón se puede verter desde el mismo principio del funcionamiento, y el depósito queda exactamente lleno cuando termina el vertido. Antes del funcionamiento no se puede verter: 110 litros no caben.

Solución 12.4.2. El primer vertido deja 85 litros, que duran hasta 95,001. El segundo bidón cabe en cuanto el depósito tiene sitio para su ganancia neta de 10 litros, y debe empezar antes de que el depósito se

vacíe; para terminar antes de que se pare el generador, en 110,001, debe empezar antes de 100,001. El plan `generator-p02-middle.plan` lo vierte en $t = 50$:

```
$ kairos validate generator-domain.pddl generator-p02.pddl
    solutions/generator-p02-middle.plan
plan valid, makespan 110.001
```

Solución 12.4.3. El fichero es `generator-safe-domain.pddl`. La norma es una condición más de la acción `refuel`:

```
    (over all (stopped ?g))
```

El hecho `(stopped ?g)` se borra cuando empieza el funcionamiento y no vuelve nunca, así que los bidones solo se pueden verter antes del funcionamiento. Con 60 litros (`generator-safe-p02.pddl`) caben los dos bidones, el depósito queda exactamente lleno y el plan son dos vertidos y después el funcionamiento, con un *makespan* de 120,002. Con 95 litros (`generator-safe-p01.pddl`) no cabe ni un bidón antes del funcionamiento, los 95 litros no duran 100 segundos y Kairos responde `no plan found`. La norma ha convertido un problema de sincronización en un problema sin solución, que es lo que a veces hace una norma de seguridad; el modelo hace bien en decirlo.

Sección 12.5 — El horno: cuándo parar

Solución 12.5.1. El horno mantiene su temperatura cuando la resistencia está apagada, así que el pan necesita una temperatura final entre 200 y 220: un calentamiento de entre 90 y 100 segundos. Si el pan puede entrar con la resistencia encendida, la condición sigue siendo la misma para la temperatura final, porque debe cumplirse hasta el final del horneado, pero el horneado puede empezar antes. En `oven-overlap.plan` la resistencia funciona 100 segundos y el pan entra en 90: durante el horneado la temperatura sube de 200 a 220, sin salir nunca del intervalo, y el plan es válido con un *makespan* de 100 segundos, una milésima mejor que el plan del capítulo. Un calentamiento de 101 segundos terminaría a 222 grados, y ningún inicio del horneado podría salvarlo.

Solución 12.5.2. El dominio es `oven-cool-domain.pddl`. La acción `cool` es `heat` con un `decrease`; retiene el cerrojo de la resistencia, para que calentar y enfriar no se solapen, y necesita el horno vacío:

```
(:durative-action cool
  :parameters ()
  :duration (and (>= ?duration 1) (<= ?duration 300))
  :condition (and
    (at start (heater-off))
    (at start (oven-empty)))
  :effect (and
    (at start (not (heater-off)))
    (at start (not (oven-empty)))
    (decrease (temperature) (* #t (cool-rate)))
    (at end (heater-off))
    (at end (oven-empty))))
```

```
$ kairos solutions/oven-cool-domain.pddl solutions/oven-cool-
    p01.pddl
; cost = 40.001000000000005
; makespan = 40.001
```

```
0.000: (cool) [30.000]
30.001: (bake meringue) [10.000]
```

A 1 grado por segundo, 30 segundos llevan el horno de 150 a 120, la primera temperatura a la que puede entrar el merengue. Otra vez, la duración es aquella en la que terminar la acción hace cierta una condición de otra acción.

Sección 12.6 — Lo que debe encontrar el planificador

Solución 12.6.1. Para el invernadero, la salida tiene una línea que empieza por `durations: (fill t1):` y termina con `planned as PDDL+`: solo pasos de tiempo. Para el mensajero tiene las líneas de la Sección 12.6, con `engine M2` y `also in steps of 0.1 s`: cambios discretos, con los pasos al lado. La diferencia está en quién lee el número. `irrigate` lee el nivel del depósito como lo bastante grande y `fill`, como lo bastante pequeño: ningún instante es seguro para un salto, porque mover el agua pronto puede desbordar el depósito y moverla tarde puede dejarlo seco. La batería del mensajero solo aparece en las condiciones como lo bastante grande, así que una descarga colocada al principio de un desplazamiento y una carga colocada al final nunca pueden esconder un incumplimiento.

Solución 12.6.2. Sin cota inferior (`courier-nobound-domain.pddl`), el plan es:

```
$ kairos solutions/courier-nobound-domain.pddl solutions/
    courier-nobound-p02.pddl
; cost = 58.007
; makespan = 58.007
0.000: (charge r1 dock) [16.000]
16.001: (move r1 dock lobby) [4.000]
...
```

Una carga de 16 segundos, hasta tener la batería llena, y un plan 11 segundos más largo que con la cota. La única cota que queda por probar es la superior. Como afirmación sobre el mundo, la cota inferior decía que no compensa ir a la base por menos de 5 segundos. Para la búsqueda era una segunda opción, una carga corta, y las cadenas de cargas cortas daban al planificador cantidades intermedias. Un modelo que quiera cargas cortas tiene que ofrecerlas: una duración libre solo es tan libre como la dejan sus cotas y las condiciones que la rodean.

Sección 12.7 — El mundo del mensajero

Solución 12.7.1. En `courier-margin-domain.pddl` la condición de `move` es `(over all (>= (battery ?r) 10))`. El plan de `p05` no cambia: sus valores más bajos eran 14, al llegar a la base, y 13, al final, ambos por encima de la reserva. El plan de `p02` cambia: una carga de 5 segundos dejaba al robot con 7 unidades en la sala A, por debajo de la reserva, así que el último plan tiene ahora dos cargas de 5 segundos antes de salir (de 20 a 70) y un *makespan* de 52,008 en lugar de 47,007. El robot llega con 32.

Solución 12.7.2. La batería pasa de 60 a 100 durante la carga (de 0 a 8 segundos). Después solo baja, una unidad por cada segundo de movimiento, con tramos planos en las dos descargas y las dos recogidas: 90 tras el camino a la farmacia, 57 al llegar a la UCI, 32

en el laboratorio y 10 en la base, en $t = 106{,}016$, que es el valor más bajo. La ronda necesita 90 unidades y el robot tenía 60, así que 30 unidades, 6 segundos de carga, eran suficientes: se podían ahorrar 2 segundos.

Solución 12.7.3. En `courier-handling-domain.pddl` las dos acciones reciben el efecto `(decrease (battery ?r) (* #t 0.5))` y la condición `(over all (>= (battery ?r) 0))`. La condición hace falta: la pregunta de la caja de método es si alguien lee la magnitud mientras se ejecuta la acción, y aquí la propia acción debe respetar un límite todo el tiempo. Sin ella, un robot con media unidad podría empezar una recogida y terminarla con una batería de −0,5. Como una recogida retiene `(idle ?r)`, nada más cambia la batería mientras tanto, y una versión discreta (pedir 1 unidad al principio y descontarla al principio) tendría los mismos planes. Para `p02` el plan es el mismo que antes, y el robot termina con 5 unidades en lugar de 7.

Sección 12.8 — El método MODEL: cambio continuo

Solución 12.8.1. (a) El combustible cambia durante el viaje, pero nadie lo lee mientras tanto y nada más lo cambia: un decremento discreto al principio, con una precondición para todo el viaje. (b) Lo mismo: la tinta se gasta durante la impresión y el cartucho solo se cambia entre trabajos. Discreto. (c) El calentador aporta calor mientras la ducha lo consume, y la ducha necesita el agua lo bastante caliente todo el tiempo: productor y consumidor se solapan, y un límite se cumple en mitad de la acción. Continuo. (d) Se solapan dos consumidores y ningún productor. Cada uno puede tomar toda su cantidad al principio: si la batería tiene suficiente para los dos totales, nunca se agota por el camino, y si no la tiene, uno de los dos no podría terminar de todos modos. Aquí un modelo discreto prudente es exacto. Deja de serlo el día en que el dron pueda cargar en vuelo, o cuando el objetivo solo necesite parte de una acción.

Solución 12.8.2. El dominio es `courier-discrete-domain.pddl`. El desplazamiento pide todo su consumo y lo descuenta al principio; la carga abona su cantidad al final:

```
(at start (>= (battery ?r)
              (* (/ (distance ?from ?to) (speed ?r))
                 (drain-rate ?r)))))
...
(at start (decrease (battery ?r)
            (* (/ (distance ?from ?to) (speed ?r))
               (drain-rate ?r))))
...
(at end (increase (battery ?r)
          (* ?duration (charge-rate ?r))))
```

En `p02` y `p05` da los planes del capítulo, con *makespans* de 47,007 y 51,008, y rechaza el plan tramposo en el mismo desplazamiento, ahora con una precondición: `8 >= 12 is false`. *Fidelidad*: igual, porque un robot que hace una sola cosa a la vez tiene una batería que nadie más toca ni lee durante una acción. *Tamaño*: igual. *Guía*: el modelo discreto es un problema temporal corriente; el continuo lo convierte en uno así Kairos. *Claridad*: el modelo continuo enuncia un ritmo y un límite; el discreto repite el producto de duración y ritmo en una condición y en un efecto, y hay que mantener iguales las dos copias a mano. *Portabilidad*: el modelo discreto necesita `:duration-inequalities` y `?duration` en un efecto, que admiten

más planificadores que `:continuous-effects`. A un planificador sin efectos continuos hay que darle la versión discreta, y para este mensajero no se pierde nada.

C.13. Soluciones del Capítulo 13

Los ficheros que se citan aquí están en `examples/ch13/solutions`, y todos los ejecuta `tools/check_examples.sh ch13`.

Sección 13.1 — Hechos que se siguen de otros hechos

Solución 13.1.1. (a) Consecuencia de `robot-at` y de `item-at`: no hay en la habitación ningún robot ni ningún artículo. (b) Básica: una puerta está abierta porque alguien la abrió. (c) Consecuencia de la carga del robot y de su capacidad; con los números del Capítulo 8 es una comparación, y no necesita ningún predicado. (d) Consecuencia de `robot-at`: hay una habitación en la que están los dos. (e) Básica: es un hecho `item-at`.

Solución 13.1.2. Con un artículo, `pick` siempre borra `free` y `drop` siempre lo añade. Con dos, lo que debe hacer una acción depende del estado: `pick` borra `free` solo si el robot ya lleva un artículo, y `drop` lo añade siempre, pero el hecho ya estaba si el robot llevaba uno. El efecto de `pick` pasa a depender de cuántos artículos lleva el robot, y los predicados no pueden contarlos sin más contabilidad (los huecos del Capítulo 7, o los números del Capítulo 8). El hecho sigue siendo una consecuencia sencilla de los hechos `carrying`, «no hay dos artículos distintos que lleve el robot», y una regla lo dice en una línea, mientras que los efectos necesitan un segundo mecanismo.

Sección 13.2 — Predicados derivados

Solución 13.2.1. El dominio es `courier-tidy-domain.pddl`: el dominio del mensajero con tipos, con `free` mantenido por efectos como siempre, más

```
(:derived (tidy ?l - location)
  (not (exists (?i - item) (item-at ?i ?l))))
```

Para `courier-tidy-p01.pddl`, con el objetivo `(and (tidy corridor) (tidy lobby) (free r1))`, Kairos devuelve un plan de 9 acciones que lleva cada artículo del pasillo a la sala B y lo deja allí. El objetivo dice dónde no deben estar los artículos y que el robot debe terminar con las manos vacías; no dice adónde van los artículos, y el planificador ha elegido una habitación vecina del pasillo que el objetivo no menciona. Sin `(free r1)` en el objetivo, el plan más barato dejaría un artículo en las manos del robot.

Solución 13.2.2. La comprobación responde `derived predicate 'free' cannot be initialized` y rechaza el problema. La redundancia solo es inofensiva mientras coincide con la regla. La misma línea en un problema en el que `r1` empieza llevando un artículo contradiría la regla, y el lenguaje tendría que decir cuál de las dos gana. Al prohibir la declaración, nunca tiene que decidirlo: un hecho derivado tiene exactamente una fuente de verdad.

Solución 13.2.3. Con una regla (`courier-keycard-domain.pddl`):

```
(:derived (may-enter ?r - robot ?l - location)
  (or (not (restricted ?l)) (has-keycard ?r)))
```

Con dos (`courier-keycard2-domain.pddl`):

```
(:derived (may-enter ?r - robot ?l - location)
  (not (restricted ?l)))

(:derived (may-enter ?r - robot ?l - location)
  (has-keycard ?r))
```

Las dos dan el mismo plan para el problema de prueba, en el que solo el robot con la tarjeta llega al laboratorio. Dos reglas se leen como una lista de motivos, y un tercer motivo (una escolta) es una tercera regla que no toca nada más; además, evitan `:disjunctive-preconditions`. Una sola regla mantiene la definición entera en un sitio, lo que ayuda cuando los motivos son pocos y no van a crecer.

Sección 13.3 — Reglas que se llaman a sí mismas

Solución 13.3.1. Ronda 1, un paso transitable desde la sala A: el pasillo. Ronda 2: el laboratorio y la sala B, y también la propia sala A (ir al pasillo y volver). Ronda 3: la UCI, a través de su puerta abierta. El vestíbulo está detrás de la puerta cerrada, y con él la farmacia y la base de carga: nunca se derivan. Así que `(reachable ward-a ward-a)` es cierto, como lo es para toda habitación que tenga una vecina transitable. Para `go` significa que `(go r1 ward-a ward-a)` es aplicable. Borra y añade el mismo hecho, no cambia nada y ningún plan gana nada con esa acción; una condición `(not (= ?from ?to))` eliminaría estas acciones instanciadas.

Solución 13.3.2. El dominio es `pipes-domain.pddl`. Su única regla une los dos pasos del circuito, el enlace y la propagación:

```
(:derived (wet ?p - point)
  (or (inlet ?p)
      (exists (?q - point ?v - valve)
        (and (wet ?q) (open ?v)
             (or (pipe ?v ?q ?p) (pipe ?v ?p ?q))))))
```

```
$ kairos solutions/pipes-domain.pddl solutions/pipes-p01.pddl
; cost = 4
(open-valve v3)
(open-valve v4)
(close-valve v1)
(close-valve v2)
```

El agua se envía a través del cobertizo, y se cierran *las dos* válvulas de la piscina: la que viene de la toma y la que va hacia el jardín, por la que el jardín llenaría la piscina por detrás. Es otra vez la lección de la cocina y el horno.

Solución 13.3.3. Tres bloques están en la misma torre exactamente cuando cada dos de ellos están ordenados por `above`, en un sentido o en el otro (`blocks-above-s01.pddl`):

```
(:goal (and (or (above a b) (above b a))
            (or (above b c) (above c b))
            (or (above a c) (above c a)))))
```

Partiendo de una mesa con `c` sobre `b`, y `a` y `d` sueltos, el plan es una sola acción, `(move-from-table a c)`. En cuanto a `p02`: en una

torre, `above` es un orden estricto, así que ningún estado tiene `a` por encima de `b` y `b` por encima de `a`. Kairos lo demuestra porque este es un problema clásico. Su espacio de estados es finito, la búsqueda lo recorre entero y ninguno de sus estados satisface el objetivo: eso es una demostración. En el Capítulo 12 los planes dependían de instantes con valores reales, la búsqueda solo podía tomar muestras de ellos, y quedarse sin muestras no demostraba nada.

Solución 13.3.4. Cada regla usa el otro predicado bajo una negación, así que `on` depende negativamente de `off`, que depende negativamente de `on`: cada uno depende negativamente de sí mismo. Las reglas no están estratificadas y no hay ningún orden en el que calcularlas. El procedimiento ronda a ronda muestra el problema. En la primera ronda no se ha derivado nada, así que «no apagada» y «no encendida» se cumplen las dos, y se derivan los dos hechos para cada lámpara; pero entonces cada hecho ha destruido el motivo del otro. Kairos rechaza el fichero `lamps-wrong-domain.pddl` con `derived predicate 'on' depends negatively on itself`, y lo mismo para `off`. Uno de los dos tiene que ser básico, y el otro, su negación.

Sección 13.4 — Qué hace un planificador con las reglas

Solución 13.4.1. En `courier-reach` hay 92 acciones y 518 axiomas; en `courier-steps`, 42 acciones y 392 axiomas. Las acciones difieren en 50: `go` se instancia para los 64 pares ordenados de habitaciones, `move` solo para los 14 conectados, y el resto de las acciones son las mismas. Las reglas difieren en 126, que es lo que el planificador construyó para la regla recursiva; las 392 que comparten los dos dominios son las de `passable`. El predicado recursivo añadió un tercio a las reglas y duplicó con creces las acciones, y a cambio consiguió planes más cortos y un vocabulario en el que el objetivo puede hablar de alcanzabilidad.

Solución 13.4.2. La regla recursiva tiene tres variables, así que hasta 100^3, un millón de reglas instanciadas, además de los 100^2 pares. Si el robot no abre ni cierra puertas, la alcanzabilidad no cambia nunca durante un plan: es una relación estática. Entonces puede calcularse una vez, fuera del planificador, y escribirse en el estado inicial como hechos básicos `(reachable a b)`, como mucho diez mil y normalmente muchos menos, sin ninguna regla. Si algunas puertas sí cambian, una vía intermedia es agrupar las habitaciones en zonas separadas por esas puertas y derivar la alcanzabilidad entre zonas, donde n es pequeño.

Sección 13.5 — Un mundo con calendario

Solución 13.5.1. El problema es `courier-hours-s01.pddl`. La farmacia está abierta en el instante cero, así que es un hecho inicial, seguido de tres literales temporizados:

```
(is-open pharmacy)
(at 8 (not (is-open pharmacy)))
(at 40 (is-open pharmacy))
(at 60 (not (is-open pharmacy)))
```

La farmacia está a 10 metros de la base de carga, así que el robot llega en el instante 10, dos segundos después de que se cerrara la primera ventana: no puede aprovecharla. Recoge en el 40, y el resto

dura 32 segundos, como en `p01`. El mejor plan que imprime Kairos tiene un *makespan* de 72,004.

Solución 13.5.2. Es válido: `courier-hours-wait.plan` empieza a moverse en el instante 20 y `kairos validate` lo acepta con un makespan de 62,006, dos milésimas peor que el plan del capítulo, porque las separaciones entre acciones consecutivas caen ahora después de la hora de apertura en lugar de antes. Según el makespan, los dos planes son iguales a efectos prácticos. Fuera del modelo hay razones en los dos sentidos. Esperar en la farmacia protege el plan frente a un trayecto lento: un robot que sale en el instante 20 y se retrasa pierde la diferencia. Esperar en la base de carga mantiene al robot en su cargador y no estorba a la gente en la puerta. Ninguna de las dos razones está en el modelo, así que el planificador no puede sopesarlas; un modelador al que le importen añade la batería, o una preferencia (Capítulo 14).

Sección 13.6 — Ventanas y plazos

Solución 13.6.1. El plan no tiene holgura: la recogida no puede empezar antes de 30, y cada acción sigue a la anterior. La entrega en el laboratorio termina en 86,008 y necesita el laboratorio abierto todo el tiempo, así que el laboratorio debe cerrar después de ese instante. Con segundos enteros, un laboratorio que cierra en 87 todavía tiene plan, y uno que cierra en 86, no. Para la farmacia, una ventana que se abre en T hace terminar la última entrega en $T + 56{,}008$, que debe ser anterior a 90: con segundos enteros, la apertura más tardía es 33, y Kairos devuelve entonces un plan con un makespan de 89,008.

Solución 13.6.2. Los ficheros son `rover-domain.pddl` y `rover-p01.pddl`. Las dos acciones tienen la condición `(over all (daylight))`, y el problema tiene el hecho `(daylight)` en el instante cero y tres literales temporizados, en 100, 150 y 250.

```
$ kairos solutions/rover-domain.pddl solutions/rover-p01.pddl
; cost = 210.001
; makespan = 210.001
0.000: (drive base dune) [30.000]
30.001: (photograph dune) [10.000]
40.002: (drive dune rock) [30.000]
70.003: (photograph rock) [10.000]
150.000: (drive rock dune) [30.000]
180.001: (drive dune base) [30.000]
```

Las fotografías se toman el primer día, antes de 80. Un desplazamiento de vuelta necesitaría luz hasta 110 y la noche cae en 100, así que el vehículo ni siquiera puede empezarlo: la condición es `over all`. Pasa la noche en la roca y vuelve el segundo día.

Solución 13.6.3. El plan es `satellite-volcano-first.plan`: observar el volcán en cuanto es visible, en el instante 12, girar, y observar el cráter en 21,002.

```
$ kairos validate satellite-domain.pddl satellite-p01.pddl
     solutions/satellite-volcano-first.plan
solutions/satellite-volcano-first.plan:4: error: step 3 (
     observe crater) at 21.002: over-all condition broken
     after the happening at 21.002000: (visible crater) is
     false [validate.invariant_failed]
 4 | 21.002: (observe crater) [4.000]
plan invalid
```

La ventana del cráter se cerró en 20. Después del volcán, la cámara no puede apuntar al cráter antes de 21, y la exposición tendría que empezar como muy tarde en 16.

Sección 13.7 — Números en el calendario

Solución 13.7.1. El problema es `courier-hours-s02.pddl`. Kairos imprime dos planes. En el primero (*makespan* 58,006) el robot no se detiene nunca: el movimiento de la farmacia al vestíbulo empieza en 12, en plena multitud, y dura 12 segundos, y el siguiente, de 10 metros, empieza en 24 y dura 20. En el segundo (54,003) el robot llega al vestíbulo en 24 y *espera* hasta 30,001, cuando la multitud se ha ido, para cruzar al pasillo en 10 segundos. Si saliera en 24, llegaría en 44; saliendo en 30, llega en 40. Un robot con prisa debería quedarse quieto seis segundos. Esto se debe a que las duraciones se fijan al inicio de cada movimiento, y un planificador lo encuentra donde la intuición no llega.

Solución 13.7.2. Por ejemplo: el ritmo de carga de la base, si el edificio limita la potencia en las horas de mayor demanda; el número de enfermeras de una sala, si una entrega necesita que haya una disponible; el precio de la electricidad, en una métrica; la velocidad de los ascensores, más lentos en los cambios de turno. Cada uno es un número que el planificador lee y que nadie cambia en el plan.

Sección 13.8 — El método MODEL

Solución 13.8.1. Podría (`courier-arrived-domain.pddl`, donde `deliver` vuelve a ser un simple `drop`):

```
(:derived (delivered ?i - item)
  (exists (?l - location)
    (and (destination ?i ?l) (item-at ?i ?l))))
```

El problema con horarios de apertura se resuelve con el mismo plan que antes. Se pierden dos cosas. Un hecho derivado no tiene memoria: `delivered` solo es cierto ahora mientras el artículo está en su destino, y se vuelve falso si un robot lo recoge de nuevo, mientras que el efecto de `deliver` registraba un suceso. Y el plazo se ha quedado sin sitio: era una condición `at end` de la acción que entregaba, y una regla no tiene ningún momento en el que consultar el calendario. Un artículo que llegara tarde alcanzaría el objetivo. Un plazo necesita o bien una acción que marque el suceso, como en el capítulo, o bien una restricción sobre el plan como `within`.

Solución 13.8.2. (a) El calendario: un literal temporizado que retira `(is-open cafeteria)`. (b) El estado: una regla sobre las posiciones de los robots. (c) El agente: un efecto de una acción. (d) El calendario. (e) El estado: una regla sobre lo que se ha entregado en la sala.

C.14. Soluciones del Capítulo 14

En `examples/ch14/solutions` están los ficheros PDDL de estas soluciones.

Sección 14.1 — Más deseos que batería

Solución 14.1.1. De la base de carga al vestíbulo, 4; del vestíbulo a la farmacia, 6: 10 unidades para llegar a `meds2`. De la farmacia a la UCI por el vestíbulo, el pasillo y la sala B: 6 + 10 + 12 + 5 = 33, 43 en total. Vuelta a la sala B a por la muestra: 5, 48 en total. De la sala B al laboratorio por el pasillo: 12 + 8 = 20, 68 en total, el valor de la figura.

Solución 14.1.2. A mano: `meds1`, después `meds2` y después `sample1` necesita 99 + 5 + 20 = 124. El orden `meds2`, `sample1`, `meds1` llega al laboratorio con 68, vuelve a la farmacia (8 + 10 + 6 = 24) y va a la sala A (6 + 10 + 12 = 28): 120. Kairos está de acuerdo: `courier-battery-full.pddl` (batería de 120) tiene plan, en ese orden, y `courier-battery-short.pddl` (batería de 119) es irresoluble. El orden es barato porque la muestra se recoge a la vuelta de la UCI: la sala B se cruza de todos modos, y el único viaje que se hace con las manos vacías es la vuelta a la farmacia. La respuesta «irresoluble» para 119 es además una información que ningún cálculo a mano da con facilidad: cubre todos los órdenes y todos los trucos, como dejar un artículo a mitad de camino y volver a por él.

Sección 14.2 — Una preferencia en el objetivo

Solución 14.2.1. El plan vacío deja el robot en la base de carga y viola las cuatro preferencias: coste 4. El mejor plan de `p02` termina en el laboratorio con 100 − 82 = 18 unidades, y el vestíbulo está exactamente a 18 metros (8 + 10): el robot puede hacer dos entregas *y* terminar en el vestíbulo, así que el mejor coste sigue siendo 1. `courier-battery-lobby.pddl` lo confirma: Kairos imprime planes de coste 4, 3, 2 y 1, y el último termina con `(move r1 corridor lobby)` con la batería vacía.

Sección 14.3 — Lo que vale cada deseo

Solución 14.3.1. La fila «`meds1`, `sample1`» solo viola `meds-icu`, así que ese peso tiene que ser menor que cada uno de los otros dos (los otros dos pares posibles violan `meds-a` o `sample-lab`). Por ejemplo, 3 para `meds-a`, 1 para `meds-icu` y 5 para `sample-lab`, como en `courier-battery-weights.pddl`: el mejor plan cuesta 1 y entrega la muestra y `meds1`.

Solución 14.3.2. Se trabaja de abajo arriba. Opcionales: peso 1; todas las entregas opcionales juntas valen 10. Normales: cualquier peso por encima de 10, por ejemplo 11; todas las normales y las opcionales juntas valen $4 \times 11 + 10 = 54$. Urgentes: cualquier peso por encima de 54, por ejemplo 55. Los pesos crecen deprisa: con muchas clases y muchos objetivos por clase, los números se hacen enormes, y ese es el límite práctico de las prioridades estrictas escritas como pesos.

Solución 14.3.3. Sea w el peso de `parked`. El mejor plan que no vuelve hace las dos entregas de medicamentos y cuesta $1 + w$. El mejor plan que vuelve atiende la UCI y el laboratorio y cuesta 3. El robot vuelve cuando $3 < 1 + w$, es decir, para $w > 2$; con $w = 2$ los dos planes empatan. Con $w = 1$ (`courier-battery-parked1.pddl`) el mejor plan cuesta 2 y termina en la UCI; con $w = 4$ (`p04`) cuesta 3 y termina en la base de carga.

Sección 14.4 — Deseos frente a esfuerzo

Solución 14.4.1. Entregar la comida después del medicamento suma $103 - 40 = 63$ a `total-cost`. La comida se entrega cuando su peso pasa de 63. Con 63 exactamente (`courier-cost-meal63.pddl`) los dos planes cuestan 103, y Kairos devuelve el que encuentra primero, la entrega única, porque la búsqueda *anytime* solo sustituye un plan por otro estrictamente mejor. Con 64 (`courier-cost-meal64.pddl`) imprime el plan con las dos entregas, de coste 103 frente a 104.

Sección 14.5 — Una preferencia, muchas instancias

Solución 14.5.1. Un medicamento: $45 + 70 = 115$. Los dos: $45 + 68 = 113$, sin ninguna violación. Nada: 140. El mejor plan entrega los dos medicamentos, con coste 113 y `(is-violated meds-icu)` igual a 0 (`courier-cost-meds70.pddl`).

Solución 14.5.2. `courier-cost-allmeds.pddl`:

```
(:goal (preference meds-icu
         (forall (?m - medicine) (item-at ?m icu))))
```

Ahora hay una sola preferencia, que se viola salvo que lleguen *los dos* medicamentos. Entregar uno cuesta 45 y sigue pagando los 50: 95. Entregar los dos cuesta 113. No hacer nada cuesta 50, y es el óptimo: Kairos solo imprime el plan vacío. Un plan que entrega un medicamento nunca puede ser óptimo, porque paga el esfuerzo y no obtiene nada a cambio. Con esta forma, el peso tiene que pasar de 113 para que el robot llegue a moverse.

Sección 14.6 — Deseos sobre el camino

Solución 14.6.1. Con peso 6 (`courier-lobby-w6.pddl`) la puerta trasera cuesta 42 y el vestíbulo $36 + 6 = 42$: un empate. Kairos imprime un plan, el de la puerta trasera, y se detiene: la otra ruta no es estrictamente mejor. Con peso 7 (`courier-lobby-w7.pddl`) el vestíbulo cuesta 43, y la puerta trasera es el único óptimo. Las dos salidas son las mismas cinco acciones con coste 42; la diferencia es que con 6 existe otro plan óptimo.

Solución 14.6.2. El dominio es `courier-night-domain.pddl`; su `move` es:

```
:precondition (and (robot-at ?r ?from) (connected ?from ?to)
                   (preference silence (not (quiet ?to))))
```

Además de `:preferences`, el dominio necesita también el requisito `:negative-preconditions`. Los hechos `(quiet ward-a)` y `(quiet ward-b)` están en el estado inicial de `courier-night-p01.pddl`, el objetivo es estricto y la métrica es `(+ (total-cost) (* 25 (is-violated silence)))`. El mejor plan cuesta 126: 74 metros, 2 manipulaciones y *dos* violaciones. A la UCI solo se llega por la sala B, así que el robot entra en la sala B a la ida y otra vez a la vuelta a la base de carga. La segunda entrada es el precio del objetivo estricto `(robot-at r1 dock)`; si ese objetivo fuera una preferencia que valiera menos de $31 + 25 = 56$, el robot se quedaría en la UCI.

Solución 14.6.3. El objetivo blando dice «el robot no debería estar en el vestíbulo cuando termine el plan». Habla de un estado, el último. Un robot que cruza el vestíbulo cinco veces con muestras y termina en el laboratorio lo cumple; un robot que nunca llevó una muestra y se detiene en el vestíbulo lo viola. La norma trata de cada momento

en que se hace un desplazamiento, y solo una preferencia en la precondición de `move` (o una restricción del capítulo siguiente) mira esos momentos.

Sección 14.7 — Otros dos mundos

Solución 14.7.1. `tourist-free.pddl`. El mejor plan cuesta ahora 2:

```
(bus hotel museum)
(visit museum)
(bus museum tower)
(visit tower)
(bus tower market)
(visit market)
(bus market cathedral)
(visit cathedral)
```

El turista coge cuatro autobuses (20 minutos), ve los cuatro lugares (220 minutos) y termina el día en la catedral sin tiempo para volver: se violan los deseos `on-foot` y `dinner`, a 1 cada uno. Una cena que vale 1, menos que el lugar más barato, se sacrifica por el mercado. Si ese plan parece absurdo, el peso estaba mal, o la cena no era un deseo para empezar: esta es la prueba del precio del paso E.

Solución 14.7.2. `satellite-long.pddl`. Kairos encuentra un plan de coste 1 con cinco imágenes en 76 segundos: `crops`, `flood`, `volcano`, `glacier` y `fire`. Solo se queda fuera `harbor`. Seis imágenes necesitan 60 segundos solo de toma de imágenes, lo que deja 20 para seis giros, y la búsqueda termina sin encontrar un plan así.

Sección 14.9 — El método MODEL: los deseos y su valor

Solución 14.9.1. *Fidelidad*: el objetivo estricto pierde soluciones (con él, nunca se pueden entregar los dos medicamentos); la preferencia es exacta si el peso es el adecuado; dos problemas seguidos son exactos para el primero y ciegos para el segundo: el primer plan puede dejar el robot varado en la UCI con una batería de 1, y entonces el segundo problema no tiene plan. *Tamaño*: el mismo para los tres, un átomo de objetivo o un par de acciones de recoger y renunciar. *Guía*: el objetivo estricto es el que mejor guía (cada estado se juzga por su distancia a la base de carga); la preferencia añade una cosa más que sopesar; la secuencia da dos búsquedas pequeñas. *Claridad*: el objetivo estricto es el más claro; la preferencia necesita que se explique su peso. *Portabilidad*: solo la preferencia necesita `:preferences`. Veredicto: objetivo estricto si un robot lejos de la base de carga es un fallo (no hay cargador en otro sitio, el turno de noche no puede ir a buscarlo); preferencia si es una comodidad; nunca la secuencia cuando los dos problemas comparten un recurso como la batería, porque el primer plan no sabe lo que necesitará el segundo.

Solución 14.9.2. *Éxito, obligatorio*: el informe está hecho. *Deseado*: ejercicios (peso 6, «debería»), gimnasio (4), lectura (3), lavadora (1) y no trabajar cansado (2 por cada tarea hecha con cansancio). Los pesos son una lectura razonable de «debería» y «me gustaría». El modelo es `evening-domain.pddl` con `evening-p01.pddl`: un fluente numérico `(minutes-left)`, una acción `work` cuyo efecto condicional deja al estudiante `(tired)` después de una tarea que cansa, una preferencia `fresh` en su precondición y una acción `rest`.

```
(:action work
```

```
    :parameters (?t - task)
    :precondition (and (not (done ?t))
                       (>= (minutes-left) (needs ?t))
                       (preference fresh (not (tired))))
    :effect (and (done ?t)
                 (when (tiring ?t) (tired))
                 (decrease (minutes-left) (needs ?t))))
```

Kairos devuelve, como mejor plan, uno de coste 3: informe, ejercicios, lavadora y el gimnasio *al final*. Las cuatro tareas ocupan exactamente 180 minutos, se renuncia a la lectura, y nadie trabaja cansado porque la tarea que cansa se ha puesto al final. Nadie le dijo al planificador que ordenara la tarde así: el orden se sigue de una preferencia sobre el camino.

C.15. Soluciones del Capítulo 15

Los ficheros PDDL de las soluciones están en `examples/ch15/solutions`.

Sección 15.1 — Una regla que el plan incumple

Solución 15.1.1. (a) Un objetivo: trata del último estado. (b) Una precondición de `pick`, con un contador o con huecos (Capítulo 7, Capítulo 8): solo `pick` añade un artículo, así que basta una acción para vigilar la regla. También es un `always`, pero aquí lo natural es la precondición. (c) El plan completo: `always`; un objetivo permitiría dejar la muestra allí un rato. (d) Una precondición: es lo que la acción necesita para ser posible. (e) El plan completo: `sometime`. (f) El plan completo: `(sometime-before (robot-at r1 ward-b) (robot-at r1 ward-a))`.

Sección 15.2 — Los estados de un plan: always

Solución 15.2.1. `courier-rules-apart.pddl`:

```
(:constraints
  (forall (?r - robot ?w - waste ?m - medicine)
    (always (not (and (carrying ?r ?w) (carrying ?r ?m)))))))
```

El plan sigue teniendo 16 acciones, pero otra forma. Con la regla de la UCI, el robot cargaba los dos artículos en la farmacia y descargaba primero los residuos. Ahora los dos no pueden compartir la cesta en ningún momento, así que lleva los medicamentos solos a la UCI, vuelve a la farmacia a por los residuos y termina en la base de carga: el plan de `courier-rules-p02-ok.plan`. Ese plan cumple las dos reglas; el plan impreso para `p02` solo cumple la regla de la UCI.

Solución 15.2.2. `unsolvable`. El robot está en la base de carga en S_0, el estado inicial pertenece a la trayectoria, y `always` debe cumplirse en todos los estados: la restricción es falsa antes de cualquier acción. (El objetivo `(robot-at r1 dock)` también la contradice, en el otro extremo).

Sección 15.3 — Lo que debe ocurrir, y en qué orden

Solución 15.3.1. `courier-rules-no-return.pddl` es irresoluble. La UCI cuelga de la sala B y de nada más, así que entrar en la UCI y volver a salir pone al robot dos veces en la sala B, con la UCI en medio:

dos tramos. Lo mismo pasa con la base de carga, donde el robot empieza y donde deben acabar los residuos. En un mapa con forma de árbol, todo viaje de ida y vuelta repite salas.

Solución 15.3.2. (a) El robot no está nunca en el laboratorio: el primer estado en el que estuviera allí necesitaría uno *anterior* en el que ya estuviera. Es una forma rebuscada de escribir `(always (not (robot-at r1 lab)))`. (b) Nada: cuando el robot está en la UCI, el mismo estado proporciona el «después», ya que $j \geq i$ permite $j = i$. La restricción se cumple siempre. (c) Tampoco nada: S_0 tiene `(sanitized r1)` y es anterior a cualquier estado en el que el robot lleve algo, puesto que en S_0 no lleva nada. La regla parece «desinfectarse antes de tocar los medicamentos» y no pide nada.

Solución 15.3.3. El problema es `courier-rules-clean.pddl`. El requisito es `:disjunctive-preconditions`, porque un `imply` es una disyunción. El plan que se desinfecta antes de tiempo se rechaza: en sus dos últimos estados el robot está en el laboratorio y la pinza está sucia desde que cogió la muestra. El plan nuevo coge la muestra *primero* y se desinfecta a la vuelta, con la muestra en la cesta:

```
(move r1 dock lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(pick r1 sample1 ward-a)
(move r1 ward-a corridor)
(sanitize r1 corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

Misma longitud, mismas acciones, otro orden, y solo este orden es lo que quería el laboratorio. En este dominio un `drop` no ensucia la pinza, así que el robot sigue limpio mientras está en el laboratorio.

Sección 15.4 — Reglas con reloj

Solución 15.4.1. `courier-rules-within6.pddl` es irresoluble. Desde la base de carga, la entrega necesita dos movimientos hasta la farmacia, coger los medicamentos, tres movimientos hasta la sala A y dejarlos: 7 acciones, así que los medicamentos están en la sala A como pronto en S_7, y $t_7 = 7 > 6$.

Solución 15.4.2. La entrega necesita $4+6+2+6+10+12+2 = 42$ segundos de acciones. Con un plazo de 43 (`courier-timed-within43.pddl`) Kairos encuentra un plan que va directo a por los medicamentos y los deja en el instante 42,006. Las seis milésimas son las separaciones entre una acción del robot y la siguiente: cada acción empieza 0,001 después de que termine la anterior, porque el final de una y el inicio de la siguiente tocan los mismos hechos (Capítulo 10). Así que los medicamentos no pueden estar en la sala A a los 42 segundos justos, y con un plazo de 42 la búsqueda de Kairos sigue hasta agotar su límite de tiempo sin encontrar un plan.

Solución 15.4.3. La primera es un único plazo, contado desde el inicio del plan: la pinza está limpia en algún estado de los cinco primeros pasos. La segunda es un plazo que vuelve a empezar en cada estado en el que el robot está en la UCI: como mucho cinco pasos después, la pinza está limpia. Un plan que se desinfecta en el paso 2, visita la UCI en el paso 9 y no vuelve a desinfectarse cumple la primera y viola la segunda. Un plan que nunca entra en la UCI y nunca se desinfecta cumple la segunda y viola la primera.

Sección 15.5 — Los diez operadores, uno junto a otro

Solución 15.5.1. El robot está en: base de carga (S_0), vestíbulo, farmacia (S_2, S_3), vestíbulo, pasillo, sala B, UCI (S_7, S_8), sala B, pasillo, vestíbulo, farmacia (S_{12}, S_{13}), vestíbulo, base de carga (S_{15}, S_{16}). Lleva `meds1` de S_3 a S_7 y `waste1` de S_{13} a S_{15}. (a) Se viola: el robot nunca está en el laboratorio. (b) Se viola: dos tramos en la farmacia. (c) Se cumple: el robot lleva los medicamentos desde S_3, antes de estar en la UCI en S_7. (d) Se cumple: los medicamentos están en la UCI en S_8, y $t_8 = 8$. (e) Se cumple: el último estado en que se llevan los residuos está a 1 paso de S_{16}, y el primero a 3. (f) Se cumple: después del instante 12 el robot está en la farmacia, el vestíbulo y la base de carga. Los ficheros `courier-rules-quiz-a.pddl` y `-quiz-b.pddl` contienen (a) y (b), y `kairos validate` rechaza el plan con cada uno; `courier-rules-quiz-true.pddl` contiene las otras cuatro juntas, y el plan es válido para él.

Sección 15.6 — Dónde viven las reglas, y reglas que ceden

Solución 15.6.1. «Un ciclo de calentamiento por turno» es una política energética que depende del día: un turno con un encargo urgente puede permitir dos ciclos, y un problema con más piezas de las que caben en el horno no tendría plan. Las políticas que cambian van en el problema, quizá como preferencias. «Nunca la puerta abierta con el horno caliente» protege a las personas: no depende de la orden de trabajo, nadie debería poder olvidarla al escribir un problema y nunca debería ser blanda.

Solución 15.6.2. `courier-rules-p11-waste.pddl`. Los dos deseos se siguen excluyendo, así que el planificador viola el más barato, que ahora es `fast-meds`, con peso 3. El mejor plan lleva primero los residuos a la base de carga (6 pasos), después los medicamentos a la UCI, y cuesta 3. La regla estricta de la UCI nunca entró en la negociación.

Sección 15.7 — Dos mundos más

Solución 15.7.1. `rover-two-rules.pddl`: cinco acciones, el plan de la sección sin su último `drive`. El rodeo por la cresta se sigue debiendo a `sometime-before`; sin reglas, el róver iría por la llanura. La regla `at-most-once` aquí no cuesta nada, porque el plan corto ya entra en el cráter una sola vez: una restricción que el mejor plan ya cumple sale gratis.

Solución 15.7.2. La cota mínima es 7 (`workshop-fresh7.pddl` tiene plan, `workshop-fresh6.pddl` no). En el plan de 11 acciones las piezas se cargan en los pasos 2, 3 y 4 y se descargan en los pasos 9, 10 y 11. El plazo de cada pieza corre desde el primer estado en el que está en el horno, así que la pieza cargada en el paso 2 debe estar fuera en el paso 9, la siguiente en el 10 y la última en el 11: las piezas deben salir *en el orden en que entraron*. El plan de Kairos hace exactamente eso (`lid`, `gear`, `bracket`, las dos veces). Con 6, la primera pieza tendría que estar fuera en el paso 8, cuando la puerta se acaba de abrir.

Sección 15.9 — El método MODEL: reglas sobre el plan completo

Solución 15.9.1. `rover-auto-domain.pddl` con `rover-auto-p01.pddl`. Dos predicados estáticos marcan los puntos de paso, `(risky ?w)` y `(lookout ?w)`, y un hecho hace de memoria del autómata, `(looked)`:

```
(:action drive
  :parameters (?from ?to - waypoint)
  :precondition (and (at ?from) (path ?from ?to)
                     (or (not (risky ?to)) (looked)))
  :effect (and (not (at ?from)) (at ?to)
               (when (lookout ?to) (looked))))
```

El plan son las mismas cinco acciones que con las dos primeras reglas de la versión con restricciones. Si el róver empieza en la cresta, esta codificación *no* lo sabe: `(looked)` solo se activa al ir *a* un mirador, así que el problema debe poner `(looked)` a mano en su estado inicial. La restricción no necesita ese cuidado, porque S_0 forma parte de la trayectoria. Ese es el punto débil de todo autómata hecho a mano: su estado inicial es una cosa más que hay que acertar, y nada lo comprueba.

Solución 15.9.2. *Lavarse las manos después de la carne cruda, antes de cualquier otra cosa*: una regla de la cocina (dominio), estricta. No basta un único operador: «antes del *siguiente* contacto» trata de lo que pasa entre medias. Un hecho `(hands-dirty)`, que se activa al tocar carne y se borra al lavarse, y `(always (imply (touching ?x) (or (raw-meat ?x) (not (hands-dirty)))))`, o simplemente una precondición de las acciones de tocar. *Horno nunca encendido sin nadie dentro*: dominio, estricta, un solo estado: `(always (imply (oven-on) (cook-in-kitchen)))`. *Probar antes de servir*: dominio, estricta, historia: `(forall (?d - dish) (sometime-before (served ?d) (tasted ?d)))`; aquí «en algún momento anterior» es justo lo que se quiere decir. *Abrir la nevera el menor número de veces posible*: un deseo, no una regla. `at-most-once` como preferencia solo pone precio a la diferencia entre una apertura y varias; para contar aperturas, una preferencia en la precondición de la acción de abrir (Capítulo 14) o un coste de acción. *Servido antes de las 14:00*: de hoy (problema), estricta o blanda según los comensales: `(forall (?d - dish) (within 840 (served ?d)))` con el tiempo en minutos desde medianoche, en un modelo temporal.

C.16. Soluciones del Capítulo 16

Los ficheros PDDL de estas soluciones se encuentran en el directorio `examples/ch16/solutions`.

Sección 16.1 — Un robot en dos habitaciones

Solución 16.1.1. *Un robot está en una habitación*: `move` añade `(robot-at ?r ?to)` y borra `(robot-at ?r ?from)`; ninguna otra acción toca `robot-at`. *Un artículo está en una habitación o con un robot*: `pick` borra `(item-at ?i ?l)` y añade `(carrying ?r ?i)`; `drop` hace lo contrario. *Libre exactamente cuando no lleva nada*: `pick` exige y borra `(free ?r)` a la vez que añade `carrying`; `drop` lo vuelve a añadir a la vez que borra `carrying`. La precondición `(free ?r)` de `pick` también forma parte del invariante: sin ella, un robot llevaría dos artículos y

quedaría libre tras dejar uno. Los tres dependen además del estado inicial, que tiene que cumplirlos de partida.

Solución 16.1.2. Sin ese borrado, coger un artículo deja una copia suya en la habitación: tras la acción de coger se cumplen a la vez `(item-at meds1 pharmacy)` y `(carrying r1 meds1)`. Cualquier objetivo que necesite un mismo artículo en dos sitios lo pone al descubierto: `(and (item-at meds1 pharmacy) (item-at meds1 ward-a))` tiene un plan de siete acciones (`courier-copy-domain.pddl` con `courier-copy-p01.pddl`). Y también uno más discreto: un objetivo que pide los medicamentos en la sala A en un problema en el que una regla (por ejemplo, una restricción del Capítulo 15) quiere que la farmacia nunca se quede sin ellos.

Sección 16.2 — Una función cuyo valor es un objeto

Solución 16.2.1. `courier-of-hands.pddl`, con el objetivo

```
(:goal (and (= (holder sample1) r2)
            (= (location-of r2) lab))))
```

Va a buscarla `r2`: dos movimientos hasta la sala A, una acción de coger, dos movimientos de vuelta, cinco acciones y ningún `drop`. No hay ninguna acción que pase un artículo de un robot a otro, así que el robot que nombra el objetivo tiene que coger el artículo él mismo, y `r1` no sirve de nada. Con predicados, el mismo objetivo sería `(carrying r2 sample1)`; el fluente no añade nada aquí, salvo que «en el laboratorio» y «con `r2`» son a la vista dos valores de una misma cosa.

Solución 16.2.2. `courier-nofree-domain.pddl`. `pick` comprueba que el robot no tiene nada, y ninguna de las dos acciones tiene un efecto sobre `free`:

```
:precondition (and (= (location-of ?r) ?l) (= (holder ?i) ?l)
                   (not (exists (?j - item)
                          (= (holder ?j) ?r))))
:effect (assign (holder ?i) ?r))
```

La condición es segura porque `holder` es total: todo artículo tiene poseedor en el estado inicial y ningún efecto lo deja indefinido, así que la comparación dentro del `exists` nunca se encuentra con un fluente sin valor. El tercer invariante del Ejercicio 16.1.1 ha desaparecido: no hay ningún `free` que pueda desacompasarse. En `courier-nofree-p01.pddl`, el robot traslada dos artículos de uno en uno, en nueve acciones.

Sección 16.3 — Fluentes como términos

Solución 16.3.1. `courier-of`: `move` tiene una instancia por cada robot y conexión, 14; `pick` y `drop`, una por cada robot, artículo y habitación, 8 cada una: $14+8+8 = 30$. Sus demás precondiciones son dinámicas y no descartan nada. `courier-terms`: `pick` no tiene habitación, 1 instancia; `drop` conserva su habitación, 8; y $23 - 9$ deja 14 para `move`. La acción sin instanciar solo tiene 8 instancias (una por destino), pero su precondición contiene el fluente como argumento de un predicado estático, y el planificador lo sustituye por cada habitación que puede representar y se queda con los pares conectados: los mismos 14 de antes. El parámetro se quitó del modelo, no del trabajo del planificador; el ahorro real está en `pick`.

Solución 16.3.2. `courier-apart-domain.pddl`. `move` gana una precondición universal sobre los demás robots:

```
                    ; no other robot is in ?to
                    (forall (?o - robot)
                      (or (= ?o ?r)
                          (not (= (location-of ?o) ?to)))))
```

En `courier-apart-p01.pddl`, el plan tiene siete acciones y todas las hace `r2`: el robot que estorbaba va a la farmacia y hace la entrega él mismo. El objetivo pedía los medicamentos en la sala A, no que los llevara `r1`. Para ver la regla en acción, dale a `r2` algo que lo mantenga ocupado, o pide una segunda entrega.

Sección 16.4 — Sin valor: indefinido

Solución 16.4.1. No. Mientras el robot lleva la muestra, `(room-of sample1)` está indefinido, y un fluente indefinido no es igual a ninguna habitación: el validador informa de `(room-of sample1) is undefined` para el plan que se detiene antes de dejarla. Solo `drop` vuelve a dar valor al fluente. Para pedir que el robot termine con las manos vacías, el objetivo necesita algo que pueda comprobar: `(free r1)`, el predicado que el dominio conservó precisamente para eso.

Solución 16.4.2. Con `holder`, dónde está un artículo y quién lo lleva son una sola variable; no queda nada en manos de las acciones. `free` sigue siendo un invariante (o desaparece, como en el Ejercicio 16.2.2). Con `room-of` y `carrying`, las acciones tienen que mantener «`room-of` está indefinido exactamente cuando algún robot lleva el artículo»: `pick` y `drop` cambian las dos cosas, y un `(assign (room-of ?i) undefined)` olvidado dejaría el artículo en la habitación y en la pinza, otra vez el error del Ejercicio 16.1.2. En Fidelidad y en Claridad gana el fluente total. El parcial tiene una ventaja: su tipo de valor es simplemente `location`, sin el supertipo `place`.

Sección 16.5 — Dos mundos más

Solución 16.5.1. `blocks-of-p02.pddl`. El mejor plan tiene cuatro acciones:

```
(unstack d)
(stack c d)
(unstack b)
(stack a b)
```

En nuestra ejecución, el primer plan que imprimió Kairos tenía seis: dejó un bloque sobre la otra torre y tuvo que deshacerlo. Sin métrica, la búsqueda devuelve el primer plan al que llega y después lo mejora; la salida *anytime* termina con el plan de cuatro acciones, que es óptimo (cada torre necesita su bloque de arriba en la mesa y su bloque de abajo encima de él).

Solución 16.5.2. `coloring-australia.pddl`, con 18 hechos `border` (cada frontera en los dos sentidos). Kairos pinta las siete regiones en siete acciones: Australia Meridional recibe un color para ella sola, y el anillo que la rodea alterna los otros dos. Tasmania no tiene fronteras, así que la precondición de `paint` es vacía para ella y cualquier color sirve; el plan la pinta porque el objetivo pide que todas las regiones tengan uno.

Sección 16.6 — Variables de estado: lo que ve el planificador

Solución 16.6.1. `courier-p02`: 18 hechos, 3 variables, 2 grupos mutex. Los grupos son los ocho hechos `robot-at`, y los ocho hechos `item-at` junto con `(carrying r1 meds1)`; la tercera variable es `(free r1)`. `courier-twin-p02`: 18 hechos, 10 variables, 1 grupo mutex. El grupo que sobrevive es el del artículo: `pick` y `drop` no se tocaron y siguen cambiando un hecho por otro. El grupo del robot se pierde, porque `move` añade ahora una posición sin quitar otra, y los ocho hechos `robot-at` son ocho variables independientes: $1 + 8 + 1 = 10$.

Sección 16.7 — El método MODEL: un valor o muchos

Solución 16.7.1. Planta de un ascensor: fluente de objeto (siempre una). Personas dentro: predicado (muchas). Estado de una puerta: fluente de objeto con tres valores constantes. Habitaciones que une una puerta: predicado, estático (dos habitaciones; una relación). Nivel de batería: fluente numérico. Carro que empuja un robot: fluente de objeto que puede estar indefinido (como mucho uno). Artículos de un carro: predicado (muchos); visto desde el artículo, «el carro o la habitación en que está» es un fluente de objeto, como `holder`. Robot en el único cargador: fluente de objeto del cargador que puede estar indefinido, o un predicado `(charging ?r)` con el invariante «como mucho un robot».

Solución 16.7.2. `door-of-domain.pddl` y `door-of-p01.pddl`. Los estados son constantes de un tipo `door-state`:

```
(:constants open closed locked - door-state)
...
(:functions
  (state-of ?d - door) - door-state)

(:action unlock
  :parameters (?d - door ?k - key)
  :precondition (and (= (state-of ?d) locked)
                     (opens ?k ?d) (have ?k))
  :effect (assign (state-of ?d) closed))
```

El plan para una puerta cerrada con llave y otra solo cerrada tiene tres acciones: abrir la cerradura, abrir, abrir. Con los predicados `open` y `locked`, una puerta podía estar abierta *y* cerrada con llave, un estado que las acciones tenían que evitar y que un estado inicial podía contener por error. Con el fluente, una puerta está en uno de tres estados y la cuarta combinación no existe. Añadir más adelante un cuarto estado (por ejemplo, `jammed`) es una constante más, no un predicado nuevo que mantener coherente con los otros dos.

Sección 16.8 — PDDL 3 de un vistazo

Solución 16.8.1. (a) Una preferencia en la precondición de `move`, con su precio en la métrica: `:preferences`. (b) Una restricción de trayectoria, `always`: `:constraints`. (c) Un fluente de objeto: `:object-fluents`. (d) Objetivos blandos con `is-violated` en la métrica: `:preferences`. (e) Una restricción blanda, `(preference on-time (within 600 ...))` en un modelo temporal con segundos: `:constraints` y `:preferences`.

C.17. Soluciones del Capítulo 17

El PDDL de estas soluciones está en `examples/ch17/solutions`. Como en el capítulo, las salidas de Kairos se obtuvieron con un hilo, y se omite la nota sobre la métrica que acompaña a cada plan PDDL+.

Sección 17.2 — Procesos: cambio mientras se cumple una condición

Solución 17.2.1. A 4 litros por segundo, el nivel es $4t$. Alcanza 9,5 litros en $t = 2{,}375$ s y la capacidad, 10 litros, en $t = 2{,}5$ s, cuando se dispara el evento `overflow`. La válvula debe cerrarse a partir de 2,375 s y antes de 2,5 s. Kairos decide en múltiplos de 0,1 s, y solo uno de ellos cae en ese intervalo, así que el plan debe mostrar 2,4 (con 9,6 litros en el depósito). Y lo muestra (`tank-p05.pddl`):

```
$ kairos tank-domain.pddl solutions/tank-p05.pddl
; cost = 2.4
; makespan = 2.400
0.000: (open-valve) [0.000]
2.400: (close-valve) [0.000]
2.400: @PlanEND
```

La ventana tiene 0,125 s de ancho, algo más de un paso. Con un objetivo de 9,7 litros tendría 0,075 s, de 2,425 a 2,5, sin ningún paso dentro, y Kairos no encontraría plan.

Solución 17.2.2. La fuga es un proceso cuya condición trata del agua, no de nada que haga el agente (`tank-leak-domain.pddl`):

```
(:process leaking
  :parameters ()
  :precondition (> (level) 0)
  :effect (decrease (level) (* #t (leak))))
```

Mientras la válvula está abierta, dos procesos cambian el nivel y sus ritmos se suman: $2 - 0{,}5 = 1{,}5$ litros por segundo. Ocho litros necesitan $8/1{,}5 = 5{,}33$ s; el primer paso posterior es 5,4, donde el nivel es 8,1.

```
$ kairos solutions/tank-leak-domain.pddl solutions/tank-leak-
    p01.pddl
; cost = 5.4
; makespan = 5.400
0.000: (open-valve) [0.000]
5.400: @PlanEND
```

El problema no pide la válvula cerrada, así que el plan tiene una acción y termina cuando se cumple el objetivo. Es deliberado: con este dominio, un objetivo de «8 litros y la válvula cerrada» solo se cumple en el instante de cerrar. Después de cerrar la válvula, `leaking` es el único proceso activo y el nivel baja a 0,5 litros por segundo hasta que el depósito se vacía. Entonces su propia condición, `(> (level) 0)`, se vuelve falsa y el proceso se detiene por sí solo: un proceso no necesita un evento para terminar. Un objetivo que deba *seguir* siendo cierto no es algo que un objetivo PDDL+ pueda pedir; el plan termina en `@PlanEND` y el objetivo se comprueba ahí.

Sección 17.3 — Eventos: cambio en un instante

Solución 17.3.1. Los ficheros son `tank-wrong-loop-domain.pddl` (en la carpeta del capítulo) y `tank-wrong-loop-p01.pddl`. Kairos sigue devolviendo el plan que cierra la válvula a los 4 segundos: en ese plan el nivel nunca alcanza la capacidad, la condición del evento nunca es cierta, y un evento erróneo que nunca se dispara no hace daño. El plan que cierra a los 6 segundos es otra cosa:

```
$ kairos validate tank-wrong-loop-domain.pddl solutions/tank-
    wrong-loop-p01.pddl solutions/tank-wrong-loop-p01-late.
    plan
solutions/tank-wrong-loop-p01-late.plan: error: events keep
    firing at 5.000000 (overflow): an event must make its own
     condition false [validate.event_loop]
plan invalid
```

El estudiante esperaba que este plan fuera ahora válido, puesto que el objetivo ya no menciona la inundación. Pero a los 5 segundos el evento se dispara, su condición sigue siendo cierta y tiene que dispararse otra vez, sin fin. El plan no tiene un resultado definido. El razonamiento del estudiante («el evento no hace falta para ese problema») confunde dos cosas: si al *objetivo* le importa el evento y si el *modelo* está bien formado. Un evento tiene que desactivarse a sí mismo en todos los estados que el mundo puede alcanzar, no solo en los planes que tenemos en mente. El error permanece oculto mientras el planificador evita el evento, y eso es lo que lo hace peligroso.

Solución 17.3.2. El interruptor es un evento que borra el hecho que necesita el proceso (`tank-switch-domain.pddl`):

```
(:event float-switch
  :parameters ()
  :precondition (and (valve-open)
                     (>= (level) (switch-level)))
  :effect (not (valve-open)))
```

Borrar `(valve-open)` hace falsa la condición del evento (se dispara una vez) y detiene `fill`. Añadimos también `(< (level) (switch-level))` a la precondición de `open-valve`: si no, el agente podría volver a abrir la válvula por encima del nivel del interruptor, y el evento la cerraría en el mismo instante, lo que es legal pero inútil. El problema con una sola acción pide 9 litros y la válvula cerrada:

```
$ kairos solutions/tank-switch-domain.pddl solutions/tank-
    switch-p01.pddl
; cost = 4.5
; makespan = 4.500
0.000: (open-valve) [0.000]
4.500: @PlanEND
```

El agente abre la válvula y el mundo la cierra. El problema `tank-switch-p02`, cuyo objetivo es `(flooded)`, muestra que el depósito no puede inundarse: Kairos no encuentra plan. El interruptor es el primer ejemplo de algo que desarrolla el Capítulo 18: los eventos como los dispositivos automáticos de un sistema, no solo como sus accidentes.

Sección 17.4 — Tres clases de cambio

Solución 17.4.1. El segundo rebote ocurre en $t = 2{,}6$ s y deja la pelota con $+6{,}4$ m/s. En $t = 3{,}0$, $0{,}4$ s después, la altura es $6{,}4 \cdot 0{,}4 - 5 \cdot 0{,}4^2 = 1{,}76$ m y la velocidad $6{,}4 - 4 = 2{,}4$ m/s. Fallan los dos términos de la ventana: la pelota está 14 cm por debajo de la mano y va demasiado rápido. El validador informa del primero que comprueba:

```
$ kairos validate ball-domain.pddl ball-p01.pddl solutions/
    ball-p01-early.plan
solutions/ball-p01-early.plan:3: error: step 2 (catch) at
    3.000: precondition false: (>= (height) 1.9): 1.76 >= 1.9
     is false [validate.precondition_failed]
 3 | 3.000: (catch)
plan invalid
```

El número 1.76 del mensaje es el valor calculado al reproducir los procesos y los dos eventos hasta ese instante: el validador hace la misma reproducción que hemos hecho a mano.

Solución 17.4.2. Kairos responde `no plan found` para `ball-p02.pddl`. Esta vez es un hecho sobre la pelota. Después del primer rebote, la pelota sale del suelo a 8 m/s. A una altura h, su rapidez es $\sqrt{8^2 - 2 \cdot 10 \cdot h}$, la misma al subir que al bajar; entre 1,9 y 2,1 m, eso está entre 5,1 y 4,7 m/s, siempre por encima de los 2 m/s que permite `catch`. La pelota solo va despacio cerca de lo alto del arco, a 3,2 m, donde no está la mano. Ningún instante cumple la precondición, ni en la rejilla de pasos ni fuera de ella. El ejercicio muestra cómo respaldar un `no plan found`: el planificador aporta el indicio y el modelador aporta el argumento.

Solución 17.4.3. *La cabina sube*: un proceso, que funciona mientras la cabina está `going-up` (su altura aumenta a la velocidad de la cabina). *La puerta se abre cuando la cabina llega a una planta*: un evento; nadie lo decide, y ocurre en el instante en que la altura alcanza la de la planta. *La persona pulsa un botón*: una acción. *La persona entra*: una acción si el modelo lo trata como instantáneo; si entrar lleva un tiempo que importa, el patrón inicio-proceso-parada. *La puerta se cierra cinco segundos después de abrirse*: un proceso (un temporizador que avanza mientras la puerta está abierta) y un evento (cuando el temporizador llega a 5); esta pareja es la forma en que PDDL+ escribe «al cabo de un rato». *La cabina está sobrecargada*: ninguna de las tres; es una condición del estado (la carga supera el límite), que puede aparecer en la precondición de un evento (una alarma) o de una acción.

Solución 17.4.4. Como mucho 1 m/s: un segundo de aceleración alcanza el límite y recorre 0,5 m; frenar lleva otro segundo y otros 0,5 m; los 9 m restantes necesitan 9 s al límite. Once segundos, y el validador acepta el plan (`car-p02-hand.plan`):

```
$ kairos validate car-domain.pddl solutions/car-p02.pddl
    solutions/car-p02-hand.plan
plan valid, makespan 11
```

Kairos, con un hilo y el límite de tiempo de las pruebas, devuelve el plan de 33,8 segundos que fue su primer plan para `p01`: 0,3 s de aceleración y medio minuto a 0,3 m/s. Ese plan no se acerca a ninguno de los dos límites, así que es válido para los dos problemas. La comparación dice lo mismo que la nota: en PDDL+, la búsqueda de Kairos busca pocas acciones, los dos planes tienen cinco y nada la empuja hacia el más corto. Sus planes son válidos, no

óptimos. Cuando la duración importa, lee el plan y, si se entretiene, escribe uno mejor y valídalo, o endurece el problema (un plazo, una cantidad limitada de combustible) para que entretenerse no sea un plan.

Solución 17.4.5. Dos procesos queman el combustible y un evento cala el motor (`car-fuel-domain.pddl`):

```
(:process burning
  :parameters ()
  :precondition (running)
  :effect (decrease (fuel) (* #t (idle-burn))))

(:process burning-more
  :parameters ()
  :precondition (and (running) (> (acceleration) 0))
  :effect (decrease (fuel) (* #t (push-burn))))

(:event engine-dies
  :parameters ()
  :precondition (and (running) (<= (fuel) 0))
  :effect (and (not (running)) (out-of-fuel)))
```

El segundo proceso no es «el combustible que se quema al acelerar», sino lo que se quema *de más* al acelerar: los dos funcionan a la vez y sus ritmos suman 3 ml por segundo. El evento borra `(running)`, lo que desactiva el evento, los dos procesos de consumo y el proceso `moving`, y hace imposible `park`. Con 15 ml:

```
$ kairos solutions/car-fuel-domain.pddl solutions/car-fuel-p01.
    pddl
; cost = 6.7
; makespan = 6.700
0.000: (accelerate) [0.000]
2.300: (brake) [0.000]
4.400: (brake) [0.000]
6.700: (accelerate) [0.000]
6.700: (park) [0.000]
6.700: @PlanEND
```

Quema $3\times2{,}3+1\times4{,}4 = 11{,}3$ ml y recorre $2{,}645+2{,}3\times2{,}1+2{,}645 = 10{,}12$ m. Sin límite de velocidad, el tiempo es combustible, y el plan es más rápido que los del capítulo. Con 3 ml (`car-fuel-p02.pddl`) no hay plan: el motor funciona como mucho 3 segundos, y no hay forma de gastarlos que recorra diez metros y termine en reposo.

Sección 17.5 — El mundo del mensajero en PDDL+

Solución 17.5.1. El plan dura 38 segundos y el robot está encendido todo el tiempo, así que la batería debe durar 38 segundos: el ritmo de descarga puede ser como mucho $30/38 = 0{,}79$ por segundo. (Justo a ese ritmo, la batería se vacía en el instante del `drop`, el evento `shutdown` se dispara antes y el robot ya no puede dejar el artículo; el ritmo tiene que ser estrictamente menor.) Los medicamentos se recogen en $t = 10$ s, cuando la batería marca $30 - 0{,}5 \times 10 = 25$. La posición del robot, la carga y la recogida no entran en el cálculo: el proceso `drain` solo mira `(powered r1)`.

Solución 17.5.2. En `courier-battery-p03.pddl`, el estado inicial no tiene `(powered r1)`. Todas las acciones del robot lo necesitan, así que el plan debe empezar con `power-on`:

```
$ kairos courier-battery-domain.pddl solutions/courier-battery-
    p03.pddl
; cost = 28
; makespan = 28.000
0.000: (power-on r1) [0.000]
0.000: (pick r1 meds1 pharmacy) [0.000]
0.000: (start-move r1 pharmacy lobby) [0.000]
6.000: (start-move r1 lobby corridor) [0.000]
16.000: (start-move r1 corridor ward-a) [0.000]
28.000: (drop r1 meds1 ward-a) [0.000]
28.000: @PlanEND
```

Tres acciones comparten el instante 0: se aplican en el orden del plan, sin tiempo entre ellas. La batería termina en $15 - 0{,}5 \times 28 = 1$. Mientras el robot estaba apagado, antes de que empezara el plan, no perdió nada; si el problema hubiera exigido esperar, quedarse apagado habría sido la forma de esperar gratis.

Solución 17.5.3. Un robot cuya batería se agotó en una habitación, o que se apagó allí, podría seguir recogiendo un artículo que estuviera en esa habitación: un robot muerto que agarra cosas. Es una precondición que falta, la clase de error clásica: una *acción* es aplicable cuando no debería serlo. Ningún proceso funciona gratis. Los dos errores se encuentran con pruebas distintas: la precondición que falta, con un problema en el que se pide a un robot apagado que haga algo; el proceso gratis, con un problema que necesita más tiempo o energía de la que hay.

Solución 17.5.4. Por número de acciones: primero los medicamentos (10), primero la muestra (11), el plan de Kairos (14). Por duración: 57, 62 y 73 segundos. El primero es inválido, así que la elección está entre los otros dos. Un planificador que minimizara el número de acciones devolvería «primero la muestra». Kairos no lo hizo porque su búsqueda no es óptima: devuelve el primer plan válido al que le llevan sus heurísticas, y después de aparcar la muestra en la nevera, el resto del problema les parece fácil. Qué plan debe ejecutar el robot depende de lo que le importe al hospital. Si es el tiempo, «primero la muestra». Si es la muestra, el plan de Kairos tiene su razón de ser: la muestra pasa 33 segundos fuera de una nevera en los dos, pero en el plan de Kairos nunca está a más de 25 segundos de una. Para que el planificador prefiera uno u otro, la preferencia tiene que estar en el modelo: una métrica (`total-time`, o la temperatura final) y un planificador que la optimice, o un límite más duro, como una `max-temperature` más baja, que deje solo los planes que queremos.

Sección 17.6 — Cómo planifica Kairos con PDDL+

Solución 17.6.1. Un problema clásico tiene un número finito de estados. Una búsqueda que los visita todos sin alcanzar el objetivo ha demostrado que no existe ningún plan, y Kairos dice `unsolvable`. En `tank-p03`, la decisión es *cuándo* cerrar la válvula, y hay infinitos instantes. Kairos prueba los instantes de una rejilla de 0,1 s; tras probarlos todos, sabe que no existe ningún plan *en la rejilla*. Un plan que cierre la válvula entre dos pasos no se ha considerado nunca. Así que informa de lo que sabe: `no plan found`, con una nota que lo llama «evidence but not a proof». Aquí podemos aportar nosotros la demostración: cualquier nivel de 12 o más ha pasado por 10, donde se dispara el evento. La segunda respuesta no es un fallo; es un planificador que no afirma más de lo que ha demostrado.

Solución 17.6.2. Es válido (`tank-p01-offgrid.plan`): a los 4,05 s el nivel es de 8,1 litros, el objetivo se cumple y la capacidad queda lejos.

```
$ kairos validate tank-domain.pddl tank-p01.pddl solutions/
    tank-p01-offgrid.plan
plan valid, makespan 4.05
```

Kairos no podría haberlo devuelto: sus planes solo contienen instantes que son sumas de pasos de 0,1 s. Aquí no importa, porque el conjunto de instantes de cierre válidos, desde 4 hasta justo antes de 5 segundos, es un intervalo de un segundo de ancho que contiene diez pasos. Importaría si el intervalo fuera más estrecho que un paso y cayera entre dos de ellos. El validador no tiene rejilla: acepta cualquier instante, y eso lo convierte en la herramienta para comprobar un plan que el planificador no puede producir.

Sección 17.7 — El método MODEL: quién causa cada cambio

Solución 17.7.1. La tabla de acciones, con la causa de cada cambio:

Nombre	Causa	Cuándo	Qué cambia
`switch-on`	agente	la cafetera está apagada	cafetera encendida, resistencia funcionando
`brew`	agente	cafetera encendida, agua a 90 o más	café hecho
`heat-water`	mundo, mientras	la resistencia está funcionando	la temperatura, a 5 grados por segundo
`thermostat`	mundo, instante	resistencia funcionando y agua a 95	resistencia parada

El estado necesita dos hechos donde la historia parece tener uno: `(machine-on)`, que pone el agente, y `(heating)`, que quita el termostato. Si el proceso dependiera de `(machine-on)`, el termostato tendría que apagar la cafetera entera. El dominio es `coffee-domain.pddl`.

```
$ kairos solutions/coffee-domain.pddl solutions/coffee-p01.
    pddl
; cost = 14
; makespan = 14.000
0.000: (switch-on) [0.000]
14.000: (brew) [0.000]
14.000: @PlanEND
```

De 20 a 90 grados a 5 grados por segundo: 14 segundos. El termostato no se dispara en este plan; lo haría a los 15 segundos, y a partir de entonces el agua se quedaría a 95, puesto que el modelo no tiene enfriamiento. Una cafetera más fiel perdería calor (un segundo proceso) y volvería a encender la resistencia por debajo de cierta temperatura (un segundo evento), que es el termostato del Capítulo 18.

Solución 17.7.2. *Fidelidad.* El proceso es fiel: la muestra se calienta siempre que está fuera de una nevera. El efecto en los desplazamientos solo la calienta cuando el robot se mueve con ella. Una muestra que se deja en una mesa, o que sostiene un robot que está quieto, se queda fría para siempre. La historia que lo rompe es el propio `courier-cold-p01`: la muestra espera en la mesa mientras el robot entrega primero los medicamentos. Con la codificación por desplazamientos, el plan «primero los medicamentos» se acepta, con la

muestra llegando a $4 + 0{,}1 \times 25 = 6{,}5$ grados en el modelo y estropeada en el mundo. *Tamaño.* La codificación por desplazamientos no necesita procesos ni eventos, pero sí una variante de `start-move` por cada muestra transportada, o un efecto condicional. *Guía.* La codificación por desplazamientos es un problema numérico sin tiempo, mucho más fácil de buscar. *Claridad.* El proceso dice en un solo lugar qué les ocurre a las muestras; la otra codificación lo reparte por todas las acciones durante las que pasa el tiempo, y cada acción nueva que lleve tiempo tiene que acordarse de ello. *Portabilidad.* La codificación por desplazamientos funciona en cualquier planificador numérico; el proceso necesita PDDL+. La decisión: si las muestras solo se calientan en manos del robot (empiezan y terminan en neveras, y el robot nunca espera), la codificación barata es aceptable y debe decirlo en sus Supuestos. En cuanto algo pueda calentarse mientras nadie actúa sobre ello, solo el proceso es correcto.

C.18. Soluciones del Capítulo 18

El PDDL de estas soluciones está en `examples/ch18/solutions`. Las salidas de Kairos se obtuvieron con un solo hilo, y se omite la nota sobre la métrica que acompaña a todo plan de PDDL+.

Sección 18.1 — Reglas que se cumplen en todo momento: restricciones globales

Solución 18.1.1. *Fidelidad.* Con problemas cuidadosos, las dos codificaciones aceptan los mismos planes. El evento depende de que cada problema repita `(not (fined))` en su objetivo: un problema que solo pida `(parked)` a diez metros acepta el plan con el pedal pisado durante 3,2 segundos, un coche que circula a más velocidad de la permitida y que el modelador quería prohibir. La restricción no tiene ese hueco. En cambio, el evento puede describir un mundo en el que la multa es una consecuencia y no una prohibición (una ambulancia que puede exceder el límite); la restricción no puede. *Tamaño.* Un evento instanciado y un hecho frente a una restricción instanciada: ninguna diferencia. *Guía.* Con la restricción, un estado que incumple la regla se descarta cuando se genera. Con el evento, los estados con `(fined)` se quedan en la búsqueda, y se explora todo lo alcanzable desde ellos aunque nada de ello lleve al objetivo. En nuestras ejecuciones, las dos versiones devolvieron los mismos planes; la diferencia crece con el problema. *Claridad.* La restricción dice la regla en un solo sitio y con un nombre, y el validador informa de ese nombre y del instante (`constraint 'keep-the-limit' broken at 2.000001`); con el evento, el mensaje es que el objetivo no se cumple, y el lector tiene que averiguar por qué. *Portabilidad.* El evento es PDDL+ tal como lo definieron Fox y Long; `:constraint` es una extensión de algunos planificadores. Decisión: la restricción cuando la regla es absoluta, que es el caso de un límite de velocidad en un modelo de planificación; el evento cuando incumplir la regla es posible y cambia lo que ocurre después.

Solución 18.1.2. El dominio `tank-safe-domain.pddl` quita el predicado `flooded` y el evento, y añade:

```
(:constraint never-flood
```

```
    :parameters ()
    :condition (<= (level) (capacity)))
```

El objetivo del problema es ahora solo «8 litros y la válvula cerrada». Kairos devuelve el plan que cierra a los 4 segundos, y el validador rechaza el tardío nombrando la regla:

```
$ kairos validate solutions/tank-safe-domain.pddl solutions/
     tank-safe-p01.pddl solutions/tank-safe-p01-late.plan
solutions/tank-safe-p01-late.plan: error: constraint 'never-
     flood' broken at 5.000001: (<= (level) (capacity)): 10 <=
      10 is false [validate.global_constraint]
plan invalid
```

Hay una pequeña diferencia en el borde: el evento se disparaba cuando el nivel *alcanzaba* la capacidad (`>=`), y la restricción permite un depósito exactamente lleno. Más importante es lo que el nuevo modelo no puede decir. No tiene ningún depósito inundado: el problema cuyo objetivo era la inundación (`tank-p02`) no se puede escribir, y no se puede hacer que nada dependa de una inundación, como una alarma o una tarea de limpieza. Una restricción quita una parte del mundo del modelo; un evento la conserva y la etiqueta.

Sección 18.2 — Cambio que no es lineal

Solución 18.2.1. (a) Lineal: el ritmo, 0,5, es una constante. (b) Polinómico, de grado dos: el ritmo de la altura es la velocidad, que cambia al ritmo constante de la gravedad; entre bote y bote, la altura es una parábola. (c) Exponencial: con el calefactor apagado, el único proceso es la pérdida, cuyo ritmo es proporcional a la diferencia entre la temperatura y el exterior, así que la diferencia se reduce en la misma fracción cada segundo. (d) Exponencial, por la misma razón: el ritmo depende del nivel que reduce. (e) Lineal, como (a). Una comprobación que no necesita cálculo: mira la expresión que multiplica a `#t`. Si no contiene ningún fluente que cambie mientras el proceso actúa, el cambio es lineal. Si contiene uno que cambian otros efectos, polinómico. Si contiene el fluente que el propio efecto cambia, exponencial o algo más raro.

Solución 18.2.2. El modelo (`cup-domain.pddl`) tiene un proceso, cuyo ritmo depende de la temperatura que baja:

```
  (:process cool-down
    :parameters ()
    :precondition (and (poured) (> (coffee-temp) (room-temp)))
    :effect (decrease (coffee-temp)
              (* #t (* (cooling) (- (coffee-temp) (room-temp)))))
   )
```

La acción `drink` necesita al menos 55 grados. Beber demasiado caliente no está prohibido, tiene una consecuencia: un evento marca `(burnt-tongue)` si el café está por encima de 65 grados una vez bebido, y el objetivo rechaza la marca. A mano: la diferencia con la habitación empieza en 70 y se reduce como $70\,e^{-0{,}1t}$. Vale 45 (el café a 65) cuando $t = 10\ln(70/45) = 4{,}42$ s, y 35 (el café a 55) cuando $t = 10\ln 2 = 6{,}93$ s. La ventana para beber tiene 2,5 segundos de ancho, muchos pasos.

```
$ kairos solutions/cup-domain.pddl solutions/cup-p01.pddl
; cost = 4.5
; makespan = 4.500
```

```
0.000: (pour) [0.000]
4.500: (drink) [0.000]
4.500: @PlanEND
```

El plan bebe a los 4,5 s, el primer paso después de 4,42, con el café a 64,6 grados. Una escalera de pasos de 0,1 s, cada uno con el ritmo de su inicio, pone el café por debajo de 65 grados ya a los 4,4 s, cuando en realidad está a 65,1: una razón más por la que el planificador integra con cuidado y el validador tiene la última palabra.

Sección 18.3 — Procesos que interactúan

Solución 18.3.1. Con el calefactor encendido, la temperatura se estabiliza donde la pérdida iguala al calentamiento: $0{,}1\,(T - 5) = 2$, luego $T = 25$ grados, en mitad de la franja de crecimiento y lejos de los 30 de la restricción. Ahora el calefactor puede dejarse encendido. El plan más corto tiene dos acciones, `switch-on` y `harvest`. Su duración: la distancia a 25 empieza en 10 grados y se reduce como $10\,e^{-0{,}1t}$; se entra en la franja a 22, cuando la distancia es 3, en $t = 10\ln(10/3) = 12{,}04$ s. Veinte segundos de crecimiento después son 32,04 s, y el primer paso posterior es 32,1.

```
$ kairos greenhouse-domain.pddl solutions/greenhouse-p02.pddl
; cost = 32.1
; makespan = 32.100
0.000: (switch-on) [0.000]
32.100: (harvest) [0.000]
32.100: @PlanEND
```

El calefactor más débil da el plan más largo y el más sencillo: tarda siete segundos más y no necesita conmutar. El cálculo con lápiz de dónde se estabiliza cada cantidad nos dijo, antes de ninguna búsqueda, qué clase de plan esperar.

Solución 18.3.2. Caída libre hasta 1,1 s: velocidad $-10 \cdot 1{,}1 = -11$ m/s, altitud $20 - 5 \cdot 1{,}1^2 = 13{,}95$ m. Rotores encendidos durante 0,9 s, con una aceleración neta de +5: velocidad $-11 + 4{,}5 = -6{,}5$ m/s, altitud $13{,}95 - 11 \cdot 0{,}9 + 2{,}5 \cdot 0{,}9^2 = 6{,}08$ m, a los 2,0 s. Caída libre durante 0,1 s: velocidad $-7{,}5$ m/s, altitud $6{,}08 - 0{,}65 - 0{,}05 = 5{,}38$ m, a los 2,1 s. Desde ahí, con los rotores encendidos, la altitud es $5{,}38 - 7{,}5s + 2{,}5s^2$, que se anula en $s = 1{,}18$ s, con una velocidad de $-7{,}5 + 5 \cdot 1{,}18 = -1{,}58$ m/s. Sin rotores, el dron cae durante 2 segundos y llega a -20 m/s. La condición de `crash` se cumple y la de `touch-down` no, así que se dispara `crash`: el dron deja de estar en el aire, ninguna acción es aplicable y `(landed)` no puede llegar a ser cierto nunca. El planificador no hace nada «al respecto» del evento; simplemente no encuentra plan por ese camino y tiene que elegir otro. El choque es un callejón sin salida marcado por el mundo.

Sección 18.4 — Eventos en cascada y eventos que no paran

Solución 18.4.1. `blinker-wrong`: `light-up` añade `(lit)`, que necesita `go-dark`; `go-dark` lo borra, que es lo que necesita `light-up`. Dos flechas, un ciclo, y no tiene que ocurrir nada entre los dos eventos: el ciclo se cierra en tiempo cero. `blinker`: las mismas dos flechas, pero cada evento pone además el temporizador a cero y los dos

necesitan el temporizador en medio segundo. Entre un evento y el siguiente, el proceso `clock` tiene que actuar durante 0,5 s. `greenhouse-auto` con 23 y 27: `too-warm` apaga el calefactor, una de las dos cosas que necesita `too-cold`; la otra es una temperatura de 23 o menos, y la temperatura está en 27. El proceso `losing-heat` tiene que bajarla, lo que lleva unos 2 segundos; la vuelta hacia arriba, contra la pérdida, lleva unos 4. Con los dos umbrales en 25, las flechas son las mismas y la distancia que recorrer es cero: tras `too-warm`, la condición de `too-cold` ya es cierta. Los cuatro grafos tienen la misma forma. Lo que distingue los modelos buenos de los malos no es el grafo, sino si alguna *cantidad* tiene que recorrer cada ciclo.

Solución 18.4.2. Cada ficha recibe un hecho `(falling ?d)`, un temporizador `(tilt ?d)`, un proceso que hace avanzar el temporizador mientras la ficha cae, y dos eventos (`dominoes-slow-domain.pddl`):

```
(:event land
  :parameters (?d - domino)
  :precondition (and (falling ?d) (>= (tilt ?d) (fall-time)))
  :effect (and (not (falling ?d)) (fallen ?d)))

(:event hit
  :parameters (?a ?b - domino)
  :precondition (and (fallen ?a) (next ?a ?b)
                     (not (falling ?b)) (not (fallen ?b)))
  :effect (falling ?b))
```

Cuando una ficha llega al suelo, `hit` se dispara en el mismo instante y pone en marcha la caída de la siguiente: una cascada de dos eventos, seguida de medio segundo de proceso.

```
$ kairos solutions/dominoes-slow-domain.pddl solutions/
    dominoes-slow-p01.pddl
; cost = 2.5
; makespan = 2.500
0.000: (push d1) [0.000]
2.500: @PlanEND
```

Sí, ahora el agente podría salvar la tercera ficha. Empieza a caer a 1 segundo, cuando la segunda llega al suelo. Una acción `remove` aplicable a una ficha que ni está cayendo ni ha caído podría colocarse en cualquier momento anterior, y `hit` no encontraría entonces nada que golpear. En el modelo instantáneo no existía ese momento. Que un agente pueda intervenir en una cadena de consecuencias depende de que el modelo deje pasar tiempo dentro de la cadena.

Sección 18.5 — Procesos con acciones durativas y literales temporizados

Solución 18.5.1. El plan es `courier-plus-p03-hand.plan`:

```
$ cat solutions/courier-plus-p03-hand.plan
; Exercise 18.5.1: charge while waiting for the pharmacy to
    open.
0.000: (move r1 lobby dock) [4.000]
4.000: (plug r1 dock)
50.000: (unplug r1)
50.000: (move r1 dock lobby) [4.000]
54.000: (move r1 lobby pharmacy) [6.000]
60.001: (pick r1 meds1 pharmacy)
60.001: (move r1 pharmacy lobby) [6.000]
66.001: (move r1 lobby corridor) [10.000]
```

```
76.001: (move r1 corridor ward-a) [12.000]
88.001: (drop r1 meds1 ward-a)
$ kairos validate courier-plus-domain.pddl courier-plus-p03.
    pddl solutions/courier-plus-p03-hand.plan
plan valid, makespan 88.001
```

Es diez segundos más corto que el plan de Kairos (98 s). El robot se enchufa con 6 unidades, carga durante 46 segundos a un ritmo neto de 1,8 por segundo y sale con 88,8; está en la puerta de la farmacia a los 60 segundos. El `pick` no puede ir a 60,000: el literal temporizado hace cierto `(is-open pharmacy)` en ese instante, y una acción que necesita un hecho no puede compartir instante con el acontecimiento que lo produce. Con la recogida a 60,000, el validador responde `(is-open pharmacy) is false`. Un épsilon después, 0,001 s por defecto, el hecho está ahí. Es la regla del Capítulo 10 sobre acontecimientos que interfieren, aplicada a un literal temporizado.

Solución 18.5.2. La farmacia está a 6 metros del vestíbulo, así que el robot llega a los 6 segundos, dos antes de que cierre: hay plan. El problema (`courier-plus-p04.pddl`) tiene el hecho en el estado inicial y un literal temporizado que lo borra:

```
    (is-open pharmacy)
    (at 8 (not (is-open pharmacy)))
```

```
$ kairos courier-plus-domain.pddl solutions/courier-plus-p04.
    pddl
; cost = 36
; makespan = 36.000
0.000: (move r1 lobby pharmacy) [6.000]
6.000: (pick r1 meds1 pharmacy) [0.000]
8.000: (move r1 pharmacy lobby) [6.000]
14.000: (move r1 lobby corridor) [10.000]
24.000: (move r1 corridor ward-a) [12.000]
36.000: (drop r1 meds1 ward-a) [0.000]
36.000: @PlanEND
```

Lo raro es el hueco entre los 6 y los 8 segundos. El robot recoge los medicamentos y después se queda en la farmacia hasta que cierra, sin ningún motivo: salir a los 6 da un plan de 34 segundos. Es la misma costumbre vista en el capítulo, una espera que corre hasta lo siguiente que cambia, aquí el literal temporizado. Los dos segundos cuestan 0,4 unidades de batería y nada más, y el plan es válido. Con la farmacia cerrando a los 5 segundos no habría plan.

Sección 18.6 — Un sistema, tres modelos

Solución 18.6.1. (a) Nivel 1. La primera pregunta se responde «no»: nada en la historia depende del tiempo, y el combustible es un presupuesto que gastan los viajes. (b) Nivel 2. El tiempo importa (dos robots en paralelo, una hora de secado), así que la primera respuesta es «sí»; pero el secado puede ir unido a la pintura, como acción durativa o como condición sobre el tiempo transcurrido desde que terminó la primera mano, y nada cambia fuera de las actividades, así que la segunda respuesta es «no». (c) Nivel 3. El horno se calienta tanto si el cocinero está haciendo algo como si no, y durante todo el tiempo que esté encendido: la segunda respuesta es «sí». Si preparar la masa tiene una duración conocida, es una acción durativa, y el modelo es mixto. (d) Nivel 2, con literales

iniciales temporizados para las ventanas de visibilidad: el mundo cambia por sí solo, pero en instantes del reloj, que es para lo que están los literales temporizados. La segunda pregunta se refiere al cambio que depende del estado.

Solución 18.6.2. Nivel 2: el robot fue primero a la base de carga y llegó con $20 - 4 = 16$. La duración de `charge` está acotada por $(100 - 16)/2 = 42$ segundos, y el plan usa la cota. Nivel 3: el robot recogió primero los medicamentos y llegó a la base de carga con 4. Enchufada y encendida, la batería gana $2 - 0{,}2 = 1{,}8$ por segundo, y llega a 100 tras $96/1{,}8 = 53{,}33$ s; el plan desenchufa en el último paso antes de eso, 53,3. Lo que la tarea necesita es menos. En el nivel 3: quedan 26 metros después de la base de carga y hay 4 unidades en la batería, así que hacen falta 22 más, 12,2 segundos de carga. Ninguno de los dos planificadores miró ahí. En el nivel 2, los motores de Kairos prueban una duración que es una desigualdad en sus extremos, y el extremo es una batería llena. En el nivel 3, la espera larga se detiene cuando una condición numérica está a punto de cambiar, y «la batería basta para el resto del plan» no es una condición del modelo: ninguna precondición, evento ni objetivo la menciona. La única condición a la vista era la de `battery-full`. Los dos planes son válidos. Para obtener el más corto de un planificador, la presión tiene que estar en el problema: un plazo para la entrega, como literal temporizado.

Sección 18.7 — Límites: pasos y tolerancias

Solución 18.7.1. A 5 litros por segundo, el nivel está en la dosis desde $7{,}9/5 = 1{,}58$ s hasta $8{,}1/5 = 1{,}62$ s: una ventana de 40 ms, menos de medio paso. Y Kairos encuentra un plan (`doser-p06.pddl`):

```
$ kairos doser-domain.pddl solutions/doser-p06.pddl
; cost = 1.6
; makespan = 1.600
0.000: (open-valve) [0.000]
1.600: (close-valve) [0.000]
1.600: @PlanEND
```

La ventana es estrecha, pero un paso cae dentro de ella: a los 1,6 s el nivel es exactamente 8. La regla del capítulo, una ventana tan ancha como el cambio de un paso, es lo que *garantiza* que algún paso caiga en la ventana, esté donde esté la ventana. Una ventana más estrecha es cuestión de suerte: con 5 litros por segundo, los pasos pasan casualmente por 8; con 7, pasan por 7,7 y 8,4. Un modelo no debería depender de esa suerte; un pequeño cambio en un ritmo, hecho meses después por otra persona, convertiría un problema resoluble en `no plan found`.

Solución 18.7.2. En la búsqueda, el interruptor se examina en los pasos. A 1,0 s el nivel es 7,0, por debajo del objetivo; a 1,1 s es 7,7, por encima, y el evento cierra la válvula con 7,7 litros, fuera de la dosis. En tiempo continuo, el interruptor cierra cuando el nivel llega a 7,6, también fuera de la dosis. La propuesta falla dos veces: la búsqueda sigue sin ver ningún estado con la dosis correcta, y ahora el validador rechazaría también el plan. Ajustar un umbral para que el mundo *por pasos* salga bien es apuntar al mundo equivocado: la búsqueda es solo de donde salen los candidatos, y el modelo tiene que ser correcto en tiempo continuo, que es donde se juzgan los planes. Las opciones honradas son las del capítulo: una tolerancia tan ancha como un paso, o un caudal de entrada más lento.

Sección 18.8 — El método MODEL: probar y elegir modelos híbridos

Solución 18.8.1. *Trivial* (`drone-p04.pddl`): el dron empieza 10 cm por encima del suelo. Lo esperado: ninguna acción; cae durante 0,14 s y toma tierra a 1,4 m/s. Kairos devuelve un plan sin acciones que termina a los 0,2 s:

```
$ kairos drone-domain.pddl solutions/drone-p04.pddl
; cost = 0.2
; makespan = 0.200
0.200: @PlanEND
```

El plan vacío no es «el objetivo ya es cierto»: el objetivo se hace cierto esperando. Prueba que `touch-down` se dispara. *Sin plan* (`drone-p03.pddl`): la batería está vacía, `rotors-up` nunca es aplicable y el dron llega a 20 m/s. Lo esperado y lo obtenido: `no plan found`. Prueba `crash`. *Tentador* (`drone-p05.pddl`): el dron empieza 5 m por debajo del techo, subiendo a 4 m/s con los rotores encendidos. Un plan que empiece esperando incumple la restricción antes de un segundo, así que la primera acción tiene que llegar enseguida.

```
$ kairos drone-domain.pddl solutions/drone-p05.pddl
...
; better plan 2
; cost = 4
; makespan = 4.000
0.000: (rotors-down) [0.000]
0.300: (rotors-up) [0.000]
0.500: (rotors-down) [0.000]
1.900: (rotors-up) [0.000]
4.000: @PlanEND
```

Los rotores se apagan en el instante cero en el mejor plan (a los 0,2 s en el primero, que aun así se mantiene por debajo del techo). La batería de pruebas aún no cubre `battery-empty`: lo haría un cuarto problema con batería para un segundo de empuje.

Solución 18.8.2. (1) *Lápiz.* Hecho en el texto: las tres sumas de ritmos (−0,2, −1, +1,8), y a partir de ellas el coste de la espera en `p03` (12 unidades para una batería de 10). (2) *Cada evento.* `shutdown` no puede dispararse nunca en un plan válido cuyo robot haga falta, así que se prueba evitándolo: `p03`, donde el plan se enchufa un paso antes, y el problema sin plan de más abajo. `battery-full` no se dispara en ningún plan impreso en el capítulo, porque los planes desenchufan un paso antes: esto falta. Lo cubriría un problema cuyo plan deje el robot enchufado hasta que la batería esté llena. (3) *Horizonte.* Faltaba un problema sin plan; `courier-plus-p05.pddl` tiene la batería a 3 y la base de carga a 4 metros, y termina con `no plan found` en medio segundo. (4) *Validación continua.* El plan del Ejercicio 18.5.1, con instantes fuera de la rejilla (60,001), es válido; el mismo plan con la recogida a 60,000 no es válido. (5) *Reproducir y dibujar.* La figura de la batería para `p03`. (6) *Ventanas.* La única ventana en la que un plan debe acertar es la apertura de la farmacia, que dura para siempre en `p03` y 8 segundos en el Ejercicio 18.5.2; las condiciones sobre la batería son umbrales con márgenes amplios. Nada es más estrecho que un paso. Lo que muestra la lista es típico: las pruebas que salen de forma natural ejercitan los planes que esperamos, y los eventos son lo que queda sin probar.

C.19. Soluciones del Capítulo 19

Cada solución es un modelo de referencia, en `examples/ch19/solutions`, con su batería de pruebas en el `tests.txt` del capítulo. Para cada proyecto damos las líneas de la ficha del modelo que tomaron una decisión, los patrones de las dos tablas, las pruebas con lo que respondió Kairos (un hilo) y la alternativa que debería sopesar el paso L. Los dominios son cortos y no se imprimen enteros: léelos junto a estas notas. Un modelo distinto del nuestro que pase sus propias pruebas honradas también es una solución.

Sección 19.2 — Las cinco listas de comprobación

Solución 19.2.1. *La biblioteca* (`library-domain.pddl`).

Ficha. Agentes: un robot, así que no necesita objeto. Alcance, fuera: dónde va exactamente un libro en la estantería; el orden de los libros en la cesta. Supuestos: un libro que se coge del mostrador no se vuelve a dejar allí.

Tablas. El corredor es una **tabla estática** con **hechos espejo**, `(adjacent ?a ?b)`; el pasillo de un libro es otra tabla estática, `(belongs ?b ?p)`. La posición del robot es un **invariante de a lo sumo uno**. La cesta es una **cadena de sucesores**: `(holding ?n)` con `(next ?n ?m)`, y el límite de tres libros es el final de la cadena en el fichero del problema. El mostrador es una constante, porque la acción `take` lo nombra. `take` y `shelve` son **acciones sin vuelta atrás**, que es lo que dice el supuesto.

Pruebas. Trivial (`p01`): 4 pasos. Típica (`p02`), cinco libros y tres pasillos: 18 pasos en dos viajes, con optimalidad demostrada tras 928 estados; el primer viaje lleva los tres libros de los pasillos lejanos. Sin plan (`p03`): un pasillo cerrado por obras, sin hecho `adjacent`, es `unsolvable` y la nota nombra `(shelved saga)`. Las trampas:

```
$ kairos validate library-domain.pddl library-p02.pddl library-
    p02-cheat.plan
library-p02-cheat.plan:5: error: step 4 (take botany n3 n4) at
     3.000: unknown object 'n4' [validate.unknown_object]
 5 | (take botany n3 n4)
plan invalid
```

Un cuarto libro ni siquiera se puede escribir: no existe un cuarto valor de la cuenta. El libro dejado en el pasillo equivocado se rechaza porque `(belongs saga history)` es falso.

Paso L. El rival de la cadena son los **huecos**, tres lugares con nombre en la cesta. Nada en la historia distingue los lugares, así que los huecos solo añadirían planes simétricos, como pasó con la bandeja del Capítulo 7. Un fluente numérico también serviría y haría perder los planificadores clásicos; la cuenta es pequeña y cambia de uno en uno, el caso que el Capítulo 8 deja para la cadena. La decisión que hay que vigilar es el mapa: con un solo corredor, `move` entre vecinos mantiene pequeño el modelo, y una **acción fusionada** que va a cualquier sitio ocultaría el único coste que hace distintos dos viajes de cinco.

Solución 19.2.2. *La planta del hotel* (`hotel-domain.pddl`).

Ficha. Alcance, fuera: la ruta entre habitaciones (`go` es una **acción fusionada** de cualquier sitio a cualquier otro; las limpiadoras conocen el camino) y el tiempo. Supuestos: la ropa de cama del cuarto de material nunca se acaba.

Tablas. La aspiradora es un **cerrojo de recurso**: `(vacuum-at ?p)` o `(has-vacuum ?c)`, nunca los dos, y el objetivo la pide de vuelta en el cuarto de material, como aconseja el Capítulo 7 para todo cerrojo. Los carteles y las salidas son tablas estáticas. El objetivo son dos reglas sobre todas las habitaciones:

```
(:goal (and
  (forall (?r - room)
    (or (do-not-disturb ?r) (vacuumed ?r)))
  (forall (?r - room)
    (imply (checkout ?r) (fresh-linen ?r)))
  (vacuum-at store))))
```

Pruebas. Trivial: 5 pasos. Típica (`p02`), cuatro habitaciones: 14 pasos, con optimalidad demostrada. La trampa, una aspiradora en dos manos, se rechaza con `(has-vacuum ben) is false`. La prueba «sin plan» es el agujero de la historia: un huésped que se ha ido y cuya puerta sigue mostrando el cartel. La segunda regla pide ropa de cama limpia y el cartel prohíbe entrar, así que `p03` es `unsolvable` tras un estado. Aquí el modelo cumplió su función: encontró una pregunta para el dueño del problema. Si la respuesta es «sáltate la habitación», la segunda regla recibe la misma excepción que la primera.

La segunda limpiadora. En el mejor plan de `p02`, una limpiadora lo hace todo y la otra no sale nunca del cuarto de material. La longitud del plan cuenta pasos, y dos personas trabajando a la vez no acortan un plan. El modelo es correcto para «qué hay que hacer»; la pregunta «quién hace qué, para terminar pronto» necesita las acciones durativas del Capítulo 10, y eso sería la versión 2 de este modelo.

Paso L. El objetivo con `forall` frente a un objetivo enumerado (Capítulo 5): la lista ocultaría las dos reglas y, con ellas, habría ocultado la contradicción de `p03`. El `or` y el `imply` hacen perder los planificadores sin disyunciones; unos predicados complementarios (`may-enter`) los recuperarían.

Sección 19.3 — Qué característica para qué necesidad

Solución 19.3.1. *El camión cisterna* (`watering-domain.pddl`).

Guía. ¿Se cuenta o se mide algo? Sí, litros, así que fluentes numéricos. ¿Un plan mejor? Sí, una suma de costes fijos de los movimientos, así que costes de acción. Todas las preguntas posteriores: no. El tiempo queda fuera porque nadie preguntó cuándo; la ficha lo dice.

Ficha. La decisión de la historia: un parterre recibe toda su agua en una sola visita. Entonces `water` es una sola acción con la precondición `(>= (tank) (need ?b))`, y un parterre que necesita más que un depósito lleno no se puede regar nunca. Esa es nuestra prueba «sin plan» (`p03`, un césped de 1200 litros), y una línea de *Supuestos* que el pueblo debería leer.

Tablas. El depósito es una **magnitud acotada**: la precondición lo mantiene por encima de cero y `refill` usa `assign`, así que no puede pasar de la capacidad. Las carreteras son hechos espejo con una distancia en cada sentido; los metros van a `(total-cost)`.

Pruebas. Trivial: coste 800. Típica (`p02`): cuatro parterres que necesitan 1900 litros. Kairos imprime planes de 5000, 3600 y 2600 metros y demuestra que el último es óptimo: el camión riega las rosas y el césped, rellena en la boca de riego de la estación, riega allí y vuelve por el parque para los tulipanes. El primer plan volvía dos veces a

la cochera para rellenar. La trampa, 1100 litros de un depósito de 1000, se rechaza con `600 >= 700 is false`.

Paso L. La alternativa es un parterre regado por partes, con `(need ?b)` como fluente dinámico que cada visita disminuye. No pierde soluciones y admite el césped grande; necesita una acción para cada cantidad, o una cantidad que elija el planificador, y hace más difícil encontrar el mejor plan. Con parterres mucho más pequeños que el depósito, como aquí, no compra nada.

Solución 19.3.2. *La red de agua* (`network-domain.pddl`).

Guía. ¿Es un hecho consecuencia de otros? Sí: hasta dónde llega el agua se sigue de las válvulas, a través de cadenas de tuberías de cualquier longitud. Es un predicado derivado recursivo (Capítulo 13); ningún conjunto de efectos podría mantenerlo correcto, porque cerrar una válvula deja sin agua un número desconocido de juntas. ¿Hay una regla sobre todo el plan? Sí, el consultorio: una restricción (Capítulo 15).

```
(:derived (supplied ?n - node)
  (or (source ?n)
      (exists (?p - pipe ?m - node)
        (and (joins ?p ?m ?n) (open ?p) (supplied ?m)))))
```

Ficha. La regla del consultorio pertenece al problema, no al dominio: otro pueblo tiene otro consultorio, o ninguno. Es estricta.

Pruebas. Trivial: cerrar, reparar, abrir. Típica (`p02`): un anillo en el que una válvula, `p5`, empieza cerrada.

```
$ kairos network-domain.pddl network-p02.pddl
; cost = 6
(open-valve p5)
(close-valve p1)
(close-valve p3)
(repair j2)
(open-valve p1)
(open-valve p3)
```

El plan abre primero el otro lado del anillo, para que el consultorio conserve el agua, y solo después aísla la junta. La reparación en tres pasos que ignora el consultorio se rechaza con `the plan violates the problem :constraints`, y una reparación con agua en la junta, con `(supplied j2) is true`. Sin la quinta tubería (`p03`) el problema es `unsolvable`: la única manera de secar la junta deja seco el consultorio.

Paso L. La regla podría ser una precondición de `close-valve`, pero qué válvulas ponen en peligro el consultorio depende del estado de todas las demás, así que la precondición necesitaría el predicado derivado *después* de la acción: no se puede escribir. Es una regla sobre estados, y la restricción es su única codificación directa. En Portabilidad el modelo paga dos veces: tres planificadores de los ocho leen predicados derivados recursivos y dos leen restricciones.

Solución 19.3.3. *La escala* (`turnaround-domain.pddl`).

Tablas. Cinco acciones durativas con duraciones constantes. Los equipos son **testigos**: `(cleaners-free)` y `(truck-free)`, borrados al inicio y añadidos al final. La regla del combustible es un **invariante**: `refuel` necesita `(over all (cabin-empty ?p))`, `deboard` añade ese hecho al final y `board` lo borra al inicio, así que el repostaje solo cabe entre los dos. El *catering* no tiene testigo, porque la historia da a cada avión su propio camión.

Pruebas. A mano, un avión: 10 minutos para desembarcar, después los tres servicios a la vez, de los que el repostaje es el más largo, y

luego 15 para embarcar: 45. Dos aviones: el camión los reposta uno tras otro, de 10 a 30 y de 30 a 50, y el segundo embarca a los 50: 65.

```
$ kairos turnaround-domain.pddl turnaround-p02.pddl
; cost = 65.002
; makespan = 65.002
0.000: (deboard vy12) [10.000]
0.000: (deboard ib31) [10.000]
10.000: (cater vy12) [12.000]
10.000: (clean vy12) [15.000]
10.000: (cater ib31) [12.000]
10.001: (refuel vy12) [20.000]
25.001: (clean ib31) [15.000]
30.001: (board vy12) [15.000]
30.002: (refuel ib31) [20.000]
50.002: (board ib31) [15.000]
```

El planificador encuentra los dos *makespan*. Las trampas son dos planes a mano: embarcar mientras entra el último combustible (`over-all condition broken ... (cabin-empty ib31) is false`) y un camión en dos aviones (`(truck-free) is false`). La prueba «sin plan» no tiene camión de combustible, y es `unsolvable` con una nota que nombra `(truck-free)`.

Paso L. La regla del combustible como invariante frente a un **testigo de fase** (desembarcado, atendido, embarcando). Las fases se leen mejor y prohíben un plan que la historia permite: embarcar en cuanto terminan la limpieza y el *catering*, cuando el combustible entró pronto. El modelo no usa precondiciones negativas a propósito: funciona en los planificadores temporales que no las tienen.

Solución 19.3.4. *La calzada* (`causeway-domain.pddl`).

Ficha. Supuestos, el calendario: la marea y el turno cambian en momentos conocidos, haga lo que haga el conductor. Son literales iniciales temporizados de cada problema. Las decisiones: un cruce necesita marea baja desde su primer minuto hasta el último, `(over all (low-tide))` y también `at start`, porque el intervalo de `over all` es abierto (Capítulo 10); y no se hace nada después del turno, `(over all (on-shift))` en las dos acciones.

Pruebas. Trivial: ida y vuelta en la primera marea, 55 minutos. Típica (`p02`), tres paquetes:

```
$ kairos causeway-domain.pddl causeway-p02.pddl
; cost = 320.001
; makespan = 320.001
0.000: (cross mainland island) [20.000]
20.001: (deliver p3 island) [15.000]
35.002: (deliver p2 island) [15.000]
50.003: (deliver p1 island) [15.000]
300.001: (cross island mainland) [20.000]
```

Cuarenta y cinco minutos de trabajo no caben entre el minuto 20 y el cruce de vuelta, que debería empezar como tarde a los 40, así que la furgoneta se queda en la isla y vuelve con la segunda marea. La trampa sale en el minuto 50 y se rechaza: `over-all condition broken after the happening at 60.000000: (low-tide) is false`. Para la prueba «sin plan» escribimos dos. Un turno que termina en el minuto 30 (`p03`) se demuestra `unsolvable`: la nota dice que los literales temporizados quitan lo que necesita el objetivo antes de que se pueda alcanzar, la demostración del Capítulo 13. Un día con una sola marea baja (`p04`), en el que la furgoneta puede repartir y no volver nunca, tampoco tiene plan, y Kairos no puede decirlo: con un límite de 3 segundos responde `timeout`, código de salida

11, y con 20 lo mismo. Conserva la primera clase en la batería de pruebas, y la segunda como una nota en la ficha.

Paso L. El final del turno como literal temporizado frente a `(within 480 ...)` sobre el objetivo (Capítulo 15): la restricción es una línea, pero deja a la furgoneta cruzar fuera de horario camino de un objetivo anterior. La marea no puede ser una restricción en absoluto: es una ventana que se abre dos veces.

Solución 19.3.5. *El robot aspirador* (`vacuum-domain.pddl`).

Ficha. Obligatorio: el robot termina en su base. Deseado, con pesos 8, 5, 4 y 2: la cocina, el salón, el cuarto del bebé y el recibidor están limpios. Los pesos solo compiten entre sí, así que su unidad es libre y la métrica no tiene otro término. Reglas: «el recibidor nunca antes que la cocina» mira la historia y es `(sometime-before (cleaned hall) (cleaned kitchen))`; «el cuarto del bebé una vez como mucho» es `(at-most-once (robot-at nursery))`. Las dos son estrictas, y las dos pertenecen al problema: otra familia tiene otras reglas, y el dominio no tiene ninguna.

Tablas. La batería es una magnitud acotada con una precondición antes de cada disminución. El dominio tiene veinte líneas: todo lo que hace interesante este proyecto está en el fichero del problema.

Pruebas. Típica (`p02`): un 90 % de trabajo deseado, un 70 de batería. El mejor plan encontrado limpia el cuarto del bebé, la cocina y el recibidor, en ese orden, y renuncia al salón: coste 5. A mano: la cocina y el salón juntos necesitan 60 y seis movimientos, 72; ningún conjunto con el salón y la cocina cabe, y ningún conjunto sin la cocina vale más. Las trampas: el recibidor al entrar (`the plan violates the problem :constraints`) y un plan que necesita un 94 %, rechazado en un movimiento con `0 >= 2 is false`. Sin plan (`p03`): el robot empieza en el salón con un 1 %. Se puede renunciar a todos los deseos, pero no al camino de vuelta: `unsolvable`.

Paso L. La regla del recibidor como precondición de `clean` con una tabla estática `before`, como los platos del camarero del Capítulo 7: portable e igual de exacta, pero lleva al dominio una regla de una familia. La regla del cuarto del bebé necesitaría el autómata del Capítulo 15, un hecho `left-nursery`. Las restricciones son el sitio adecuado para las dos mientras el planificador las lea, y dos planificadores de los ocho lo hacen.

Solución 19.3.6. *La esclusa* (`lock-domain.pddl`).

Quién causa cada cambio. El esclusero abre válvulas y compuertas y el barco navega: acciones. El agua sube mientras las válvulas de arriba están abiertas, lo mire el esclusero o no: un proceso, y otro para vaciar. El agua se detiene cuando los niveles se igualan, en un instante: un evento para cada sentido, que además cierra las válvulas en el modelo y pone el nivel al del tramo. Con ese `assign` el nivel vuelve a ser un valor exacto, la única manera segura de compararlo con una igualdad (Capítulo 18); las compuertas lo siguen comprobando con `>=` y `<=`.

Ficha. «Las dos compuertas nunca están abiertas a la vez» se escribe como precondiciones de las dos acciones que abren una compuerta, porque solo ellas pueden romperla. Una restricción global lo diría una sola vez; hemos dejado el modelo libre de una extensión que leen menos planificadores.

Pruebas. Primero, el lápiz: tres metros a medio metro por minuto son seis minutos.

```
$ kairos lock-domain.pddl lock-p02.pddl
; cost = 12
; makespan = 12.000
0.000: (close-gate upper-gate) [0.000]
0.000: (open-lower-paddles) [0.000]
6.000: (open-lower-gate) [0.000]
6.000: (sail lower chamber lower-gate) [0.000]
6.000: (close-gate lower-gate) [0.000]
6.000: (open-upper-paddles) [0.000]
12.000: (open-upper-gate) [0.000]
12.000: (sail chamber upper upper-gate) [0.000]
12.000: (close-gate upper-gate) [0.000]
12.000: @PlanEND
```

La cámara que empieza llena se vacía y se vuelve a llenar: doce minutos, el doble de los seis de la prueba trivial. Una prueba ha disparado cada uno de los dos eventos, como pide el Capítulo 18. La trampa abre la compuerta de arriba a los cuatro minutos y el validador, en tiempo continuo, da el nivel: `(>= (level) (upper-level)): 2 >= 3 is false`. La prueba «sin plan» pide las dos compuertas abiertas y termina con `no plan found` tras 1512 estados: código de salida 14, como todo problema PDDL+ sin plan, y una búsqueda corta porque nada en este modelo puede seguir cambiando mucho tiempo.

Paso L. El llenado como acción durativa de seis minutos. Según la guía del Capítulo 18, aquí basta: la duración se conoce al empezar y nada la interrumpe. El proceso se gana su sitio en cuanto crece la historia: una cámara que empieza medio llena, un esclusero que cierra las válvulas a mitad, una fuga. Elegimos el proceso porque una esclusa es un sistema cuya ley es el agua, y escribimos en la ficha que la versión durativa es la portable.

Sección 19.7 — Hacer crecer un modelo por versiones

Solución 19.7.1. *La lavandería, con monedas* (`laundry-cost-domain.pddl`).

Ficha. La *Calidad* es nueva: entre los planes que lo doblan todo, cuantas menos monedas, mejor. *Supuestos*: cargar, descargar y doblar son gratis. Nada más se mueve.

Tablas. Paso O: una función estática `(price ?m)` y el contador `(total-cost)`, que empieza en cero. Paso D: la columna de coste, ceros incluidos: `wash` y `dry` suman el precio de su máquina, y las otras tres acciones no suman nada, a propósito. El dominio es la versión 2 con un requisito, dos funciones y dos efectos más.

Pruebas. Las pruebas de la versión 2 conservan sus veredictos. La típica tiene ahora un coste que se puede calcular antes de ejecutar, dos lavados y dos secados: 10 monedas.

```
$ kairos laundry-cost-domain.pddl laundry-cost-p01.pddl -v
...
kairos: solved (proven optimal), cost 10
...
```

Con la segunda lavadora (`p02`) el mejor plan cuesta 9: un lavado de las tres coladas en el tambor grande, 5 monedas, y dos secados, porque la secadora sigue admitiendo 8 kg. Es la prueba con dos planes de coste conocido que pide el Capítulo 9: el planificador debe preferir 5 monedas a dos lavados de 3. La trampa se ahorra la secadora y dobla mojada la ropa blanca; se rechaza como en la versión 1.

Qué viene después. Las monedas encajan en la versión 3 como segundo término de la métrica, una suma ponderada de minutos y monedas (Capítulo 9), y entonces la ficha debe decir cuántos minutos vale una moneda. Es de esperar que las dos medidas tiren en sentidos opuestos: el plan más barato llena los tambores y los espera, y el más rápido hace más ciclos. En la versión 4, las monedas serían una cosa más que un deseo tiene que valer.

C.20. Soluciones del Capítulo 20

Los ejercicios de este capítulo son proyectos, y un proyecto tiene más de una buena solución. Lo que sigue son soluciones de referencia: ficheros de `examples/ch20/solutions` que pasan sus pruebas, y las decisiones que hay detrás. Una solución tuya que tome otras decisiones es correcta si sus pruebas pasan y puedes defenderlas con los cinco criterios.

Sección 20.3 — Versión 1: quién lleva qué

Solución 20.3.1. La ficha del modelo cambia en *Supuestos*: «La UCI es restringida; un robot solo entra en ella con tarjeta de acceso; `r1` tiene una.» En la tabla de estado son dos filas estáticas con los nombres del Capítulo 5, `(restricted ?l)` y `(has-keycard ?r)`, y en la tabla de acciones una condición más de `move`. El dominio de referencia (`courier-fleet-keycard-domain.pddl`) la escribe junto a la regla del laboratorio, con la misma forma:

```
    (robot-at ?r ?from) (connected ?from ?to)
    (or (not (restricted ?to)) (has-keycard ?r))
    (or (not (sterile ?to))
```

La decisión clave es que los dos hechos son tablas estáticas del problema y que el dominio no dice nada sobre qué habitación ni qué robot: otra planta, o una segunda tarjeta, es una línea en un fichero de problema. Una acción para pasarse la tarjeta sería otra historia, y la responsable no la contó.

Pruebas: la prueba trivial del capítulo sigue pasando (la sala A no es restringida). La prueba típica, `courier-fleet-keycard-p01`, es la ronda con `(restricted icu)` y `(has-keycard r1)`; Kairos devuelve 35, 33 y luego 31 acciones. La prueba sin plan, `p02`, quita la tarjeta y es `unsolvable`, con la misma nota que para la puerta cerrada: `deliver` nunca es aplicable. Una prueba de la trampa es `p01` con el objetivo de que `r2` acabe en la UCI.

El mejor plan sigue teniendo 31 acciones y sigue usando un solo robot, pero ahora es `r1`: las 31 acciones son suyas. En un modelo que cuenta acciones, la manera más barata de respetar una regla sobre quién puede entrar es dárselo todo al que puede. La tarjeta cambia *quién*; solo el tiempo, en la versión 2, hace que merezca la pena usar dos robots.

Solución 20.3.2. Las dos codificaciones difieren en un estado en el que se lleva una muestra sin sellar dentro del laboratorio sin que haya entrado por la puerta en manos de un robot. El problema `courier-fleet-leak-p01` lo construye: la muestra empieza en el laboratorio y tiene que ir a la sala B.

```
$ kairos courier-fleet-domain.pddl courier-fleet-leak-p01.pddl
; cost = 4
(pick r1 sample1 lab)
(move r1 lab corridor)
(move r1 corridor ward-b)
(deliver r1 sample1 ward-b)
$ kairos courier-fleet-rule-domain.pddl courier-fleet-leak-p01.
    pddl
note: no plan: every state reachable from the initial one was
    searched (9 states) and none satisfies the goal. [
    orchestrator.why]
kairos: unsolvable (exhausted)
```

Después de la recogida, el robot está en una habitación estéril con una muestra sin sellar: la restricción se viola, y como `seal` necesita que la muestra se esté llevando, no hay manera de evitarlo. El dominio con la precondición no se entera, porque al estado prohibido se llegó con `pick` y solo estaba vigilado `move`. Es la fuga del Capítulo 15, en nuestro propio modelo.

La reparación, `courier-fleet-tight-domain.pddl`, da a `pick` la misma regla: en una habitación estéril, una muestra que se recoge tiene que estar sellada.

```
                    (or (not (sterile ?l))
                        (forall (?s - sample)
                          (or (not (= ?i ?s)) (sealed ?s)))))
```

Necesita `:equality`, porque `?i` es un artículo y `sealed` habla de muestras. Con ella, los dos dominios responden `unsolvable` al problema de la fuga, y la ronda conserva su plan de 31 pasos.

En cuanto a la fila de Fidelidad: «solo vigila la puerta» era una diferencia real, inofensiva para el Alcance de la ficha (las muestras van al laboratorio) y equivocada fuera de él. Quedarse o no con el dominio más sencillo es una decisión del paso M: si el laboratorio también puede enviar muestras fuera, la ficha tiene que decirlo y el modelo es el dominio ajustado. El ejercicio muestra también para qué sirve la versión con la restricción: encontramos la fuga preguntando dónde no están de acuerdo las dos codificaciones, no mirando fijamente una de ellas.

Sección 20.4 — Versión 2: tiempo y batería

Solución 20.4.1. No hace falta cambiar el dominio porque los cargadores son datos: `has-charger` y `charger-free` toman un lugar como parámetro, y `charge` funciona dondequiera que se cumplan los dos. El problema `courier-fleet-time-p05` añade una línea a la ronda:

```
    (has-charger ward-b) (charger-free ward-b)
```

Con tres segundos, Kairos devuelve planes de 230,631, 177,224 y 137,619. El último carga `r2` en la base de carga durante 14 segundos, como antes, y más tarde carga `r1` en la sala B durante 19,6 segundos antes de la última entrega, que es la caja urgente, a 137.

El plan es peor que el de 113,017 del capítulo, aunque todo plan del problema antiguo sigue siendo un plan del nuevo. Al modelo no le pasa nada. Un segundo cargador añade acciones instanciadas y, con ellas, muchas maneras nuevas de gastar tiempo que a la heurística le parecen útiles (una carga siempre es aplicable a un robot que no está lleno y está junto a un cargador); en los mismos

tres segundos, la búsqueda *anytime* aún no había llegado al plan que ignora el cargador nuevo. La respuesta a la pregunta es no: más opciones nunca son peores en principio y a menudo son peores en la práctica, con un límite de tiempo. Es el criterio de Guía, y es un motivo para comparar los planes de una versión nueva con los de la antigua. Cuando el plan nuevo es peor y el antiguo sigue siendo válido, valida el antiguo y quédate con él:

```
$ kairos validate courier-fleet-time-domain.pddl courier-fleet-
    time-p05.pddl courier-fleet-time-p02.plan
plan valid, makespan 113.017, metric 113.017
```

Solución 20.4.2. El dominio `courier-fleet-time-blocks-domain.pddl` sustituye la carga por un bloque: 5 segundos, 5 veces el ritmo (25 unidades), aplicable solo mientras quepa en la batería un bloque entero, de modo que nunca se supera la capacidad.

```
:duration (= ?duration 5)
...
  (at start (<= (+ (battery ?r) (* 5 (charge-rate ?r)))
                (battery-capacity ?r)))
...
  (at end (increase (battery ?r) (* 5 (charge-rate ?r))))
```

Sobre la ronda devuelve cuatro planes en tres segundos, el último con *makespan* 107,015: `r2` carga un bloque, de 30 a 55, y hace dos entregas con 46 metros; `r1` hace las otras dos y entrega la caja urgente la última, a 107.

La nueva columna de la tabla: *Fidelidad*, cercana a la duración elegida (cargas de 5, 10, 15 segundos), pero un robot por encima de 75 no puede cargar en absoluto, lo que es una pequeña solución perdida; *Tamaño*, 82 acciones instanciadas, como la carga completa; *Guía*, buena, 107,015, el mismo makespan que la duración elegida con una acción de carga en lugar de cuatro; *Claridad*, sencilla, con una constante arbitraria; *Portabilidad*, cualquier planificador temporal con números.

Si un robot puede dejar el cargador cuando quiera, elegiríamos los bloques, para este planificador y este problema: llegan al plan de la duración elegida, con un solo envío al cargador en lugar de cuatro, y funcionan en más planificadores. El tamaño del bloque decidió dos cosas: la precisión de una carga (aquí bastó un bloque porque $30 + 25 \geq 46$; con un bloque de 2 segundos el plan necesitaría ocho) y la zona muerta cerca del tope de la batería. Merece el mismo cuidado que cualquier otro número del modelo: una buena elección es la carga más pequeña por la que merece la pena ir a la base de carga.

Sección 20.5 — Versión 3: un calendario y deseos

Solución 20.5.1. La acción es `deliver` y la condición es `over all`: quien recibe tiene que estar allí durante los dos segundos, como en el `pick` del capítulo. En `courier-fleet-day-hours-domain.pddl`:

```
(over all (robot-at ?r ?l))
(over all (is-open ?l))
(at end (in-time ?i)))
```

El cambio tiene una consecuencia fácil de olvidar y que la primera ejecución descubre: ahora hay que declarar abierto en el problema cada destino. Con solo el vestíbulo y la sala B abiertos, como en los

ficheros del capítulo, la caja urgente no se puede entregar y el problema no tiene plan. El problema `courier-fleet-day-hours-p01` abre la sala A, la UCI y el laboratorio, y cierra el laboratorio a 90. El plan de `r1`, con tres segundos: coste 12, luego coste 11. `r1` entrega la caja urgente a 67 y se va directo a casa, adonde llega a 98. Pierde la muestra: no puede estar en el laboratorio antes de 67 + 5 + 2 + 3 + 20 = 97, y el laboratorio cierra a 90. El coste 11 son la caja rutinaria, la muestra y la comida.
`r2` tampoco puede recuperar la muestra si además lleva la caja rutinaria: desde la sala A, a 62, la muestra está a 24 metros y el laboratorio a 20 más. Podría hacer la muestra *en lugar de* la caja (después de su carga: sala B a 40, recogida, sellado, laboratorio a 65, entregada a 67), pero la caja vale 5 y la muestra 4, así que el día que sigue los pesos renuncia a la muestra y a la comida: coste total 6 para este orden. El plan de la Figura 20.4 no lo hace mejor: su muestra se entregaría de 97 a 99, con la recepción cerrada desde 90, así que el nuevo `deliver` la rechaza; sin la muestra, su coste sería 4 + 1 = 5. Una hora de cierre ha sacado la muestra de la ronda, la lleve quien la lleve, y los pesos dicen que ella, y no los medicamentos, es la pérdida correcta.

Solución 20.5.2. El problema `courier-fleet-day-within` es la prueba trivial sin `(at 80 (not (in-time meds1)))` y con dos líneas más:

```
(:requirements :constraints)
...
(:constraints (within 80 (delivered meds1))))
```

El hecho `(in-time meds1)` se queda en el estado inicial, porque `deliver` sigue pidiéndolo; ahora simplemente es siempre cierto. Kairos devuelve el mismo plan que para `p01`, con makespan 67,005. De los literales temporizados de la ronda, los cuatro plazos se pueden escribir con `within`: el urgente como restricción estricta, y los tres de 100 también, pero solo *dentro* de las preferencias, porque una entrega rutinaria es un deseo: `(preference meds2-done (within 100 (delivered meds2)))` en la sección de restricciones, como mostró el Capítulo 15. Los dos literales de la farmacia no. La apertura es un hecho que se hace cierto en un instante y que una acción necesita *durante todo* su desarrollo; `within` habla del instante en que una condición se alcanza por primera vez, no de cuándo puede ejecutarse una acción. Una ventana es un horario del mundo, y los horarios son literales temporizados. Es el veredicto del Capítulo 13, encontrado de nuevo en nuestros propios ficheros.

Solución 20.5.3. El problema `courier-fleet-day-meal` es el día de `r2` (`p06`) con `(* 6 (is-violated meal-done))`.

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-meal.
    pddl --timeout 3
...
; better plan 3
; cost = 5
; makespan = 70.008
0.000: (charge r2 dock) [14.000]
14.001: (move r2 dock lobby) [4.000]
18.002: (pick r2 meal1 lobby) [2.000]
20.003: (move r2 lobby corridor) [10.000]
30.004: (move r2 corridor ward-b) [12.000]
42.005: (deliver r2 meal1 ward-b) [2.000]
44.006: (move r2 ward-b corridor) [12.000]
56.007: (move r2 corridor lobby) [10.000]
66.008: (move r2 lobby dock) [4.000]
```

`r2` entrega la comida y vuelve a casa; renuncia a la caja rutinaria de medicamentos, coste 5. No puede hacer las dos cosas, como se argumentó en el capítulo, y ahora la comida es la más valiosa de las dos.

Lo que se dijo después de la ficha fue que los pesos son el orden de la responsable convertido en números. Ya no lo son: ella dijo «los medicamentos, luego la muestra, luego la comida», y con los pesos 5, 4 y 6 la comida va primero. Las dos frases para ella: *«Con un 6 en la comida, una mañana en la que un robot solo pueda hacer una entrega más, se llevará la comida y dejará los medicamentos rutinarios para la ronda siguiente. Si no es eso lo que quieres, la comida tiene que quedarse por debajo de 5, y podemos decirle a la cocina que solo gana a la muestra.»*

La lección del Capítulo 14 se mantiene a tamaño real: un peso no es un control de volumen para un deseo, es una afirmación sobre todos los demás.

Sección 20.6 — Versión 4: la batería es cosa del mundo

Solución 20.6.1. El dominio `courier-fleet-plus-sleep-domain.pddl` añade las dos acciones del Capítulo 17, con una condición más: solo se apaga un robot ocioso, de modo que ningún movimiento lo corta su propio robot.

```
(:action power-off
  :parameters (?r - robot)
  :precondition (and (powered ?r) (idle ?r))
  :effect (not (powered ?r)))

(:action power-on
  :parameters (?r - robot)
  :precondition (and (not (powered ?r)) (> (battery ?r) 0))
  :effect (powered ?r))
```

No cambia nada más: los procesos `drain` y `motor-drain` ya tienen `(powered ?r)` como condición, así que un robot apagado no se descarga. Esa es la ventaja de haber dado a cada cambio su causa. El plan a mano, `courier-fleet-plus-p04-sleep.plan`, apaga a 0, enciende a 30,1, recoge y entrega:

```
$ kairos validate courier-fleet-plus-sleep-domain.pddl courier-
    fleet-plus-p04.pddl courier-fleet-plus-p04-sleep.plan
plan valid, makespan 58.2
```

El robot conserva sus 32 unidades durante la espera y llega con 4 (un poco menos: está encendido una décima de segundo antes de recoger). El planificador, con los mismos ficheros, devuelve planes de 114,9 y 94,7 que no usan el interruptor: van a la base de carga y cargan, como en el capítulo.

No encuentra nuestro plan, que es 36 segundos más corto y tiene menos acciones (siete frente a diez). Solo podemos suponer por qué. Desde el estado inicial, el camino hacia nuestro plan empieza con `power-off`, una acción que hace inaplicables todas las demás acciones del robot, cosa que no le gusta a ninguna heurística; y el plan tiene que dejar pasar después exactamente el tiempo justo sin que ocurra nada. A alguien que concluya que el interruptor no sirve para nada: la respuesta del planificador muestra que existe un plan sin el interruptor, y nada sobre los planes con él. El validador aceptó uno mejor. Un planificador es una fuente de planes, y para un modelo híbrido el Capítulo 18 ya nos dijo que escribiéramos el plan que esperamos y lo validáramos. Si el plan más corto importa,

el modelo puede ayudar a la búsqueda, por ejemplo con un deseo de que los robots estén apagados mientras esperan, o convirtiendo la propia espera en una acción.

Sección 20.7 — Del plan a la ejecución

Solución 20.7.1. Las decisiones, por orden.

¿Qué hacemos con `r2`? A 30 está empezando a recoger la caja urgente. Molestarlo pondría en riesgo el primer objetivo estricto de la ronda. Dejamos que siga con su plan y planificamos solo para `r1`: la descomposición del capítulo, ahora impuesta por la ejecución.

El estado de `r1`. Está en la sala B, libre y ocioso. Ha recorrido $4 + 22 = 26$ metros: batería 74.

El calendario. El instante 0 del problema nuevo es el 30 de la ronda. La farmacia ya ha abierto: ese literal temporizado está en el pasado y se convierte en el hecho normal `(is-open pharmacy)`. Cierra dentro de 60, y la ronda acaba dentro de 70.

Los objetivos. La petición nueva es urgente, así que es un objetivo estricto con el final de la ronda como plazo. La caja rutinaria era un deseo y lo sigue siendo, con su peso; y lo mismo la vuelta a casa. Debe decidir el planificador, no nosotros, si la caja todavía cabe.

```
    (item-at sample2 ward-b) (destination sample2 lab)
    (item-at meds2 pharmacy) (destination meds2 ward-a)
    (is-open ward-b) (is-open pharmacy)
    (at 60 (not (is-open pharmacy)))
    (in-time sample2) (in-time meds2)
    (at 70 (not (in-time sample2)))
    (at 70 (not (in-time meds2))))
  (:goal (and
    (delivered sample2)
    (preference meds2-done (delivered meds2))
    (preference r1-home (robot-at r1 dock))))
  (:metric minimize (+ (* 5 (is-violated meds2-done))
                       (* 1 (is-violated r1-home)))))
```

```
$ kairos courier-fleet-day-domain.pddl courier-fleet-day-
    replan.pddl --timeout 3
; cost = 6
; makespan = 27.004
...
; better plan 2
; cost = 5
; makespan = 49.007
0.000: (pick r1 sample2 ward-b) [2.000]
2.001: (move r1 ward-b corridor) [12.000]
14.002: (seal r1 sample2) [3.000]
17.003: (move r1 corridor lab) [8.000]
25.004: (deliver r1 sample2 lab) [2.000]
27.005: (move r1 lab corridor) [8.000]
35.006: (move r1 corridor lobby) [10.000]
45.007: (move r1 lobby dock) [4.000]
```

La muestra está en el laboratorio 27 segundos después de la petición, a 57 de la ronda. Se renuncia a la caja rutinaria: desde el laboratorio, la farmacia está a 24 metros; el robot recogería a 53, a tiempo, y llegaría a la sala A a 81 del reloj nuevo, once segundos después del final de la ronda. Así que `r1` se va a casa, con $74 - 42 = 32$ de batería, y el coste es 5. La petición nueva le ha costado a la ronda sus medicamentos rutinarios, y la responsable debería oírlo con esas palabras.

Conflictos con `r2`. Suma 30 a cada instante y coloca el plan sobre la

Figura 20.4. `r1` entrega en el laboratorio de 55 a 57; `r2` llega allí a 97. Los dos usan el pasillo, cosa que nuestro modelo permite. El único recurso compartido del dominio es el cargador, y ninguno de los dos planes lo usa después de 30. Para validar los dos planes juntos hace falta un problema conjunto para el instante 30, y ese problema tiene que describir a `r2` en mitad de su recogida; la manera honesta más sencilla es empezar el problema conjunto a 32, cuando la recogida ha terminado, con las posiciones y baterías que predicen los dos planes.

¿Podría haber ayudado `r2`? Después de la caja urgente recoge la primera muestra de la sala B y está en el laboratorio a 99; no le sobra ni un segundo, y la caja rutinaria se pierde se le pida a quien se le pida. Hay algo que la solución da por supuesto, y un sistema real no debe hacerlo: planificar ha llevado tiempo. Si el planificador necesita tres segundos, el problema debería escribirse para 33 y no para 30; nuestro plan tiene más de cuarenta segundos de holgura en su objetivo estricto y sobrevive.

Sección 20.10 — El método MODEL: una historia, cuatro modelos

Solución 20.10.1. *Preguntas para la autora.* ¿Puede una carretilla llevar dos palés? (No.) ¿Se puede dejar un palé en algún sitio que no sea el suyo? (No: la misma decisión que con `deliver`.) ¿Es «delante de cada estantería» un sitio para una carretilla? (Sí.) ¿Pueden cruzarse dos carretillas entre el pasillo y una estantería? (No: eso es lo que significa estrecho.) ¿Tienen que acabar las carretillas en algún sitio? (No se dice: ningún objetivo sobre ello.) ¿Batería? (Se enchufan de día; fuera del alcance.)

Ficha del modelo. Historia: dos carretillas colocan los palés que esperan en el muelle, cada uno en su estantería. Agentes: las carretillas, a la vez. Éxito: todos los palés colocados. Alcance, dentro: los sitios y sus conexiones, qué sitios son estrechos, dónde están las carretillas y los palés; fuera: la batería, la altura de las estanterías, las personas. Supuestos: un palé por carretilla; en un sitio estrecho cabe una carretilla; en el muelle caben las que sean; en la versión temporal, 8 segundos por trayecto y 5 por carga o colocación.

Versión clásica (`warehouse-domain.pddl`). La idea nueva es la fila «un sitio estrecho en el que no hay nadie», `(clear ?l)`: un hecho sobre una ausencia, mantenido con efectos de adición y de borrado, el cerrojo de recurso del Capítulo 7 con un cerrojo por sitio. Entrar en un sitio amplio y entrar en uno estrecho son dos acciones, `drive` y `squeeze`, lo que mantiene el dominio en `:strips`; una sola acción con un efecto condicional sería la otra codificación razonable.

```
  ; Into a wide place: always possible.
  (:action drive
    :parameters (?f - forklift ?from ?to - place)
    :precondition (and (lift-at ?f ?from) (linked ?from ?to)
                       (wide ?to))
    :effect (and (not (lift-at ?f ?from)) (lift-at ?f ?to)
                 (clear ?from)))

  ; Into a narrow place: it must be clear, and it is taken.
  (:action squeeze
    :parameters (?f - forklift ?from ?to - place)
    :precondition (and (lift-at ?f ?from) (linked ?from ?to)
                       (clear ?to))
    :effect (and (not (lift-at ?f ?from)) (lift-at ?f ?to)
```

```
                (not (clear ?to)) (clear ?from)))
```

Pruebas. Típica (`p01`): dos palés, uno por estantería; ocho acciones, y el plan manda una carretilla hasta su estantería antes de que la otra entre en el pasillo. Trampa (`p02`): `f1` delante de la estantería A y `f2` en el pasillo tienen que intercambiar sus sitios. El plan tentador son dos acciones, cada una atravesando a la otra; el modelo devuelve seis, en las que `f2` sale marcha atrás hasta el muelle, `f1` sale detrás de ella, y vuelven a entrar en el orden contrario. Sin plan (`p03`): el mismo intercambio con una tercera carretilla delante de la estantería B y el muelle cortado; `unsolvable`, y la nota explica que ningún sitio estrecho puede quedar nunca libre. Una prueba trivial es un palé y una carretilla.

Lo que la versión clásica no puede decir: cuánto dura el trabajo de la noche, y qué significa «caber» durante un trayecto. En una secuencia, una carretilla está en un sitio y luego en el siguiente. En el tiempo, está entre los dos durante 8 segundos.

Versión temporal (`warehouse-time-domain.pddl`). La decisión está en `squeeze`, y es la pregunta del Capítulo 10 sobre dónde va un efecto: el sitio al que se entra se ocupa *al empezar* el trayecto, y el sitio que se deja solo se devuelve *al terminar*. Durante el trayecto, la carretilla ocupa los dos.

```
  ; The place entered is taken at the start; the place left is
  ; given back only at the end.
  (:durative-action squeeze
    :parameters (?f - forklift ?from ?to - place)
    :duration (= ?duration 8)
    :condition (and (at start (ready ?f))
                    (at start (lift-at ?f ?from))
                    (at start (linked ?from ?to))
                    (at start (clear ?to)))
    :effect (and (at start (not (ready ?f)))
                 (at start (not (lift-at ?f ?from)))
                 (at start (not (clear ?to)))
                 (at end (lift-at ?f ?to))
                 (at end (clear ?from))
                 (at end (ready ?f))))
```

Con los dos efectos al revés (liberar el sitio antiguo al empezar, ocupar el nuevo al terminar), dos carretillas podrían salir a la vez cada una hacia el sitio de la otra, y una tercera podría entrar en un sitio al que alguien está a punto de llegar. La ronda de dos palés dura 42,005: las dos cargan a la vez en el muelle, `f2` entra, y `f1` entra en el pasillo a 21,003, un acontecimiento después de que `f2` lo haya dejado camino de su estantería. Compárala con el plan clásico: las mismas ocho acciones, y ahora el orden entre las dos carretillas lo impone el pasillo y se ve en los instantes.

Paso L. La alternativa a un cerrojo por sitio es un único cerrojo para toda la zona estrecha (una sola carretilla más allá del muelle cada vez). Es más pequeño y no tiene bloqueos mutuos que explorar en la búsqueda, y pierde los planes en los que una carretilla trabaja en la estantería A mientras la otra está en la estantería B: con dos estanterías, ese es todo el beneficio de la segunda carretilla. Nos quedamos con el cerrojo por sitio. Las versiones siguientes seguirían al capítulo: un calendario si los camiones salen a horas fijas, deseos si no todos los palés tienen que colocarse esta noche.

Bibliografía

Las referencias se listan por orden alfabético.

Bacchus, F., & Kabanza, F. (2000). Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, *116*(1–2), 123-191.

Benton, J., Coles, A. J., & Coles, A. I. (2012). Temporal Planning with Preferences and Time-Dependent Continuous Costs. *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 2-10.

Bonet, B., & Geffner, H. (2001). Planning as Heuristic Search. *Artificial Intelligence*, *129*(1–2), 5-33.

Bylander, T. (1994). The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, *69*(1–2), 165-204.

Coles, A. J., Coles, A. I., Fox, M., & Long, D. (2010). Forward-Chaining Partial-Order Planning. *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, 42-49.

Coles, A. J., Coles, A. I., Fox, M., & Long, D. (2012). COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research*, *44*, 1-96.

Corrêa, A. B., Pommerening, F., Helmert, M., & Francès, G. (2020). Lifted Successor Generation Using Query Optimization Techniques. *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, 80-89.

Cushing, W., Kambhampati, S., Mausam & Weld, D. S. (2007). When is Temporal Planning Really Temporal? *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, 1852-1859.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal Constraint Networks. *Artificial Intelligence*, *49*(1–3), 61-95.

Edelkamp, S., & Hoffmann, J. (2004). *PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition* (inf. téc. N.º 195). Institut für Informatik, Albert-Ludwigs-Universität Freiburg.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, *2*(3–4), 189-208.

Fox, M., & Long, D. (2003). PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, *20*, 61-124.

Fox, M., & Long, D. (2006). Modelling Mixed Discrete-Continuous Domains for Planning. *Journal of Artificial Intelligence Research*, *27*, 235-297.

Gazen, B. C., & Knoblock, C. A. (1997). Combining the Expressivity of UCPOP with the Efficiency of Graphplan. *Proceedings of the 4th European Conference on Planning (ECP)*, 221-233.

Geffner, H., & Bonet, B. (2013). *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool.

Gerevini, A., & Long, D. (2005). *Plan Constraints and Preferences in PDDL3* (inf. téc. N.º RT 2005-08-47). Department of Electronics for Automation, University of Brescia.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., & Dimopoulos, Y. (2009). Deterministic Planning in the Fifth International Planning Competition: PDDL3 and Experimental Evaluation of the Planners. *Artificial Intelligence*, *173*(5–6), 619-668.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Ghallab, M., Nau, D., & Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press.

Haslum, P., Lipovetzky, N., Magazzeni, D., & Muise, C. (2019). *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool.

Helmert, M. (2002). Decidability and Undecidability Results for Planning with Numerical State Variables. *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS)*, 44-53.

Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, *26*, 191-246.

Helmert, M. (2009). Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, *173*(5–6), 503-535.

Hoffmann, J. (2003). The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research*, *20*, 291-341.

Hoffmann, J., & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, *14*, 253-302.

Howey, R., Long, D., & Fox, M. (2004). VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 294-301.

Keyder, E., & Geffner, H. (2009). Soft Goals Can Be Compiled Away. *Journal of Artificial Intelligence Research*, *36*, 547-556.

Kovacs, D. L. (2011). *BNF Definition of PDDL 3.1* [Unpublished manuscript from the IPC-2011 website].

Lipovetzky, N., & Geffner, H. (2012). Width and Serialization of Classical Planning Problems. *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, 540-545.

Martín, F., Ginés Clavero, J., Matellán, V., & Rodríguez, F. J. (2021). PlanSys2: A Planning System Framework for ROS2. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9742-9749.

Martín Rico, F., Aguado González, E., Moreno Olivo, F. M., Pérez Rodríguez, R., Guerrero Hernández, J. M., & Rodríguez Lera, F. J. (2026). Kairos: Semantics-Aware Orchestration of Planning Engines across Classical, Numeric, Temporal and Hybrid PDDL. https://arxiv.org/abs/2610.39454

McDermott, D. (2000). The 1998 AI Planning Systems Competition. *AI Magazine*, *21*(2), 35-55.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998). *PDDL: The Planning Domain Definition Language* (inf. téc. N.º CVC TR-98-003/DCS TR-1165). Yale Center for Computational Vision y Control.

Nebel, B. (2000). On the Compilability and Expressive Power of Propositional Planning Formalisms. *Journal of Artificial Intelligence Research*, *12*, 271-315.

Pednault, E. P. D. (1989). ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 324-332.

Pnueli, A. (1977). The Temporal Logic of Programs. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, 46-57.

Richter, S., & Westphal, M. (2010). The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, *39*, 127-177.

Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th). Pearson.

Scala, E., Haslum, P., Thiébaux, S., & Ramírez, M. (2016). Interval-Based Relaxation for General Numeric Planning. *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI)*, 655-663.

Thiébaux, S., Hoffmann, J., & Nebel, B. (2005). In Defense of PDDL Axioms. *Artificial Intelligence*, *168*(1–2), 38-69.

Índice alfabético

abstracción, 146
acciones paralelas, 232
acción, 4, 23
  aplicable, 24
  aplicación, 24
  en PDDL+, 389
acción aplicable, 50
acción conjunta, 255
acción de coste cero, 202
acción durativa, 222
acción envolvente, 251
acción fusionada, 157
acción instanciada, 24, 50
acción sin vuelta atrás, 155
acontecimiento, 227
ADL, 127, 133
alcanzabilidad, 301
alternancia de turnos, 153
anomalía de Sussman, 36
antipatrón, 158
  borrado olvidado, 158
  catálogo, 446
  gemelos
    desacompasados, 159
  objetivo en el estado inicial, 158
  parámetro ocioso, 159
autonomía basada en modelos, 3
axioma, 301

batería de pruebas, 13, 30
beneficio neto, 329
borrado olvidado, 158
búsqueda, 57
búsqueda anytime, 61, 203
búsqueda ciega, 58
búsqueda en anchura, 58
búsqueda hacia delante, 57
búsqueda voraz primero el mejor, 58

cadena de sucesores, 151
calidad del plan, 197
callejón sin salida, 66, 155
cambio continuo, 270, 272
cambio discreto, 270, 272
cambio lineal, 272
cambio no lineal, 410
capacidad, 152, 179
carga, 258
caso de estudio, 472
cerrojo de recurso, 155
claridad, 15
coloreado de mapas, 373
comparación, 173
Competición Internacional de Planificación, 10
  IPC-5, 337
compilación, 110
  de efectos condicionales, 133
complejidad, 57
comportamiento de Zenón, 418
concurrencia, 232
concurrencia requerida, 249
condiciones, 100
condición numérica, 173
constantes, 77, 88
contar sin números, 151
coordinación, 249
costes de acción, 197
cronograma, 223

depuración, 160
  procedimiento ampliado, 448
despacho, 496
  disparado por eventos, 496
  disparado por tiempo, 496
diagnósticos, 514
documentación, 462
dominio, 9
dominio de la cafetera, 261
dominio de la cocina, 200, 237
dominio de la lavandería, 451
dominio del ascensor, 238
dominio del camarero, 145
dominio del camión, 180
dominio del circuito, 304
dominio del coche, 392
dominio del generador, 279
dominio del horno, 282
dominio del invernadero, 277
dominio del maletín, 126
dominio del róver, 356
dominio del satélite, 311, 335
dominio del taller, 357
dominio del turista, 333
dominio del viaje por carretera, 210
dominio logistics, 84
dron, 414
duración
  calculada, 230
  elegida, 258

efecto, 4, 24, 119
  de adición, 24
  de borrado, 24
efecto condicional, 119, 121
  compilación, 133
  semántica, 122
  universal, 124
efecto continuo, 270, 273
  planificación, 284
efecto de adición, 50
efecto de borrado, 50
efecto numérico, 173
efecto secundario, 130
efecto universal, 119, 123
either, 86
ejecución del plan, 496
ejecutante de acción, 500
espacio de estados, 52
  infinito, 187
  tamaño, 56
estado, 4, 23, 50
  como conjunto de hechos, 23
estado alcanzable, 52
estado inicial, 4, 25
estado sucesor, 50
estratificación, 306
evento, 383
  cascada, 416
  debe hacer falsa su condición, 388
  disparo infinito, 418
exclusión mutua, 155, 234
expansión, 57
explosión combinatoria, 56
expresión S, 21

Fast Downward, 59
FF, 59
ficha del modelo, 13, 473

plantilla, 462
fidelidad, 14
fluente de objeto, 367
como término, 370
fluente inicial temporizado, 313
fluente numérico, 170
forall
con imply, 109
función, 171
función estática, 178

gemelos desacompasados, 159
granularidad, 157
granularidad de las acciones, 157
gripper, 37
grupo mutex, 54, 151, 374
guía, 15
guía de decisión, 441

hecho, 23, 50
hecho dinámico, 27, 39
hecho estático, 27, 39, 54, 66
hecho redundante, 137, 159
hechos espejo, 150
Helmert, Malte, 188
heurística, 58
puntos ciegos, 61
hipótesis del mundo cerrado, 23, 50
HSP, 59
huecos, 152

igualdad, 77, 89
incertidumbre, 502
indecidibilidad, 187
instanciación, 52
de cuantificadores, 110
tamaño, 65
invariante, 150, 228, 367
invariante de a lo sumo uno, 150
IPC, *véase* Competición Internacional de Planificación

jarras de agua, 182
jerarquía de tipos, 81

Kairos, 11, 510
–debug, 516
–dump, 53, 516
–no-anytime, 63
–target-cost, 63
–timeout, 63
-v, 53, 516
--no-anytime, 205
--target-cost, 205
check, 26, 30, 514
códigos de salida, 448, 514
instalación, 510
motores, 254, 512
opciones, 512
PDDL+, 399
Python, 517
solve, 511
validate, 30, 515
órdenes, 511
kairos check, 80

LAMA, 59
Lights Out, 124
lista abierta, 57
lista de comprobación, 438
literal inicial temporizado, 298, 307, 308
longitud del plan, 197
límite de tiempo, 63
límite estricto, 212
lógica temporal lineal, 354

magnitud, 170
makespan, 226
meseta, 60
modelado, 145, 436
modelo, 3
versiones, 451, 474
monitor, 358
mundo de bloques, 35, 108
above, 304
con fluentes de objeto, 372
mundo del mensajero, 3, 26
batería, 286
cadena de frío, 397
carro, 119, 127
courier-fleet, 475
courier-fleet-day, 485
courier-fleet-plus, 491
courier-fleet-time, 480
en PDDL+, 394
flota, 472
laboratorio estéril, 130
método MODEL, 13, 38
calidad del plan, 213
cambio continuo, 290
caso de estudio, 502
cinco pasos, 436
completo, 436
condiciones, 111
consolidado para la planificación clásica, 161
coordinación, 263
criterios, 14
fluentes de objeto, 376
listas de comprobación, 438
literales iniciales temporizados, 314
magnitudes, 188
paso D, 13, 135
paso E, 13, 30, 68, 160, 448
paso L, 13, 68
paso M, 13, 472
paso O, 13
predicados derivados, 314
preferencias, 338
procesos y eventos, 401
restricciones, 359
sistemas híbridos, 428
tiempo, 240
tipos, 92
versiones y documentación, 464
métrica, 197
ponderada, 209
sobre fluentes, 206

niveles de modelado, 423
no plan found, 449

objetivo, 4, 26, 51
cuantificado, 109
negativo, 103
objetivo blando, 323
objetivo en el estado inicial, 158
objetivo negativo, 103
objeto, 22
observabilidad parcial, 502
over all
condiciones numéricas, 274, 276

par inverso, 155
parámetro, 24, 65
parámetro ocioso, 159
pasos de tiempo, 399
límites, 425
patrón, 145
acción fusionada, 157
acción sin vuelta atrás, 155
cadena de sucesores, 151
catálogo, 444
cerrojo de recurso, 155

hechos espejo, 150
huecos, 152
invariante de a lo sumo uno, 150
par inverso, 155
para el control, 152
para el estado, 148
para las acciones, 155
tabla estática, 149
testigo de fase, 153
testigo de turno, 153
patrón inicio-proceso-parada, 390
PDDL, 9
:action, 23
:action-costs, 198
:adl, 133
:condition, 224
:conditional-effects, 121
:constants, 88
:constraint, 408
:constraints, 347
en el dominio, 355
:continuous-effects, 272
:derived, 299
:derived-predicates, 299
:disjunctive-preconditions, 104
:duration, 224
:duration-inequalities, 258
:durative-action, 224
:durative-actions, 224
:effect, 24
:equality, 89
:event, 387
:existential-preconditions, 105
:fluents, 171
:functions, 171
con valor objeto, 368
:goal, 26
:init, 25
:metric, 198, 231
:negative-preconditions, 100
:numeric-fluents, 171
:object-fluents, 368
:objects, 25
:parameters, 24
:precondition, 24
:predicates, 22
:preferences, 324
:process, 384
:quantified-preconditions, 107
:requirements, 24
:strips, 24
:time, 384
:timed-initial-fluents, 313
:timed-initial-literals, 307
:types, 78
:typing, 78
:universal-preconditions, 107
=, 89
?duration, 224, 259
#t, 272
1.2, 519
2.1, 274, 521
2.2, 298, 522
3.0, 524
3.1, 524
always, 347
always-within, 350
assign, 173
fluente de objeto, 368
at end, 228, 348
at start, 228
at-most-once, 349
decrease, 173
define, 22
either, 86
elegir una característica, 441
exists, 105
forall, 107
con preferencias, 330
forall (en efectos), 123
hold-after, 350
hold-during, 350
imply, 104
increase, 173
is-violated, 324
maximize, 208
not, 100
object, 82
or, 104
over all, 228
preference, 324
referencia rápida, 518
requisitos, 518
scale-down, 183
scale-up, 183
sintaxis, 519
sometime, 348
sometime-after, 349
sometime-before, 349
total-cost, 198
total-time, 231
undefined, 371
versiones, 9
versión 3.0, 337, 377
versión 3.1, 376, 377
when, 121
within, 350
PDDL+, 383, 525
caso de estudio, 491
con acciones durativas, 420
Pednault, Edwin, 127
pelota que rebota, 390
percepción, 502
plan, 2, 26, 51
formato, 513
secuencial, 222
temporal, 226
validación, 12, 515
plan relajado, 59
plan óptimo, 203
planificación, 2
aplicaciones, 8
definición, 2
frente a la programación, 6
frente al aprendizaje, 6
planificación clásica, 5
supuestos, 5
planificación de tareas, 7
planificación numérica, 170
planificación satisfactoria, 61
planificación temporal, 222
planificación óptima, 61
planificador, 3, 49
soporte de PDDL, 527
PlanSys2, 498
plazo, 309, 310
within, 316
portabilidad, 15
por característica, 443
precondición, 4, 24, 50, 100
frente a condición de un efecto, 123
precondición disyuntiva, 104
precondición existencial, 105
precondición negativa, 100
precondición universal, 107
predicado, 22
predicado básico, 298
predicado de cerrojo, 263
predicado de ocupación, 234
predicado de tipo, 28, 77
predicado derivado, 298, 299

planificación, 306
recursivo, 301
restricciones, 300
preferencia, 323
bajo forall, 330
compilación, 336
en precondiciones, 331
peso, 327
elección, 329
sin métrica, 340
sobre el estado final, 328
sobre una restricción, 355
violada, 325
y costes de acción, 329
presupuesto, 212
problema, 9, 25
problema irresoluble, 13, 33, 63
proceso, 383
interacción, 412
prueba de la trampa, 396
PSPACE, 57
puerta, 251
puzle deslizante, 55

recuento, 184
recurso, 234
red temporal simple, 261
regla (predicado derivado), 300
regla del marco, 39
regla recursiva, 302
reglas estratificadas, 306
relación simétrica, 150
relajación, 59
relajación de borrados, 59
replanificación, 497
representación de dominio finito, 374
requisitos, 10
restricción
sobre el plan, 345
restricción blanda, 355
restricción de trayectoria, 345
compilación, 358
semántica, 353
restricción global, 408
ritmo, 272
rompecabezas de los cofres, 106
ROS 2, 498

SAS+, 375
semántica, 49
separación épsilon, 227
Shakey, 8
simetría, 67
sistema híbrido, 383
modelado, 408
sobresuscripción, 323
soluciones a los ejercicios, 529
STN, 261
STRIPS, 8, 21
subtipo, 81
suma ponderada, 209
supertipo, 81
sótano de las cerillas, 249

tabla de acciones, 13, 28
tabla de estado, 13, 27
tabla estática, 149
tamaño (criterio), 14
temporalmente expresivo, 254
temporalmente simple, 254
termostato, 417
testigo, 152
testigo de fase, 153
testigo de turno, 153
tipos, 77, 78
torres de Hanói, 55
trayectoria, 347

validación
en tiempo continuo, 400
valor indefinido, 185, 371
variable de dominio finito, 54
variable de estado, 367, 374
velocidad, 230
ventana temporal, 309

árbol de comportamiento, 499