Preferencias y restricciones: PDDL 3

Planificación y PDDL, Parte V (Capítulos 14–16)

Francisco Martín Rico

Intelligent Robotics Lab, URJC



Índice

1. Objetivos blandos y preferencias
2. Restricciones sobre todo el plan
3. Fluentes de objeto
4. Resumen

1 Objetivos blandos y preferencias

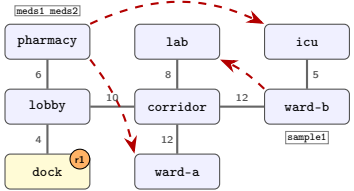
2 Restricciones sobre todo el plan

3 Fluentes de objeto

4 Resumen

Más deseos que batería

- Mensajero con tipos + batería: un desplazamiento cuesta sus metros
- Batería de 100, hoy sin `charge`
- Lista de la mañana: `meds1` a la sala A, `meds2` a la UCI, `sample1` al laboratorio



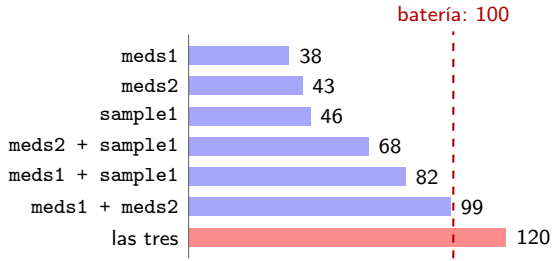
Las tres entregas como un único objetivo:

```
(:goal (and (item-at meds1 ward-a)
            (item-at meds2 icu)
            (item-at sample1 lab)))
```

```
$ kairos courier-battery-domain.pddl
    courier-battery-p01.pddl
...
kairos: unsolvable (exhausted)
```

Correcto e inútil: el robot pasa la mañana en la base de carga

Sobresuscripción



Batería necesaria para cada conjunto de entregas, desde la base de carga

- Dos entregas cualesquiera caben; las tres, 120
- **Sobresuscrito**: más objetivos de los que caben
- El caso normal: muchas peticiones, un robot
- Borrar objetivos a mano: el modelador hace de planificador

La pregunta del hospital

¿Qué entregas hacer con la batería que hay?

Una preferencia en el objetivo

`courier-battery-p02.pddl`: la misma lista como deseos (requisito `:preferences`)

```
(: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))))
```

- **Preferencia**: una condición con nombre, `(preference name C)`
- En un objetivo: un **objetivo blando**; los demás son **objetivos estrictos**
- Violarla no incumple ninguna regla: solo cuesta lo que diga la métrica
- `(is-violated meds-a)`: 0 si se cumple, 1 si no
- La métrica: las entregas *no* hechas, minimizadas

La búsqueda *anytime* en acción

```
$ kairos courier-battery-domain.pddl
    courier-battery-p02.pddl
; cost = 3
; better plan 2
; cost = 2
(move r1 dock lobby)
...
(drop r1 meds1 ward-a)
; better plan 3
; cost = 1
(move r1 dock lobby)
...
(drop r1 meds1 ward-a)
...
(drop r1 sample1 lab)
```

- El plan 1 cuesta 3 y *no tiene acciones*: con solo objetivos blandos, no hacer nada es un plan, el peor
- El plan 2 entrega `meds1`: coste 2
- El plan 3 añade `sample1`: coste 1, y la búsqueda se detiene sola
- El coste impreso es el valor de la métrica

Válido no es lo mismo que bueno

Un plan escrito a mano que solo entrega `meds1`:

```
$ kairos validate courier-battery-domain.pddl courier-battery-p02.pddl
    courier-battery-p03-lazy.plan
plan valid, makespan 7, metric 2
```

- Validez: precondiciones y objetivos estrictos. Todo lo relativo a los deseos vive en la métrica

Paso M: obligatorio o deseado

La línea de *Éxito* de la ficha del modelo se divide en dos: **Obligatorio** (un plan tiene que conseguirlo para ser siquiera aceptable) y **Deseado** (se consigue si es posible, cada deseo con su valor). Prueba: imagina el plan que deja fuera la condición. ¿*Inservible*? Obligatorio. ¿Solo *peor*? Un deseo.

Lo que vale cada deseo

La supervisora de enfermería: la UCI va primero. p03 solo cambia la métrica:

```
(:metric minimize (+ (* 3 (is-violated meds-a))
                     (* 5 (is-violated meds-icu))
                     (* 1 (is-violated sample-lab)))))
```

- Un **peso**: el precio de violar esa preferencia
- Kairos imprime costes 9, 4, 3 y 1: cuatro filas de la tabla
- Último plan: los dos medicamentos; llega a la UCI con 1 unidad de sobra

Entregas hechas	Batería	(1, 1, 1)	(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**

Pesos iguales (p02): tres conjuntos empatan a 1. Con pesos (p03): solo uno es el mejor

Un deseo sobre el robot, y un objetivo que debe cumplirse

p04: terminar en la base de carga, peso 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)))))
```

p05: el laboratorio debe tener su muestra

```
(: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)))))
```

- El mejor, coste 3: UCI y laboratorio, de vuelta en la base con 10 unidades
- Los dos medicamentos, varado en la UCI: $1 + 4 = 5$
- El plan vacío cuesta 9, no 13
- Objetivos estrictos y blandos se mezclan en un solo `and`
- Sin plan vacío; costes 8, 5 y 3
- El plan perezoso: `plan invalid`

Cómo elegir los pesos

- Solo importa la relación entre los pesos: (3, 5, 1) y (30, 50, 10) son el mismo problema
- Los objetivos estrictos quedan fuera del regateo

Tres patrones

- **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 ganar a uno grande
- **Prioridad estricta**: una clase superior gana a *cualquier número* de inferiores. Cada clase pesa más que la suma de todos los pesos por debajo: $5 > 3 + 1$

Deseos frente a esfuerzo

`courier-cost`: un desplazamiento suma sus metros a `total-cost`; `pick` y `drop` suman 1

```
(: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)))))
```

```
$ kairos courier-cost-domain.pddl
    courier-cost-p01.pddl
; cost = 80
; better plan 2
; cost = 60
(move r1 dock lobby)
...
(drop r1 meds1 ward-a)
```

- Pesos *en metros*: el medicamento vale un viaje de hasta 60
- Nada: $60 + 20 = 80$. Medicamento: $40 + 20 = 60$
- La comida suma 63 para ahorrar 20: mal negocio
- p02, la comida vale 100: las dos, coste 103

Beneficio neto

Valor conseguido frente al coste de conseguirlo: el planificador consigue todo lo que compensa

Una preferencia, muchas instancias

`courier-cost-p03.pddl`: dos medicamentos y una comida en la farmacia

```
(:goal (forall (?m - medicine)
         (preference meds-icu (item-at ?m icu))))
(:metric minimize (+ (total-cost)
                     (* 50 (is-violated meds-icu)))))
```

```
$ kairos courier-cost-domain.pddl
    courier-cost-p03.pddl
; cost = 100
; better plan 2
; cost = 95
(move r1 dock lobby)
...
(drop r1 meds2 icu)
```

- Un `forall` *alrededor* de una preferencia: una preferencia por medicamento, todas llamadas `meds-icu`
- (`is-violated` `meds-icu`): el *número* de medicamentos fuera de la UCI
- Plan vacío: $2 \times 50 = 100$
- Mejor: un medicamento, $45 + 50 = 95$; el segundo necesita 68 más para ahorrar 50
- La comida no es un `medicine`: no tiene preferencia

Dónde va el forall

	`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

- El mismo recuento siempre que un nombre se usa más de una vez: dos preferencias separadas llamadas `meds` dan a `(is-violated meds)` 0, 1 o 2
- Un nombre identifica una **clase** de deseos con un solo precio, no un deseo concreto

Deseos sobre el camino: preferencias en las precondiciones

`courier-lobby`: no cruzar el vestíbulo con una muestra

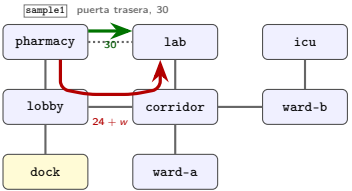
```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
    (preference clean-lobby
      (or (not (= ?to lobby))
          (not (exists (?s - sample) (carrying ?r ?s))))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (increase (total-cost) (distance ?from ?to))))
```

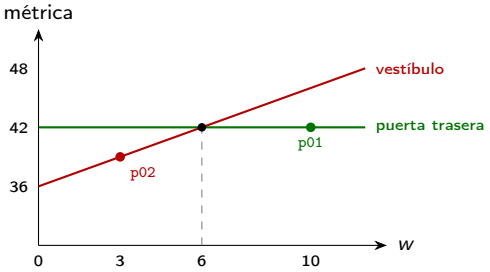
- Aplicable de todos modos; aplicarla sin la preferencia: una violación
- (`is-violated` `clean-lobby`): cuántas *veces*
- Un objetivo blando no puede decirlo: al final el robot no está en el vestíbulo

No es una precondición

Se incumple siempre que sale más barato, y gratis si la métrica no la menciona. Una regla que nunca debe incumplirse va sin `preference`: estricta y sin puerta trasera, Kairos responde `unsolvable`

El precio decide la ruta





- Los problemas abren una puerta trasera, 30 m; por el vestíbulo, 24 m
- La muestra debe (objetivo estricto) llegar al laboratorio
- p01, $w = 10$: puerta trasera, coste 42 (vestíbulo: $36 + 10 = 46$)
- p02, $w = 3$: por el vestíbulo, $36 + 3 = 39$
- Las dos rutas se cruzan en $w = 6$
- p03, sin puerta trasera, dos muestras: coste 106, paga dos violaciones

Otros dos mundos: el día de un turista

240 minutos, cuatro lugares, un autobús de 5 minutos entre dos sitios cualesquiera

```
(: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)))))
```

- Un objetivo estricto, deseos de distinto valor y un deseo de que algo *no* ocurriera
- `on-foot` cuesta lo mismo con un viaje que con cuatro: el último estado solo sabe `(used-bus)`

```
$ 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)
```

- Se salta el mercado (2), un viaje en autobús (1)

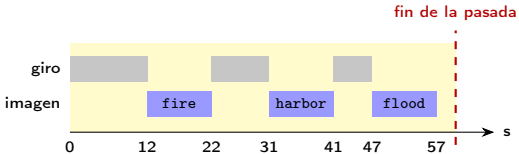
Otros dos mundos: un satélite con más peticiones que tiempo

- `slew` y `take-image` (10 s) durativas en una pasada de 60 s: `(at 60 (not (in-pass)))`
- Cinco peticiones, dos urgentes

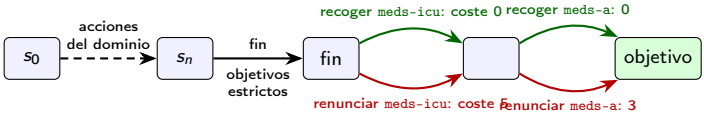
```
(:goal (and
  (forall (?t - urgent)
    (preference urgent-done (imaged ?t)))
  (forall (?t - routine)
    (preference routine-done (imaged ?t)))))
(:metric minimize (+ (* 5 (is-violated urgent-done))
                     (is-violated routine-done))))
```

- Tres blancos rutinarios: $5 > 3 \times 1$, prioridad estricta

- Kairos: costes 13, 3, 2
- Mejor plan: las dos imágenes urgentes, y `harbor` de camino de `fire` a `flood`
- Coste y *makespan* son ahora números distintos



Qué hace el planificador con una preferencia



- Objetivos blandos compilados en costes: una acción *fin* y, por cada preferencia, **recoger** (gratis si la condición se cumple) o **renunciar** (paga el peso)
- Kairos compila las preferencias del objetivo (también bajo `forall`) y hasta cuatro por acción en lo alto de una precondición (copias de la acción); el resto se *cuentan*: cada plan se valora con exactitud y solo se guardan los que mejoran
- La búsqueda parte del plan más barato de *encontrar*, a menudo el vacío; si una búsqueda secuencial termina sola, el último plan es `proven optimal`

El método MODEL: los deseos y su valor

Preguntas nuevas para el paso M (Mapear la situación)

- **¿Qué objetivos son obligatorios y cuáles son deseos?**
- **¿Cuánto vale cada deseo?** En qué unidad; frente a otros deseos o también frente al esfuerzo
- **¿Algún deseo trata del camino y no del final?**
- **¿Cuál es el plan si no se concede nada?** Si el plan vacío no es aceptable, falta un objetivo estricto

La ficha del modelo gana una tabla de *Deseos*: nombre, condición, peso, motivo del peso.

- Paso E: ejecuta sin métrica; después, la **prueba del precio**: sube cada peso hasta que el plan cambie; ¿tiene sentido?

Alternativas comparadas: no cruzar el vestíbulo con una muestra

	Precondición estricta	Preferencia	Coste de acción (efecto condicional)
Fidelidad	Pierde soluciones: sin puerta trasera, no hay plan	Exacta: permitido, a un precio	Exacta, los mismos planes y precios
Tamaño	No crece	Copias de `move`, acciones de recoger y renunciar	No crece
Guía	La mejor: se podan los movimientos prohibidos	Buena si se compila	Kairos no usa un coste condicional para guiar la búsqueda
Claridad	Una norma, que se lee como norma	Deseo en el dominio, precio en cada problema	Precio enterrado en el dominio
Portabilidad	Precondiciones negativas, disyuntivas	`:preferences`: Kairos, OPTIC, POPF, ENHSP	`:conditional-effects` + `:action-costs`

- Las tres: las mismas cinco acciones por la puerta trasera, coste 42. **Estricta**: incumplirla hace inservible el plan (seguridad, física, ley); un peso enorme no la sustituye
- **Preferencia**: aceptable a un precio, sobre todo si el precio cambia por problema
- **Coste de acción**: hay que evitar la acción en sí, siempre (un peaje)

Errores comunes (1)

Preferencias sin métrica

p02 sin su `:metric`:

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

Un plan vacío y ningún aviso. *Corrección*: toda preferencia aparece en la métrica. Lo mismo para una preferencia sin nombre

Un nombre que no es de ninguna preferencia

Lo detecta `kairos check`: ejecútalo 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]
...
invalid
```

La misma errata en el objetivo *y* en la métrica no es ningún error

Errores comunes (2)

Maximizar las violaciones

`maximize` pide el plan que más falla: el plan vacío, coste 3. *Corrección*: `minimize` una suma de violaciones; un beneficio es una constante menos las violaciones

Pesos en la unidad equivocada

Un peso de 3 en `courier-cost` significa «vale tres metros»: el robot nunca se mueve por él, y el plan es `proven optimal`. *Corrección*: la prueba del precio; escribe la unidad junto a cada peso

Quedarse con el primer plan

Con objetivos blandos, el primer plan suele ser el vacío:

```
$ kairos courier-battery-domain.pddl
    courier-battery-p03.pddl
    --no-anytime
; cost = 9
```

Corrección: deja correr la búsqueda *anytime* y quédate con el último plan, o usa `--target-cost`. Un límite de tiempo corto puede dar un plan lejos del mejor

1 Objetivos blandos y preferencias

2 Restricciones sobre todo el plan

3 Fluentes de objeto

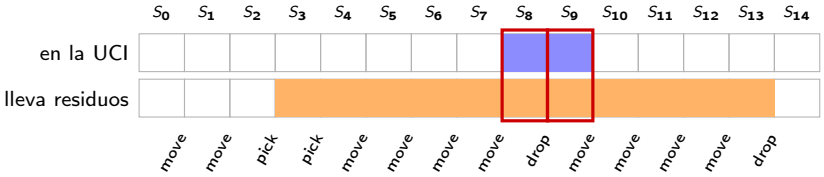
4 Resumen

Una regla que el plan incumple

```
$ 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 ward-b icu)
(drop r1 meds1 icu)
...
(drop r1 waste1 dock)
```

- `courier-rules`: una cesta (cualquier número de artículos), artículos `medicine`, `sample`, `waste`; `sanitize` limpia la pinza, `pick` la ensucia
- Medicamentos a la UCI, residuos a la base de carga, terminar en la base
- El plan más corto entra en la UCI con residuos
- Falta: *ningún robot está nunca en la UCI mientras lleva residuos*
- No es un objetivo; como precondiciones, queda dispersa por todas las acciones

Los estados de un plan



- Un plan de n acciones pasa por $S_0, S_1, \ldots, S_n$: su **trayectoria**, *con los dos extremos*
- Un objetivo solo mira S_{14}; la regla mira cada columna y falla en S_8, S_9
- **Restricción de trayectoria**: una condición sobre la trayectoria, en una sección `:constraints` (requisito `:constraints`)

always

`courier-rules-p02.pddl` = p01 + 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)))))))
```

- `(always C)`: `C` se cumple en todos los estados
- El `forall`: una restricción por robot y residuo
- Kairos: 16 acciones; deja los residuos en la base antes de subir a la UCI
- El dominio no ha cambiado ni un carácter
- S_0 cuenta: un `always` falso en él es irresoluble

El validador también comprueba las restricciones:

```
$ 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
```

sometime, at end, at-most-once

p03: la enfermera de la UCI quiere ver pasar al robot

```
(:goal (item-at waste1 dock))
(:constraints
  (sometime (robot-at r1 icu))))
```

- (sometime C): en al menos un estado
- 10 acciones en vez de 8: entra en la UCI y sale
- Un objetivo que no tiene que durar
- (at end C): en el último estado, como un objetivo (así un objetivo puede ser preferencia en la misma sección)

p04: la puerta de la base de carga es lenta y ruidosa

```
(:goal (and (item-at meds1 ward-a) (item-at waste1 dock)))
(:constraints
  (at-most-once (robot-at r1 dock))))
```

- (at-most-once C): como mucho un tramo de estados consecutivos, de cualquier longitud, o ninguno
- Plan corto: 11 acciones, dos visitas
- Con la regla: 12 acciones, una visita (llega, deja los residuos, recoge los medicamentos)

sometime-before y la prueba de la trampa

```
(: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)
```

- `(sometime-before A B)`: siempre que `A`, `B` en algún estado *anterior*
- Desinfecta a la *ida*; el `pick` ensucia: entra sucio al laboratorio
- Cumplida a la letra: la regla estaba mal escrita

Paso E: la prueba de la trampa para una regla

¿La forma más barata de cumplir las palabras sin hacer lo que se quería? «Limpio *cuando* entra» habla de un solo estado:

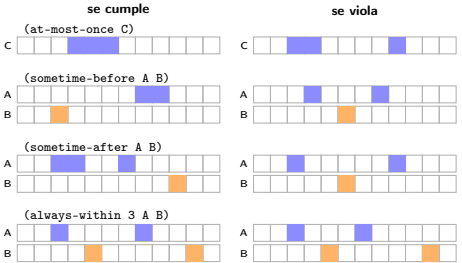
```
(always (imply (robot-at r1 lab) (sanitized r1)))
```

sometime-after, y cuatro operadores como patrones

p06: después de la UCI, desinfección en la base de carga

```
(:goal (item-at meds1 icu))
(:constraints
  (sometime-after (robot-at r1 icu) (robot-at r1 dock))))
```

- `(sometime-after A B)`: siempre que `A` se cumple, `B` se cumple entonces o *después*
- Objetivo alcanzado tras 7 acciones; el plan tiene 11
- Una obligación que saldar antes de que el plan pueda terminar



A la derecha: `C` vuelve; el primer `A` no tiene `B` antes; el último `A` no tiene `B` después; el segundo `A` espera 4 pasos

Reglas con reloj

- (`within t C`): `C` en algún estado de instante $\leq t$. Un plazo
- (`always-within t A B`): siempre que `A`, `B` entonces o como mucho t después. Un plazo que se reinicia
- (`hold-during t1 t2 C`): `C` en todo estado con $t_1 \leq t_i < t_2$. Una ventana
- (`hold-after t C`): `C` en todo estado con $t_i > t$. Una ventana que nunca se cierra

El instante de un estado

t_i: cuando empieza el estado S_i, $t_0 = 0$.

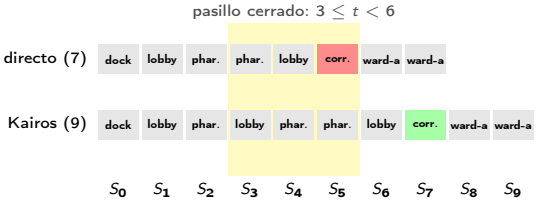
- Plan temporal: el instante del i-ésimo acontecimiento, en la unidad de las duraciones
- Plan secuencial: sin reloj; Kairos usa la posición, S_i en el instante i

Un plazo de 7 en un plan secuencial: «dentro de las 7 primeras acciones»

Contar pasos: una ventana en el pasillo

```
(:goal (item-at meds1 ward-a))
(:constraints
  (hold-during 3 6 (not (robot-at r1 corridor)))))
```

```
$ 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)
...
(drop r1 meds1 ward-a)
```



- Un plan secuencial no puede esperar: el planificador *inventa* una espera (ir al vestíbulo y volver)
- También: p07 (`within 7 ...`), p08 (`always-within 3 ...`), p10 (`hold-after 4 ...`)

Contar segundos

`courier-timed`: desplazamientos durativos (metros = segundos), pick y drop 2 s

```
(:goal (and (item-at meds1 ward-a) (item-at waste1 dock)))
(:constraints
  (within 45 (item-at meds1 ward-a)))
(:metric minimize (total-time)))
```

- Medicamentos en la sala A en 44,007: a tiempo
- Residuos primero: medicamentos en 54, rechazado por `kairos validate`
- p02, (`hold-during` 20 40 ...): un plan temporal espera

```
$ 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]
...
```

Los diez operadores, uno junto a otro

Restricción	Se cumple en la trayectoria $S_0, \ldots, S_n$ 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 $j \geq i$ 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 $j \geq i$ con $S_j \models B$
`(within t C)`	$S_i \models C$ para algún i con $t_i \leq t$
`(always-within t A B)`	para todo i con $S_i \models A$ existe $j \geq i$ con $S_j \models B$ y $t_j - t_i \leq t$
`(hold-during t1 t2 C)`	$S_i \models C$ para todo i con $t_1 \leq t_i < t_2$
`(hold-after t C)`	$S_i \models C$ para todo i con $t_i > t$

- `sometime-before`: $j < i$ estricto. `sometime-after`, `always-within`: $j \geq i$, basta el mismo estado
- «Si `A` se cumple...» lo satisface un plan en el que `A` nunca se cumple, incluido el plan vacío

Un problema por operador, dos planes cada uno

- De `courier-rules-p02` a `p10`: una restricción cada uno
- `-ok.plan` aceptado, `-bad.plan` rechazado; el plan malo es válido sin la restricción

```
$ 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: valida contra copias que conserven una restricción cada una

Lo que los operadores no pueden decir

Un vocabulario pequeño y fijo. En PDDL 3.0 no se pueden anidar (no hay `(always (sometime C))`); sus condiciones hablan de un solo estado. «*Cada vez* que toca residuos, se desinfecta antes de la *siguiente* entrada» necesita un hecho que recuerde, restringido con `always`. Los operadores son un fragmento de la **lógica temporal lineal** (Pnueli, 1977)

Reglas en el dominio

`courier-safe`: la regla de la UCI en el dominio (`icu` es una constante)

```
(:constraints
  (forall (?r - robot ?w - waste)
    (always (not (and (robot-at ?r icu) (carrying ?r ?w))))))
```

- Obliga en todos los problemas; un plan debe cumplir las dos secciones
- p01: las mismas 16 acciones
- p02: los residuos están *en* la UCI

```
$ kairos courier-safe-domain.pddl
    courier-safe-p02.pddl
...
kairos: unsolvable (exhausted)
```

- Leída al pie de la letra, prohíbe limpiar la UCI; el hospital quería decir «no meter residuos *en* la UCI»

Paso M: de quién es la regla

Alcance: una ley del mundo (precondiciones y efectos), una norma de la organización (restricción del dominio), una condición de hoy (restricción del problema). **Fuerza**: ¿debe cumplirse, o convendría?

Restricciones como preferencias

`courier-rules-p11.pddl`: la regla estricta de la UCI y dos deseos sobre el tiempo

```
(: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)))))
```

Un reglamento con excepciones

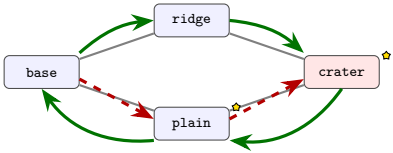
Las reglas que protegen a los pacientes siguen siendo estrictas; las de conveniencia reciben un precio

- **Restricción blanda**: una restricción dentro de una `preference`; 1 en la métrica si la trayectoria la incumple
- 8 pasos: medicamentos directos a la UCI; 6: residuos directos a la base. No pueden ir los dos primero
- Costes 5, 3, 2: el último plan entrega antes los medicamentos y paga 2
- La regla estricta se cumple en los tres

Dos mundos más: un róver y un cráter

```
(: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)
```



- Sin reglas (p01, en rojo): 4 acciones por la llanura
- En verde: por la cresta, una sola vez por el cráter, vuelta a casa aunque ningún objetivo lo pida
- Ningún hecho «el cráter ya se ha explorado»: la trayectoria guarda la historia

Dos mundos más: un taller con un horno

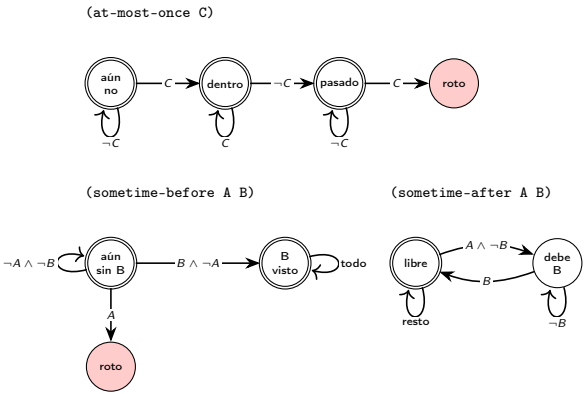
Dos reglas de seguridad del taller, así que restricciones del dominio:

```
(: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))))
```

- Acciones solo con la física: `heat` y `open-door` se ignoran entre sí
- `close-door` antes de `heat`, `cool` antes de abrir: regla 1. Una sola tanda: regla 2
- Un plan apresurado: `the plan violates the domain :constraints`

```
$ 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)
```

Qué hace el planificador con una restricción



Un plan solo puede terminar en un círculo doble; de un estado rojo no se sale nunca

- Un **monitor**: un pequeño autómata que recuerda justo lo necesario del pasado
- Kairos los compila: lo prohibido $\rightarrow$ precondición de todas las acciones; memoria $\rightarrow$ hecho nuevo con efectos condicionales; al final $\rightarrow$ objetivo
- Las reglas podan la búsqueda; el precio es el tamaño
- Solo Kairos y OPTIC devolvieron planes válidos

El método MODEL: reglas sobre el plan completo

Preguntas nuevas para los pasos M y D

Paso M. Enumera las reglas aparte de los objetivos. Para cada una:

- **¿Sobre una acción o sobre todo el plan?**
- **¿Un estado o historia?** Un estado cada vez: un `always`. «Antes», «después», «una vez», «como tarde»: memoria
- **¿De quién es la regla, y puede ceder?** Dominio o problema; estricta o preferencia

Paso D. Una precondición nueva: ¿lo que la acción *necesita* (dinámica) o lo que alguien *quiere* del plan (una regla, quizá mejor escrita una vez como restricción)?

- La ficha del modelo gana una tabla de reglas: frase, alcance, fuerza, operador, y la prueba de la trampa hecha a cada una

Alternativas comparadas: una regla sobre el plan completo

	Restricción	Precondiciones en todas las acciones	Autómata con predicados extra
Fidelidad	Una vez, sobre todos los estados; sin fugas. Puede ser *demasiado* fuerte	Exacta solo si no se olvida ninguna acción; solo reglas de estado	Exacto si el autómata es correcto; cualquier regla
Tamaño	Monitores en todas las acciones	Solo las acciones afectadas	Uno o dos hechos por regla y objeto
Guía	Buena en Kairos	Buena	Buena
Claridad	La mejor: una línea, puede hacerse blanda	Dispersa, mezclada con la física	La peor: oculta en la contabilidad
Portabilidad	Mala	Buena: ADL	Buena: ADL, efectos condicionales

- `courier-icu-pre` (regla en `move` y `pick`) coincide con la restricción; `courier-icu-leaky` (solo `move`) encuentra un plan que recoge los residuos *en* la UCI
- `courier-dock-auto` añade `(left-dock ?r)`: las mismas 12 acciones que `at-most-once`
- **Veredicto**: escribe primero la regla como restricción, aunque el modelo final no pueda usarla; es la especificación y, con `kairos validate`, la prueba

Errores comunes (1)

Olvidar que el estado inicial cuenta

La base de carga como mucho una vez, con el robot empezando *en* la base:

```
$ kairos courier-rules-domain.pddl
    courier-rules-once-start.pddl
...
kairos: unsolvable (exhausted)
```

La única visita se gasta en S_0. *Corrección*: decide qué significa la regla para el estado en que empieza el día

Un operador fuera de su sección

En un objetivo, `sometime` es un predicado que nadie declaró:

```
$ 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]
...
```

Después, `unsolvable`. *Corrección*: muévelo a `:constraints`; lee los avisos

Errores comunes (2)

Una ventana que el plan no alcanza

`(hold-after 10 (robot-at r1 dock))`: vale un plan de 7 acciones que acaba en la sala A. *Corrección*: si debe estar en la base, dilo también en el objetivo

Quieto durante toda una ventana

Kairos mira la ventana cuando *empiezan* los estados: un robot quieto en el pasillo de 14 a 45 pasa (`hold-during 20 40 ...`). *Corrección*: literales temporizados cambian `(closed corridor)` en 20 y 40, y un `always`

Pasos tomados por segundos

`(within 45 ...)` en un problema secuencial da 45 *acciones*. *Corrección*: un número va en la unidad del plan: posiciones o la de las duraciones

Una regla más fuerte que el mundo

`unsolvable` en un problema fácil. *Corrección*: quita restricciones una a una hasta que salga un plan; valídalo contra cada una sola

1 Objetivos blandos y preferencias

2 Restricciones sobre todo el plan

3 Fluentes de objeto

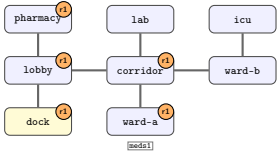
4 Resumen

Un robot en dos habitaciones

`courier-twin-domain.pddl`: `move` ha perdido una línea

```
(: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 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)
```



- 6 acciones en vez de 10; sale *desde el vestíbulo*
- El comprobador y el validador lo aceptan
- 2^8 situaciones, solo 8 tienen sentido

Una función cuyo valor es un objeto

Se declara en `:functions`, con el tipo del valor:

```
(:functions
  (location-of ?r - robot) - location    ; the robot's room
  (holder ?i - item) - place             ; a room, or a robot
```

- **Valor inicial**: `(= (location-of r1) dock)`
- **Comprobación**: `(= (location-of ?r) ?from)` (necesita `:equality`)
- **Cambio**: `(assign (location-of ?r) ?to)`; el valor antiguo desaparece, nada que borrar
- Requisito: `:object-fluents`

Predicado: ocho respuestas

lobby	corr.	phar.	lab	ward-a	ward-b	icu	dock
cierto	falso	cierto	falso	falso	falso	falso	cierto

`(robot-at r1 _)`

Fluente de objeto: un valor

`pharmacy` `(location-of r1)`

una casilla; asignar la sobrescribe

`(holder ?i)`: una habitación o un robot (supertipo `place`); sustituye a `item-at` y `carrying`

El mundo del mensajero con fluentes de objeto

```
(: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))))
```

```
  (= (location-of r1) dock) (free r1)
  (= (holder meds1) pharmacy))
(:goal (= (holder meds1) ward-a)))
```

- `move`: una asignación, ningún borrado que olvidar
- `pick`: el artículo cambia de poseedor, de la habitación al robot
- `(free ?r)` sigue siendo un predicado: cierto o falso, no un valor
- Las mismas 7 acciones que el dominio con tipos
- El error no se puede escribir: no hay segundo valor

Fluentes como términos

`courier-terms`: un fluente donde se espera un objeto

```
; 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))))
```

- Desaparecen los parámetros que solo llevaban un valor
- Acciones instanciadas: 30 (`courier-of`) frente a 23

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

- El precio: (`move r1 lobby`) ya no dice desde dónde

Sin valor: indefinido

`courier-cargo`: `(room-of ?i)` no tiene valor mientras se lleva

```
(: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)))
```

- Ausente de `:init`, o asignado `undefined`; no es igual a nada
- Mejor fluentes totales (`holder`) que parciales

undefined no se puede comparar

Nada de `(= (room-of ?i) undefined)`: `kairos check` lo rechaza. *Corrección*: conserva un hecho

Parar una acción antes:

```
$ 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
```

Dos mundos más: bloques con un fluente

```
(: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)))
```

- `(on-top-of ?b)`: un bloque o la mesa
- *Despejado* no se guarda: ningún bloque lo tiene como soporte
- Dos acciones, un efecto cada una; nada que se pueda desacompasar

```
$ kairos blocks-of-domain.pddl
    blocks-of-p01.pddl
; cost = 3
(unstack c)
(stack b c)
(stack a b)
```

Dos mundos más: colorear un mapa

```
(:types region color)

(:predicates
  (border ?a ?b - region))

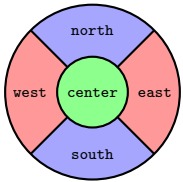
(:functions
  (color-of ?r - region) - color)   ; undefined until painted

(: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)))
```

Objetivo de `coloring-p01`: «toda región tiene *algún* color»

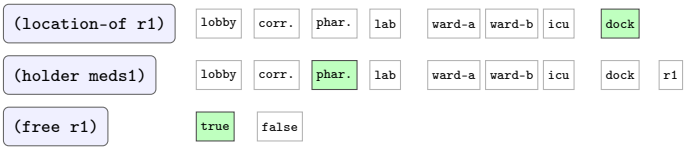
```
(:goal (forall (?r - region)
         (exists (?c - color) (= (color-of ?r) ?c)))))
```

- Un vecino sin color pasa: indefinido no es igual a ?c



- Kairos: 5 acciones `paint`, tres colores
- Dos colores (`p02`): 94 estados, `unsolvable`: una pequeña demostración
- Un CSP: una variable por región

Variables de estado: lo que ve el planificador



Un estado es un valor por variable: $8 \times 9 \times 2 = 144$ combinaciones, frente a 2^{18} conjuntos de hechos. Con predicados, el planificador debe *descubrir* las variables (**grupos mutex**):

```
$ kairos courier-domain.pddl courier-p01.pddl -v
kairos: 18 atoms, 30 actions (0 durative), 3 variables (2 mutex groups), ...
$ kairos courier-of-domain.pddl courier-of-p01.pddl -v
kairos: 1 atoms, 30 actions (0 durative), 3 variables (0 mutex groups), ...
```

- Los fluentes *eran* las variables. `courier-twin-p02`: 10 variables (1 mutex groups): un invariante que no se cumple

El método MODEL: un valor o muchos

Nuevas preguntas para el paso O (Objetos y estado)

Para cada propiedad dinámica, ¿cuántos valores tiene para un objeto?

- **¿Exactamente uno, siempre?** Habitación de un robot, poseedor de un artículo: un fluente de objeto, o un predicado con un invariante declarado y probado
- **¿Como mucho uno?** Color de una región: un fluente que puede estar indefinido; ¿quién necesita preguntar?
- **¿Cualquier número?** Habitaciones visitadas, artículos en un carro: un predicado
- **¿Un número?** Un fluente numérico

La tabla de estado gana una columna, *valores por objeto*. Cada «uno» escrito como predicado es un invariante, y el paso E le debe una prueba.

Alternativas comparadas: dónde están las cosas

	Predicados	Fluentes de objeto	Fluentes como términos
Fidelidad	Correcta solo mientras cada acción mantenga el invariante	El invariante no puede romperse	Igual
Tamaño	18 hechos, 30 acciones instanciadas	1 hecho, 2 fluentes, 30 acciones instanciadas	23 acciones instanciadas
Guía	3 variables, tras encontrar 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
Portabilidad	Todos los planificadores	`:object-fluents`: de los comparados, solo Kairos	Igual

Veredicto

Piensa siempre en fluentes: «un valor o muchos» pertenece al paso O sea cual sea la sintaxis. Escribe fluentes cuando el planificador admita PDDL 3.1 por completo; si no, predicados, con cada propiedad de un solo valor apuntada como invariante y probada (el recuento de variables y grupos mutex de `kairos -v` es una prueba barata)

Errores comunes

Las funciones no son predicados

`(location-of r1 dock)` en `:init`: un aviso, el hecho se ignora, `unsolvable`. *Corrección*: compara o asigna siempre

Dos valores iniciales, o ninguno

Dos: un aviso, se usa el último. Ninguno: `unsolvable` tras un estado. *Corrección*: lee los avisos; comprueba primero los valores iniciales

Un valor del tipo equivocado

```
$ 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]
```

Un fluente para una relación

El vestíbulo tiene tres vecinos: como fluente, se pierden soluciones

1 Objetivos blandos y preferencias

2 Restricciones sobre todo el plan

3 Fluentes de objeto

4 Resumen

PDDL 3 de un vistazo

Característica	Requisito	Versión	Dónde
Objetivos blandos, preferencias en precondiciones, `is-violated`	`:preferences`	3.0	Cap. 14
Restricciones de trayectoria, en el dominio y en el problema	`:constraints`	3.0	Cap. 15
Restricciones blandas (una preferencia sobre una restricción)	las dos anteriores	3.0	Cap. 15
Funciones con objetos como valores, `undefined`	`:object-fluents`	3.1	Cap. 16
Funciones con números como valores (el antiguo `:fluents`)	`:numeric-fluents`	3.1	Cap. 8
Fluentes numéricos y de objeto juntos	`:fluents`	3.1	
`total-cost` incrementado por las acciones	`:action-costs`	3.1	Cap. 9

- Ninguna clase nueva de acción ni de mundo: un sitio para decir directamente lo que quiere decir el modelador. *Esto se desea, esto es una regla, esto tiene un valor*
- El coste: menos planificadores lo leen. La portabilidad fue la última palabra de cada comparativa

Qué llevarse

- Una **preferencia** puede violarse a un precio fijado en la métrica: en un objetivo, sobre el último estado; en una precondición, sobre cada aplicación. Siempre `minimize`
- Pesos en la unidad del coste: deseos frente a esfuerzo (beneficio neto). Con objetivos blandos, deja correr la búsqueda *anytime*
- Una **restricción de trayectoria** juzga todos los estados, S_0 incluido; diez operadores, estrictos salvo en una preferencia; en dominio o problema
- El dominio dice lo que *se puede* hacer; las restricciones, lo que *está permitido* hacer. Haz la prueba de la trampa a cada regla
- Un **fluente de objeto** tiene un valor: ningún borrado que olvidar; `undefined` se puede asignar, nunca comprobar
- MODEL: el paso M separa lo obligatorio de lo deseado, y las reglas por alcance y fuerza; el paso O pregunta «un valor o muchos»

Ejercicios para practicar

- **14.3.2**: pesos para una prioridad estricta entre tres clases
- **14.4.1**: el peso exacto con el que se entrega la comida
- **14.6.3**: por qué un objetivo blando no expresa la regla del vestíbulo
- **14.9.2**: el paso M a partir de la tarde de un estudiante
- **15.1.1**: ¿objetivo, precondición o plan completo?
- **15.3.3**: la regla de desinfección escrita con `always`
- **15.6.2**: cambia un peso en `courier-rules-p11`
- **15.9.2**: el paso M sobre una historia de cocina
- **16.7.1**: ¿predicado, fluente de objeto o fluente numérico?
- **16.8.1**: qué característica de la Parte V para cada necesidad