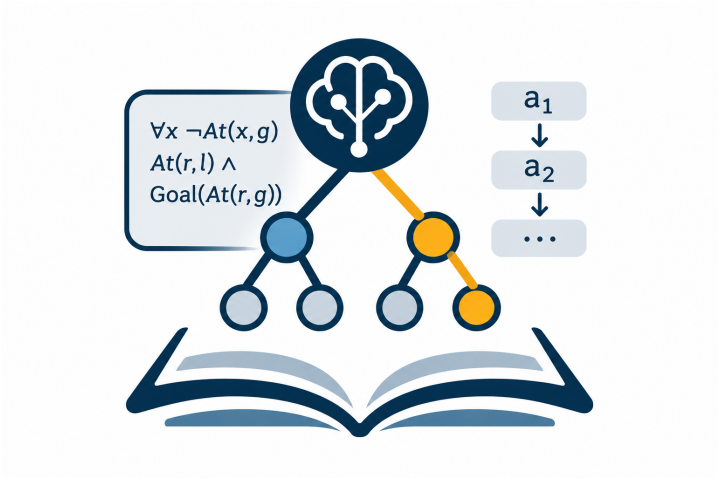
Automated Planning

Planning and PDDL

Modeling problems for automated planners, from PDDL 1.2 to PDDL+



Francisco Martín Rico

v0.1.0

November 21, 2026

Intelligent Robotics Lab, Universidad Rey Juan Carlos

Copyright
© 2026 Francisco Martín Rico.



License
This book is distributed under the *Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International* license (CC BY-NC-ND 4.0).

You may **share** (copy and redistribute) the material in any medium or format, under these terms: **attribution** (give credit to the author), **non-commercial** use and **no derivatives** (do not remix, transform or build upon the material).

Full text of the license:
https://creativecommons.org/licenses/by-nc-nd/4.0/

The examples
Every PDDL domain and problem printed in this book is a file in the `examples` folder that comes with it, and every plan shown was produced and validated by the Kairos planner (version 1.0.6). The command `make check-examples` runs them all again. The outputs were obtained with the options of that command, in particular `--threads 1`: with more threads, Kairos runs several searches at once and may print other plans, as valid as those shown. Long outputs are cut, and a line `...` marks each cut.

Colophon
This document was typeset with KOMA-Script and LaTeX, using the kaobook class.

"Plans are worthless, but planning is everything."

— Dwight D. Eisenhower

"All models are wrong, but some are useful."

— George E. P. Box

Preface

A planner is a program that receives a description of a situation, of what can be done in it and of what is wanted, and answers with a sequence of actions that leads from one to the other. The program is general: the same planner moves robots in a hospital, schedules the observations of a satellite or solves a puzzle. What changes from one use to the next is the description, and writing that description well is a skill of its own. This book teaches that skill.

The description is written in PDDL, the Planning Domain Definition Language. PDDL was born in 1998 as the common language of the first planning competition and has grown ever since: types and logical conditions first (PDDL 1.2), then numbers and time (PDDL 2.1), derived facts and events fixed on the calendar (PDDL 2.2), preferences and constraints over the whole plan (PDDL 3), state variables that hold objects (PDDL 3.1) and, in PDDL+, a world that changes on its own, continuously, whether the agent acts or not. The book walks the whole language in that order, from the simplest model that a planner can solve to the hybrid systems of the last chapters.

What makes this book different

Most introductions to PDDL explain the language: what each construct means and how to write it. This book explains the language too, but its subject is *modeling*: how to go from a situation described in words (a robot that must deliver medicines before the pharmacy closes, without running out of battery) to a PDDL model that a planner solves well, and how to choose among the several models that are always possible.

To that end the book builds a method, called MODEL after its five steps: **M**ap the situation, name the **O**bjects and their state, describe the **D**ynamics, **E**xercise the model with a planner, and **L**ook again at the alternatives. The method is introduced in the first chapter with the little PDDL that it needs, and every later chapter adds to it: each new feature of the language brings new questions to ask of a problem, new ways to encode it, and new ways to get it wrong. The last part of the book puts the complete method to work on a problem taken from robotics.

Three decisions follow from that purpose.

- **One world that grows.** A courier robot in a hospital floor appears in every chapter. It starts as a robot that picks and drops; it gains doors, carts, a battery, a clock, opening hours, preferences and, in the end, a battery that drains by itself. Seeing the same world modeled with more and more of the language shows what each feature is *for*. Around it, every chapter brings other small domains, so that no idea is tied to a single example.
- **Alternatives, compared.** Almost anything can be encoded in several ways. Each chapter sets two or three encodings of the same thing side by side, runs them, and says when each one is the better choice.
- **Everything runs.** Every domain and problem in the book is a file that comes with it, and every plan printed was produced by a planner and validated. A model is a program, and like a program it is learned by running it, breaking it and repairing it.

The planner

The examples use Kairos, an open-source planner that covers all the PDDL of this book with a single program: classical, numeric and temporal planning, preferences and constraints, and PDDL+. It also checks models without planning, with error messages that point at the line and suggest the fix, and it validates plans. Appendix A explains how to install and use it. Nothing in the models depends on Kairos: they are standard PDDL, and the book says so where other planners read a construct differently or do not support it.

Who this book is for

The book is written for a university course on AI Planning, and for anyone who has to write planning models: roboticists, engineers of autonomous systems, researchers who use a planner as a tool. It assumes that the reader can program and has met basic logic (and, or, not, for all, exists) and graph search. No previous contact with planning is needed.

A one-semester course can follow the book in order. Parts I and II are the core and should not be skipped; Part III (numbers) is needed for everything after it; Parts IV, V and VI can be taught in that order or cut short at the end of any of them. Every chapter ends with exercises for each of its sections, most of them to be done at the keyboard, and the solutions are in Appendix C.

Contents

Contents vi

Part I. Planning and models 1

1. Planning: deciding before acting 2
1.1. Deciding before acting . . . 2
The courier world . . . 3
1.2. A general solver plus a model . . . 3
The three ingredients: states, actions, goal . . . 4
1.3. The classical assumptions . . . 5
1.4. Planning and its alternatives . . . 6
1.5. Where planning is used . . . 7
1.6. Domain, problem and PDDL . . . 8
PDDL and its versions . . . 9
1.7. Meeting Kairos . . . 10
Same domain, another problem . . . 11
1.8. The MODEL method: five steps and five criteria . . . 12
The five criteria . . . 13
Alternatives compared . . . 14
1.9. Common mistakes . . . 15
1.10. Summary . . . 16
1.11. Further reading . . . 16
1.12. Exercises . . . 17

2. A first model: STRIPS 20
2.1. Lists inside lists: the shape of PDDL . . . 20
2.2. Predicates, objects and facts . . . 21
What can be said: predicates . . . 21
What is true: facts and states . . . 22
2.3. Actions: preconditions and effects . . . 22
Applying an action . . . 23
2.4. The problem: objects, initial state and goal . . . 24
2.5. Modeling the courier world . . . 25
Step M: map the situation . . . 25
Step O: objects and state . . . 26
Step D: dynamics . . . 26
A problem for the domain . . . 28
2.6. Exercising the model . . . 29
Check, solve, validate . . . 29
The test set . . . 30
2.7. Two classics: blocks world and gripper . . . 33
Blocks world . . . 33
Gripper . . . 35
2.8. The MODEL method: the first full pass . . . 37
What this chapter adds . . . 37
Alternatives compared . . . 38
2.9. Common mistakes . . . 40
2.10. Summary . . . 42
2.11. Further reading . . . 43

2.12. Exercises . . . 43

3. What a planner does with your model **47**
3.1. What a model means . . . 47
A problem small enough to draw . . . 47
States . . . 48
Applicable actions and successors . . . 48
Plans . . . 49
The state space . . . 50
3.2. Grounding: from variables to objects . . . 50
Counting by hand . . . 51
What the planner says . . . 51
The whole floor, the puzzle and the towers . . . 52
3.3. Why the space explodes . . . 54
Counting states . . . 54
3.4. Forward search . . . 55
The loop . . . 55
Searching blind . . . 55
3.5. Heuristics: asking a simpler problem . . . 56
A number for each state . . . 56
Ignoring the deletes . . . 56
The heuristic over the whole space . . . 57
When the relaxation sees little . . . 58
3.6. A plan, a better plan, the best plan . . . 59
Anytime: the first plan, then better ones . . . 59
Choosing how much to ask . . . 60
3.7. No plan, or no plan found . . . 60
Proved by grounding . . . 61
Proved by search . . . 61
Not found . . . 62
3.8. What this means for the modeler . . . 62
Parameters multiply . . . 62
Static preconditions prune . . . 63
Dead ends . . . 63
Symmetry . . . 64
3.9. The MODEL method: size and guidance in numbers . . . 65
Alternatives compared . . . 66
3.10. Common mistakes . . . 67
3.11. Summary . . . 68
3.12. Further reading . . . 69
3.13. Exercises . . . 69

Part II. Classical planning: PDDL 1.2 **73**

4. Types, constants and equality **74**
4.1. What type predicates cost . . . 74
4.2. Types . . . 75
What the checker now catches . . . 77
What the planner gains . . . 77
4.3. Type hierarchies . . . 78
A second domain: logistics . . . 81
4.4. A parameter of several types: `either` . . . 83
The alternative: name the union . . . 84
4.5. Constants . . . 84

4.6. Equality . . . 86
When the inequality is needed . . . 88
4.7. The MODEL method: kinds of things . . . 88
Alternatives compared . . . 89
4.8. Common mistakes . . . 91
4.9. Summary . . . 92
4.10. Further reading . . . 93
4.11. Exercises . . . 93

5. Richer conditions **96**
5.1. Conditions about absence: negative preconditions . . . 96
Three states with two predicates . . . 97
Exercising the doors . . . 98
5.2. Negative goals . . . 99
5.3. Alternatives in a condition: `or` and `imply` . . . 100
5.4. Some object: `exists` . . . 101
A puzzle of keys and chests . . . 102
5.5. Every object: `forall` . . . 103
Blocks world: `clear` is the absence of `on` . . . 103
One `move` for all passages . . . 104
Quantified goals . . . 105
5.6. What a planner does with rich conditions . . . 106
5.7. The MODEL method: absence, any, every . . . 106
Alternatives compared . . . 107
5.8. Common mistakes . . . 109
5.9. Summary . . . 111
5.10. Further reading . . . 111
5.11. Exercises . . . 112

6. Richer effects **114**
6.1. Effects that depend on the situation . . . 114
6.2. Conditional effects: `when` . . . 115
What a conditional effect means . . . 116
Running it . . . 117
A condition in the effect or in the precondition . . . 117
6.3. Universal effects: `forall` . . . 118
6.4. Both together: for every object such that . . . 119
Lights Out . . . 119
The briefcase . . . 120
The cart . . . 121
6.5. Effects to avoid: the sterile laboratory . . . 124
Exercising the model . . . 125
One action per case . . . 126
6.6. The requirement `:adl` . . . 127
6.7. What planners do with conditional effects . . . 127
6.8. The MODEL method: effects that depend on the situation . . . 129
Alternatives compared . . . 129
A fourth way: do not store what can be deduced . . . 131
6.9. Common mistakes . . . 132
6.10. Summary . . . 134
6.11. Further reading . . . 134
6.12. Exercises . . . 135

7. Modeling classical problems: patterns and choices **138**
7.1. A problem from a story: the waiter . . . 138
Step M: map the situation . . . 139
Step O: objects and state . . . 139
Step D: dynamics . . . 140
Step E: exercise the model . . . 140
7.2. Patterns for the state . . . 141
Static table . . . 142
Mirrored facts . . . 143
At-most-one invariant . . . 143
Successor chain . . . 144
Slots . . . 145
7.3. Patterns for control . . . 145
Phase token . . . 145
Turn token . . . 146
Resource lock . . . 147
7.4. Patterns for the actions . . . 148
Inverse pair and one-way action . . . 148
Merged action . . . 150
7.5. Anti-patterns . . . 150
Forgotten delete . . . 150
Goal in the initial state . . . 151
Out-of-step twins . . . 151
Idle parameter . . . 152
7.6. Debugging a model . . . 152
No plan . . . 152
A wrong plan . . . 154
7.7. The MODEL method: the classical method, consolidated . . . 154
Alternatives compared . . . 155
7.8. Common mistakes . . . 156
7.9. Summary . . . 157
7.10. Further reading . . . 157
7.11. Exercises . . . 158

PART III. NUMBERS: PDDL 2.1 **161**

8. Numeric fluents **162**
8.1. What a fact cannot say . . . 162
8.2. Functions: numbers in the state . . . 163
8.3. Comparing and changing numbers: the battery . . . 164
Numeric conditions . . . 164
Numeric effects . . . 165
Running it . . . 166
Exercising the model . . . 169
8.4. Numbers that never change: distances, weights and capacity . . . 170
A load capacity . . . 170
Another domain: a truck and its fuel . . . 172
8.5. Setting, scaling and asking: jugs, dilutions and stock . . . 173
The water jugs . . . 173
Scaling: a dilution . . . 175
Counting: stock in a warehouse . . . 176
8.6. Values that were never given . . . 177
8.7. Why numbers make planning harder . . . 178

8.8. The MODEL method: quantities . . . 180
Alternatives compared . . . 181
8.9. Common mistakes . . . 183
8.10. Summary . . . 184
8.11. Further reading . . . 185
8.12. Exercises . . . 185

9. Plan quality: metrics and action costs **188**
9.1. When any plan is not enough . . . 188
9.2. Action costs . . . 189
The requirement and its rules . . . 191
9.3. Costs from data: a kitchen with prices . . . 191
Actions that cost nothing . . . 193
9.4. The planner at work: anytime search . . . 193
What `proven optimal` means . . . 195
Good enough . . . 195
9.5. Metrics over any fluent . . . 196
Maximizing . . . 198
9.6. Several wishes in one number . . . 199
A road trip: tolls and fuel . . . 200
9.7. A limit or a metric . . . 202
9.8. The MODEL method: what makes a plan better . . . 203
Alternatives compared . . . 203
9.9. Common mistakes . . . 205
9.10. Summary . . . 206
9.11. Further reading . . . 207
9.12. Exercises . . . 207

Part IV. Time: PDDL 2.1 and 2.2 **210**

10. Durative actions **211**
10.1. What a sequence cannot say . . . 211
10.2. A first durative action . . . 212
The courier with fixed durations . . . 213
Reading a temporal plan . . . 215
10.3. What happens at each instant . . . 215
Why 10.001 and not 10 . . . 216
The validator agrees . . . 216
10.4. Conditions in time . . . 217
The interval is open . . . 217
10.5. Durations that depend on the action . . . 218
10.6. Working in parallel . . . 220
10.7. One thing at a time . . . 222
10.8. Other domains: a kitchen and an elevator . . . 225
A kitchen: things that work alone . . . 225
An elevator: waiting for people . . . 226
10.9. The MODEL method: time . . . 228
Alternatives compared . . . 228
10.10. Common mistakes . . . 230
10.11. Summary . . . 232
10.12. Further reading . . . 233
10.13. Exercises . . . 233

11. Coordination in time **236**
11.1. Plans that exist only if actions overlap . . . 236
11.2. The envelope: a door held open . . . 238
11.3. Temporally simple, temporally expressive . . . 240
What Kairos does . . . 241
11.4. Working together: two robots and a crate . . . 242
A first model, and why it is brittle . . . 242
A model with slack . . . 243
11.5. Durations that the planner chooses . . . 244
`?duration` in effects . . . 245
How a planner chooses a duration . . . 246
A coffee machine . . . 247
11.6. How planners order time . . . 248
11.7. The MODEL method: coordination . . . 249
Alternatives compared . . . 249
11.8. Common mistakes . . . 251
11.9. Summary . . . 253
11.10. Further reading . . . 253
11.11. Exercises . . . 254

12. Continuous change **256**
12.1. A staircase where the world has a ramp . . . 256
12.2. Continuous effects . . . 258
12.3. Reading a number while it changes . . . 260
Water that comes and goes: the greenhouse . . . 262
12.4. The generator: when to start . . . 265
12.5. The oven: when to stop . . . 267
12.6. What the planner must find . . . 270
12.7. The courier world: a battery that drains and charges . . . 271
Exercising the model . . . 272
Where to charge, and for how long . . . 274
12.8. The MODEL method: continuous change . . . 275
Alternatives compared . . . 276
12.9. Common mistakes . . . 277
12.10. Summary . . . 278
12.11. Further reading . . . 279
12.12. Exercises . . . 279

13. Derived predicates and a world with a calendar **282**
13.1. Facts that follow from other facts . . . 282
13.2. Derived predicates . . . 283
Where a derived predicate may appear . . . 284
13.3. Rules that call themselves . . . 285
Two more recursions: towers and circuits . . . 287
Rules and negation . . . 290
13.4. What a planner does with rules . . . 290
13.5. A world with a calendar: timed initial literals . . . 291
13.6. Windows and deadlines . . . 293
Deadlines . . . 293
A satellite and its windows . . . 295
13.7. Numbers on the calendar . . . 296
13.8. The MODEL method: consequences and the calendar . . . 297
Alternatives compared . . . 298
A deadline: timed literal or constraint . . . 299
13.9. Common mistakes . . . 299

13.10. Summary . . . 301
13.11. Further reading . . . 301
13.12. Exercises . . . 302

Part V. Preferences and constraints: PDDL 3 **305**

14. Soft goals and preferences **306**
14.1. More wishes than battery . . . 306
14.2. A preference in the goal . . . 307
14.3. What each wish is worth . . . 309
A wish about the robot itself . . . 310
Hard and soft goals together . . . 311
14.4. Wishes against effort . . . 312
14.5. One preference, many instances . . . 313
14.6. Wishes about the way: preferences in preconditions . . . 314
14.7. Two more worlds . . . 316
A tourist's day . . . 316
A satellite with more requests than time . . . 317
14.8. What the planner does with a preference . . . 319
14.9. The MODEL method: wishes and their worth . . . 320
Alternatives compared . . . 321
14.10. Common mistakes . . . 322
14.11. Summary . . . 324
14.12. Further reading . . . 324
14.13. Exercises . . . 325

15. Constraints on the whole plan **327**
15.1. A rule that the plan breaks . . . 327
15.2. The states of a plan: always . . . 329
15.3. What must happen, and in which order . . . 330
sometime and at end . . . 330
at-most-once . . . 330
sometime-before and sometime-after . . . 331
15.4. Rules with a clock . . . 332
Counting steps . . . 332
Counting seconds . . . 333
15.5. The ten operators, side by side . . . 334
15.6. Where rules live, and rules that bend . . . 336
In the domain or in the problem . . . 336
Constraints as preferences . . . 337
15.7. Two more worlds . . . 337
A rover and a crater . . . 337
A workshop with an oven . . . 338
15.8. What the planner does with a constraint . . . 339
15.9. The MODEL method: rules about the whole plan . . . 340
Alternatives compared . . . 341
15.10. Common mistakes . . . 342
15.11. Summary . . . 344
15.12. Further reading . . . 344
15.13. Exercises . . . 345

16. Object fluents **347**
16.1. One robot in two rooms . . . 347
16.2. A function whose value is an object . . . 348

16.3. Fluents as terms . . . 350
16.4. No value: undefined . . . 351
16.5. Two more worlds . . . 352
Blocks: one fluent instead of four predicates . . . 352
Coloring a map . . . 353
16.6. State variables: what the planner sees . . . 354
16.7. The MODEL method: one value or many . . . 355
Alternatives compared . . . 356
16.8. PDDL 3 at a glance . . . 356
16.9. Common mistakes . . . 357
16.10. Summary . . . 358
16.11. Further reading . . . 358
16.12. Exercises . . . 359

Part VI. Hybrid systems: PDDL+ **361**

17. Processes and events **362**
17.1. A world that does not wait . . . 362
17.2. Processes: change while a condition holds . . . 363
The first plan . . . 364
The plan is not unique, and neither is the time . . . 365
17.3. Events: change at an instant . . . 366
An event must switch itself off . . . 366
A problem without a plan . . . 367
17.4. Three kinds of change . . . 367
The bouncing ball . . . 368
The car . . . 370
17.5. The courier world in PDDL+ . . . 372
A move is an action, a process and an event . . . 373
The battery: a process and an event . . . 373
Exercising the model . . . 374
The cheat test: a robot that coasts . . . 375
A sample that warms up . . . 376
17.6. How Kairos plans PDDL+ . . . 378
Search in steps of time . . . 378
Validation in continuous time . . . 378
What it can miss . . . 379
17.7. The MODEL method: who causes each change . . . 380
Alternatives compared . . . 380
17.8. Common mistakes . . . 381
17.9. Summary . . . 382
17.10. Further reading . . . 383
17.11. Exercises . . . 383

18. Modeling hybrid systems **386**
18.1. Rules that hold at all times: global constraints . . . 386
18.2. Change that is not linear . . . 388
A car with air drag . . . 388
18.3. Processes that interact . . . 390
A drone that lands . . . 392
18.4. Events that cascade, and events that never stop . . . 394
Events as devices: the thermostat . . . 395
Events that never stop . . . 396

18.5. Processes with durative actions and timed literals . . . 398
Waiting is not free . . . 399
18.6. One system, three models . . . 401
What each level cannot say . . . 402
18.7. Limits: steps and tolerances . . . 403
An event does not rescue the search . . . 404
Exact values . . . 405
18.8. The MODEL method: testing and choosing hybrid models . . . 406
Alternatives compared . . . 407
18.9. Common mistakes . . . 407
18.10. Summary . . . 408
18.11. Further reading . . . 409
18.12. Exercises . . . 409

Part VII. The method at work **412**

19. The MODEL method, complete **413**
19.1. The method on one page . . . 413
19.2. The five checklists . . . 415
19.3. Which feature for which need . . . 417
What each feature costs . . . 419
19.4. The catalog of patterns . . . 420
19.5. The catalog of anti-patterns . . . 423
19.6. Debugging, beyond classical . . . 425
Read the verdict first . . . 425
No plan, with a proof . . . 426
No plan, without a proof . . . 426
A wrong plan, and a poor one . . . 427
19.7. Growing a model in versions . . . 428
Version 1: classical . . . 429
Version 2: numbers . . . 431
Version 3: time . . . 432
Version 4: wishes . . . 434
Version 5: processes . . . 436
Five versions, side by side . . . 438
19.8. Documenting a model . . . 438
19.9. The MODEL method: the whole method . . . 441
Alternatives compared . . . 441
19.10. Common mistakes . . . 442
19.11. Summary . . . 444
19.12. Further reading . . . 444
19.13. Exercises . . . 445

20. Case study: a robot mission from story to execution **448**
20.1. The story . . . 448
20.2. Step M: map the situation . . . 449
A plan for the modeling . . . 450
20.3. Version 1: who carries what . . . 451
Steps O and D . . . 451
Step E: the test set . . . 453
Step L: three ways to write the rule of the lab . . . 454
What version 1 cannot say . . . 455
20.4. Version 2: time and battery . . . 456
Steps O and D: what changes . . . 456

Step E: the round in time . . . 457
Step L: how long is a charge . . . 459
What version 2 cannot say . . . 460
20.5. Version 3: a calendar and wishes . . . 460
Steps O and D: what changes . . . 461
Step E, first part: the tests that pass . . . 462
Step E, second part: the round, and a limit . . . 463
Step L: three ways to plan the wishes . . . 464
What version 3 cannot say . . . 466
20.6. Version 4: the battery belongs to the world . . . 467
Steps O and D: what changes . . . 467
Step E: one robot and a shutter . . . 468
Step L: does the round need version 4? . . . 470
20.7. From plan to execution . . . 471
What a plan is for a robot . . . 471
What can go wrong . . . 471
Planning again . . . 472
20.8. PlanSys2 . . . 473
Four nodes . . . 473
A plan becomes a behavior tree . . . 474
Action performers . . . 474
The application . . . 475
What this asks of the model . . . 476
20.9. What PDDL leaves out . . . 476
20.10. The MODEL method: one story, four models . . . 477
Alternatives compared . . . 478
20.11. Common mistakes . . . 479
20.12. Summary . . . 480
20.13. Further reading . . . 481
20.14. Exercises . . . 481

Appendices **484**

A. Kairos **485**
A.1. Installing . . . 485
With pip . . . 485
From source . . . 485
In a colcon workspace . . . 485
A.2. The commands . . . 486
A.3. Planning . . . 486
Options . . . 487
Engines and searches . . . 487
A.4. Plan formats . . . 488
A.5. Status and exit codes . . . 488
A.6. Checking a model . . . 489
A.7. Validating a plan . . . 490
A.8. Looking inside . . . 491
A.9. From Python . . . 491

B. PDDL quick reference **493**
B.1. Requirements . . . 493
B.2. Syntax templates . . . 494
Classical: PDDL 1.2 . . . 494
Numeric: PDDL 2.1, level 2 . . . 496

Temporal: PDDL 2.1, levels 3 and 4, and PDDL 2.2 497
Preferences, constraints and object fluents: PDDL 3 499
Hybrid: PDDL+ 500
B.3. Which planners support what 502

C. Solutions to the exercises **504**
C.1. Solutions for Chapter 1 504
C.2. Solutions for Chapter 2 508
C.3. Solutions for Chapter 3 515
C.4. Solutions for Chapter 4 521
C.5. Solutions for Chapter 5 527
C.6. Solutions for Chapter 6 532
C.7. Solutions for Chapter 7 537
C.8. Solutions for Chapter 8 542
C.9. Solutions for Chapter 9 546
C.10. Solutions for Chapter 10 549
C.11. Solutions for Chapter 11 554
C.12. Solutions for Chapter 12 560
C.13. Solutions for Chapter 13 565
C.14. Solutions for Chapter 14 570
C.15. Solutions for Chapter 15 573
C.16. Solutions for Chapter 16 576
C.17. Solutions for Chapter 17 579
C.18. Solutions for Chapter 18 585
C.19. Solutions for Chapter 19 591
C.20. Solutions for Chapter 20 597

Bibliography **605**

Alphabetical index **607**

Boxes used in this book

Four kinds of boxes interrupt the main text. The first one carries the thread of the book; the other three can be skipped on a first reading.

Method box

A green box marks a step of the MODEL method, the modeling method this book builds chapter by chapter: a step applied to the example at hand, or a new question that the chapter adds to the method.

Concept box

A gray box holds a self-contained digression: a clarification of terms, a look at how planners treat a feature, or a comparison that widens the topic without breaking its line.

History box

A blue box tells a real episode of the history of automated planning: who did what, when, and what came of it.

Pitfall box

A red box shows a mistake that modelers make often: what it looks like in the PDDL, the symptom it produces in the planner, and how to fix it.

Part I. Planning and models

1. Planning: deciding before acting

A courier robot waits at its charging dock on a hospital floor. A nurse in ward A asks for a box of medicines that sits on a shelf of the pharmacy. Nobody tells the robot which doors to cross or in which order to do things: it is told only what is wanted. Before its wheels turn, the robot has to work out that it must go to the pharmacy first, take the box, and only then head for the ward, through the lobby and the corridor because no other way exists. Working that out is *planning*, and a program that does it for any floor, any robot and any request is a *planner*.

This chapter explains what planning is and what a planner needs from us in order to work: a description of the world, of what can be done in it and of what is wanted. That description is called a model, and writing good models is the subject of the whole book. The chapter also presents the three companions of the rest of the text: PDDL, the language in which models are written; Kairos, the planner that runs them; and the MODEL method, five steps to go from a situation told in words to a model that a planner solves well.

We start with the idea of deciding before acting and with the courier world, the example that grows through every chapter. Then we look at how a general solver and a model share the work, at the assumptions that the simplest kind of planning makes and at what planning offers compared with programming the behavior by hand, searching a hand-coded state space, or learning. After a short tour of where planners are used, we meet PDDL and its versions, run Kairos for the first time on a model that we treat as a black box, and close with the MODEL method. No PDDL syntax is taught yet: that starts in Chapter 2.

1.1 Deciding before acting . . 2
The courier world 3
1.2 A general solver plus a model 3
The three ingredients: states, actions, goal 4
1.3 The classical assumptions 5
1.4 Planning and its alternatives 6
1.5 Where planning is used . 7
1.6 Domain, problem and PDDL 8
PDDL and its versions . . . 9
1.7 Meeting Kairos 10
Same domain, another problem 11
1.8 The MODEL method: five steps and five criteria . . . 12
The five criteria 13
Alternatives compared . . 14
1.9 Common mistakes 15
1.10 Summary 16
1.11 Further reading 16
1.12 Exercises 17

1.1. Deciding before acting

Some decisions need no thought about the future. A robot that sees an obstacle half a meter ahead stops: the right reaction depends only on what it perceives now. Other decisions are different, because what is right now depends on what will come later. The courier robot at the dock gains nothing by moving toward ward A, although ward A is where the medicines must end: it has to move *away* from the ward first, to the pharmacy. The first step makes sense only as part of a whole sequence.

Planning is the reasoning that produces such a sequence: given a situation, the things that can be done and a goal, find which actions to carry out, and in which order, so that the goal is reached. The result is a **plan**. For the delivery of the medicines a plan is: go to the lobby, go to the pharmacy, pick the box, go back to the lobby, go to the corridor, go to ward A, drop the box. Planning happens *before* acting, on a description of the world and not on the world itself. The robot imagines the consequences of its actions, and it can discard a thousand bad sequences without bumping into a single wall.

Planning is finding, before acting, a sequence of actions that leads from the current situation to a goal. It works on a description of the world, not on the world.

Three features tell a planning problem from other computing problems. First, the answer is a course of action, not a number or a classification. Second, the actions interact: one action makes another possible (the robot can pick the box only once it is in the pharmacy) or impossible (once it

holds a box it cannot pick a second one). Third, there are far too many sequences to try them all blindly, so a planner needs to search with some intelligence. How it does so is the subject of Chapter 3; in this chapter the planner is a black box.

The courier world

One example runs through the whole book, and it is time to present it. The **courier world** is a hospital floor with eight places: a lobby, a corridor, a pharmacy, a laboratory, two wards (A and B), an intensive care unit (ICU) and a charging dock. The places are joined as Figure 1.1 shows: the lobby leads to the pharmacy, to the dock and to the corridor; the corridor leads to the lab and to both wards; and the ICU is reached only through ward B. Every connection can be crossed in both directions.

One or two courier robots, `r1` and `r2`, live on this floor. Their job is to carry items: boxes of medicines (`meds1`, `meds2`), a blood sample (`sample1`), a meal tray (`meal1`). In its first version a robot can do three things: move from a place to a neighboring one, pick an item that lies where the robot is, and drop the item that it carries. It carries one item at a time.

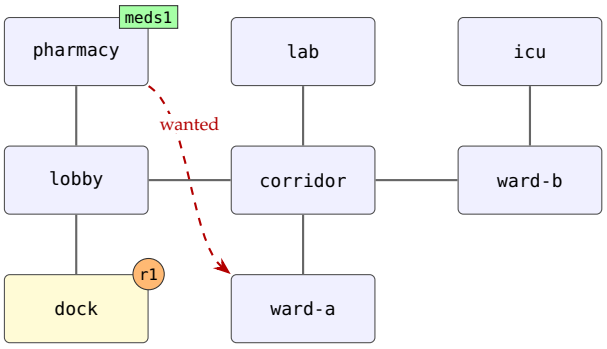


Figure 1.1.: The courier world: eight places and the connections between them. The robot `r1` rests at the dock, the medicines `meds1` are in the pharmacy, and somebody wants them in ward A (dashed arrow). The arrow is the wish, not the way: the way must go along the gray connections.

The world is small on purpose. It is easy to hold in the head, so that we can always check by hand what the planner answers. And it has room to grow: later chapters give it doors and keycards, a cart, a battery that runs down, distances and durations, opening hours, deliveries that are wished but not required, and in the end a battery that drains on its own whether the robot acts or not. Each addition will need a new piece of the language, and that is how the book walks through PDDL.

1.2. A general solver plus a model

How could we make the courier robot deliver the medicines? One way is to write a program that contains the answer: a function `deliver_medicines_to_ward_a()` with the seven steps inside. It works until something changes: the medicines are in the lab today, the corridor is closed, there are two requests. Each change needs a programmer.

Planning takes another way, called **model-based autonomy**. The work is split in two parts (Figure 1.2):

- A **model**: a description of the world in a formal language. It says which things exist, what can be true of them, what the agent can

do and what is wanted. It contains no instructions on *how* to reach the goal.

- A **planner**: a general program that reads any model written in that language and searches for a plan. It knows nothing about hospitals, robots or medicines. It knows how to search.

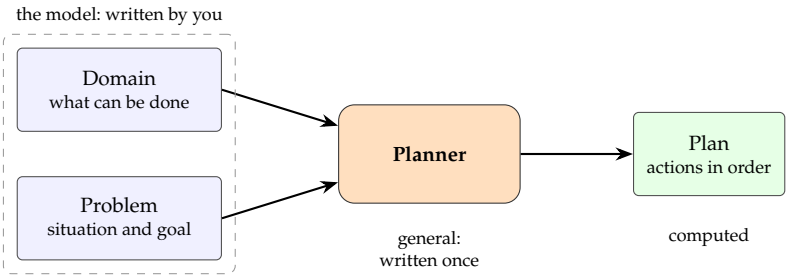


Figure 1.2.: Model-based autonomy. The model, in two files, is the only part that speaks about the hospital. The planner is the same program for every model. Change the model and the same planner gives the plan for the new situation.

The split pays off as soon as the situation changes. If the medicines are in the lab, one line of the model changes and the planner finds the new plan. If the hospital opens a new wing, the model gains some places and connections. If a new kind of robot arrives, the description of what it can do goes into the model. In none of these cases does anybody write, by hand, a new sequence of steps.

The planner is general and knows only how to search. Everything about the application is in the model. To change the behavior, change the model.

The split also moves the difficulty. The planner is written once, by specialists, and we take it as given. The model is written for every application, by us. A planner can only be as good as the model it receives: if the model says that the robot may carry ten boxes at a time, the plan will pile ten boxes on a robot that has one tray, and the planner will have done its job perfectly. This book is about that half of the work.

The three ingredients: states, actions, goal

Whatever the language, a planning model has three ingredients.

A **state** is a description of the world at one moment, with everything that matters for deciding and nothing else. In the courier world a state says where each robot is, where each item is and whether each robot carries something. It does not say what color the walls are, how charged the battery is (not yet) or what the weather is like. The state in which everything starts is the **initial state**.

An **action** is something the agent can do that changes the state. An action has two parts: the condition under which it can be done (its **precondition**) and the change that it makes (its **effects**). The robot can move from the lobby to the corridor if it is in the lobby and the two places are connected; after the move it is in the corridor and no longer in the lobby.

A **goal** is a condition on states: what must be true at the end. "The medicines are in ward A" is a goal. It does not describe one state but many: it says nothing about where the robot ends, and any state in which the medicines are in ward A will do.

With these three ingredients the task of the planner has an exact statement. A plan is a sequence of actions such that the first one can be done in the initial state, each of the following can be done in the state that the previous one leaves, and the state left by the last one satisfies the goal. Figure 1.3 draws that chain for the delivery of the medicines.

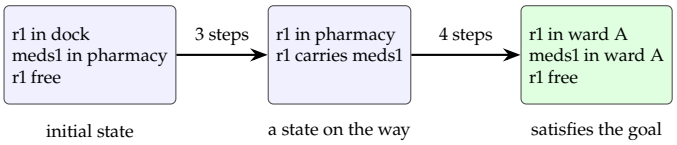


Figure 1.3.: A plan is a chain of states joined by actions. Each box lists what a state of the courier world says. The last state satisfies the goal "`meds1` in ward A"; that the robot is there too is a consequence, not something the goal asked for.

Readers who know graph search will recognize the picture: states are nodes, actions are edges, and a plan is a path from the initial state to any state that satisfies the goal. That is right, with one difference that makes planning a field of its own: nobody draws the graph. The model describes it in a compact way (a handful of action descriptions that apply to any robot, item and place) and the planner builds, as it goes, the small part that it needs. A graph with more nodes than atoms in the universe can be described in one page of model.

1.3. The classical assumptions

The simplest kind of planning, and the one where we begin, is **classical planning**. It simplifies the world with five assumptions. Each one is false in a real hospital, and each one makes planning much easier. A large part of the story of planning languages, and of this book, consists of lifting them one at a time.

Finite. There is a finite number of things, and a finite number of facts that can be true or false about them. A robot is in the lobby or it is not; it is never "2.37 meters into the lobby". The number of states is therefore finite, although huge.

Deterministic. An action applied in a state has exactly one result. When the robot picks the box, it holds the box. It never slips.

Fully known. The planner knows the initial state completely. There are no boxes whose place is unknown, and nothing needs to be sensed while the plan runs.

Static. The world changes only when the agent acts. Nobody else moves the medicines, doors do not close on a timetable, batteries do not drain by themselves.

Instantaneous actions. An action has no duration. It happens in a step, one action after another, and it makes no sense to ask how long the plan takes or whether two actions overlap.

Classical planning assumes a finite, deterministic, fully known and static world with instantaneous actions. The parts of the book lift these assumptions one by one.

Under these assumptions a plan is a plain sequence of actions and it is guaranteed to work: if the model is right, carrying out the plan in the world leads to the goal, with no need to look at the world on the way.

Table 1.1 shows which part of the book lifts which assumption. Two of them, determinism and full knowledge, stay with us to the end: PDDL, in all the versions that this book covers, describes a world whose behavior is known and predictable. That is less limiting than it sounds. A robot whose plan fails (the box slips, a door is locked) can observe the new situation and plan again from there, and Chapter 20 shows how plans are executed and repaired that way.

Assumption	What lifts it	Where
Finite	Numbers: battery levels, loads, distances	Part III
Instantaneous actions	Actions with duration, which can overlap	Part IV
Static	Facts that change at known times; then processes and events that the world runs by itself	Part IV (Chapter 13), Part VI
Deterministic	Not lifted in PDDL: execute, observe and plan again	Chapter 20
Fully known	Not lifted in PDDL	Chapter 20

Table 1.1.: The five classical assumptions and the part of the book that lifts each one. Parts I and II stay inside all five.

A sixth assumption hides in the word "goal". In classical planning a goal is a condition on the last state, and every plan that reaches it is as good as any other. Real requests are richer: the shortest route, the least energy, "if possible, bring the meal too", "never carry waste through the ICU". Part III adds a measure of quality to plans, and Part V adds goals that are wishes and conditions on the whole plan and not only on its end.

A wrong model that is useful

The classical assumptions do not claim that hospitals are finite, predictable and frozen. They are a decision about what to leave out. A model of the courier world in which rooms are points and moving takes no time is wrong, and it still tells the robot in which order to visit the rooms, which is what we asked. The question to ask of a model is never "is it true?" but "is it good enough for the decisions that we need from it?". Step M of the MODEL method, at the end of this chapter, asks that question explicitly.

1.4. Planning and its alternatives

Planning is one of several ways to make a machine choose its actions. Seeing the others side by side shows when a planner is the right tool.

Programming the behavior by hand. The engineer decides what to do in each situation and writes it down: a script, a state machine, a behavior tree. For a small, stable set of situations this is the best choice: it is fast, predictable and easy to certify. Its cost appears with variety. A courier with one robot and one kind of request needs a short script. With two robots, four items, closed doors and low batteries, the situations multiply, and the hand-written program must foresee every combination. The ones that nobody foresaw are the ones that fail at night.

Search on a hand-coded state space. The engineer writes a program in a general language: a data structure for the state, a function that generates successors, a goal test, perhaps a heuristic, and runs a search algorithm such as A*. This already separates the problem from the search and solves any instance. But everything is written for one problem: the successor function for the courier is of no use for a warehouse, the heuristic has to be invented each time, and a change in the rules means changing code. A planner is this same idea taken one step further: the state space is not coded but *declared* in a modeling language, so the search algorithm and its heuristics, which the planner derives from the model by itself, are reused for every problem.

Learning. The behavior is not written at all: it is learned from examples or from trial and error. Learning shines where no model can be written: recognizing a box in an image, grasping a soft object. It needs data or many trials, what it learns is tied to the situations it has seen, and it rarely explains or guarantees its decisions. For a problem like the courier's, where the rules are known and exact, a model states in a page what learning would need thousands of trials to approximate.

Approach	You provide	A new situation needs	Weak point
Hand-written behavior	The decisions	New code, if it was not foreseen	Variety
Hand-coded search	States, successors, heuristic, in code	New data; new code if the rules change	Reuse
Learning	Examples or a reward	More data, if it is far from what was seen	Guarantees
Planning	A model	A new problem file	The model must exist

Table 1.2.: Four ways to obtain behavior. Planning asks for a model and gives back flexibility: new situations are new data, not new code.

Planning fits problems with clear rules, many possible situations and goals that change. It asks for a model in exchange.

The four approaches are not rivals in a real system. In a robot they sit at different levels: learned perception finds the box, a hand-written controller keeps the arm steady, a path planner coded for geometry draws the route across the room, and a planner of the kind studied in this book decides which box to fetch first and where to take it. This last level, where decisions are about *what* to do and in which order, is often called **task planning**.

The limits of planning come from the same place as its strength. It needs a model: where the rules cannot be stated (how to grasp a wet glass), there is nothing to plan with. The plan is as good as the model. And planning is hard in the technical sense: even under the classical assumptions the search can take long, and a careless model makes it much longer. The criteria of the MODEL method exist to keep both dangers in sight.

1.5. Where planning is used

Space. A spacecraft far from Earth cannot wait for instructions: a radio signal to Mars takes minutes, and to the outer planets hours. In May 1999 the Remote Agent experiment let an on-board planner take control of NASA's Deep Space 1 probe: given goals from the ground, the software planned the activities of the spacecraft and executed them by itself. Planners also work on the ground side of space missions. The daily activities of the Mars rovers Spirit and Opportunity, which landed in 2004, were prepared by operators with the help of a planning tool that kept the plans consistent with the rovers' limits of time, energy and data.

Logistics. Moving goods with a fleet of vehicles, loading and unloading containers, assigning cranes and trucks, sequencing the operations of a factory: all these are planning problems, large ones. The domain called *logistics*, with packages, trucks, planes and cities, has been a standard test for planners since the first competition, and we will model it in Chapter 4.

Robotics. A service robot receives goals (bring this, tidy that, guide a visitor) that change all the time, in an environment that changes too. Writing every behavior by hand does not scale, so a task planner decides the sequence of actions and the rest of the robot's software carries each one out. The first planner in this line was built for a robot: STRIPS, written at the Stanford Research Institute for a mobile robot called Shakey Fikes and Nilsson, 1971. Today frameworks such as PlanSys2 Martín et al., 2021 connect PDDL planners with robots that run ROS 2: the model says what the robot can do, the planner decides, and each action of the plan is tied to the piece of code that performs it. We come back to this in Chapter 20.

Planning is used where the rules are known, the situations are many and the goals change: spacecraft, logistics, service robots.

Planners are also used in places with no robots at all: to configure and repair computer systems, to compose the steps of a business process, to control elevators, to generate the sequence of operations in a chemical plant, to find the moves of a puzzle. What these uses share is the shape of the problem: known rules, many possible situations, and a goal that changes from one day to the next.

STRIPS and a robot called Shakey

In 1971 Richard Fikes and Nils Nilsson, of the Stanford Research Institute, published STRIPS, the *Stanford Research Institute Problem Solver* Fikes and Nilsson, 1971. It was the planner of Shakey, a mobile robot that pushed boxes between rooms. STRIPS described each action by the facts it needs, the facts it makes true and the facts it makes false. The planner itself was soon surpassed, but that way of describing actions survived: more than fifty years later it is still the core of PDDL, and Chapter 2 teaches it under the same name.

1.6. Domain, problem and PDDL

Figure 1.2 showed the model in two files. The split is the first modeling idea of the book, so it deserves a careful look.

The **domain** describes a kind of world: what can be said about it and what can be done in it. The domain of the courier world says that robots are at places, that items are at places or carried by robots, and that a robot can move, pick and drop, with the condition and the effect of each action. It mentions no particular robot, room or item.

The **problem** describes one situation in that kind of world and one request: which robots, items and places exist, how the places are connected, where everything is at the start, and what is wanted. "Robot `r1` is at the dock, `meds1` is in the pharmacy, we want `meds1` in ward A" is a problem.

One domain serves many problems. The hospital writes the courier domain once; every request of every day is a new problem, generated in practice by a program from the hospital's database. Even the map belongs to the problem: the same domain serves a different floor or a different hospital.

The domain holds the rules of a kind of world: what can be true and what can be done. The problem holds one situation and one goal. One domain, many problems.

To give the flavor of the language, Listing 1.1 shows how the courier domain describes the action of moving. We do not need to understand the notation yet, only to notice its shape: a name, some parameters, a condition and an effect, written with many parentheses. The words that start with a question mark are variables: the same description serves any robot and any pair of connected places.

```
(: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)))
```

Listing 1.1: The action `move` of the courier domain, shown only for its flavor: Chapter 2 explains every symbol.

PDDL and its versions

The language of that listing is **PDDL**, the *Planning Domain Definition Language*. Until 1998 nearly every planner read its own input language, so comparing two planners meant rewriting every test problem. PDDL was created by Drew McDermott and a committee of researchers McDermott et al., 1998 as the common language of the first planning competition, held that year, and it has been the standard language of the field ever since. With a standard language, a model written once can be given to any planner, and a new planner can be tried at once on thousands of existing models.

PDDL has grown in versions. Each one added to the previous the means to describe a richer kind of world, and the versions map onto the parts of this book (Table 1.3 and Figure 1.4).

Version	Year	What it adds	Chapters
PDDL 1.2	1998	Actions with preconditions and effects; types; richer conditions (negation, *or*, *for all*, *exists*) and effects that depend on the situation	2 to 7
PDDL 2.1	2002	Numbers; a measure of plan quality; actions with duration; continuous change	8 to 12
PDDL 2.2	2004	Facts derived from others by rules; facts that change at known times	13
PDDL 3.0	2006	Preferences (goals that are wishes); constraints on the whole plan	14, 15
PDDL 3.1	2008	State variables whose value is an object; action costs	16, 9
PDDL+	2006	Processes and events: a world that changes by itself	17, 18

Table 1.3.: The family of PDDL versions. Each version contains the previous ones; PDDL+ is a branch that grows from PDDL 2.1. The year is that of the competition or publication that introduced the version.

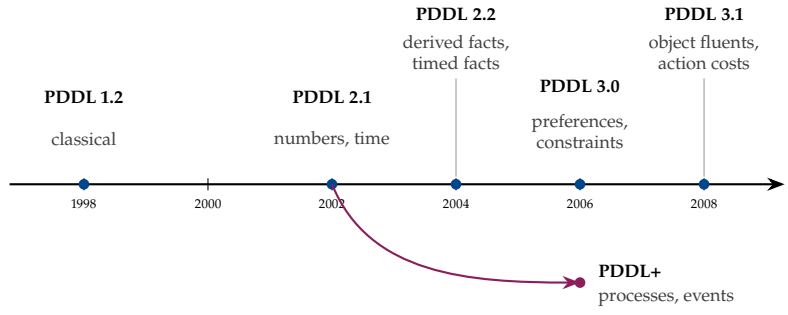


Figure 1.4.: Timeline of PDDL. The main line adds, version by version, what the classical assumptions leave out. PDDL+ branches from PDDL 2.1 toward worlds that change continuously and by themselves.

PDDL 1.2 is the language of classical planning: everything in it respects the five assumptions. PDDL 2.1 Fox and Long, 2003 lifted two of them at once, with numbers (the world is no longer finite) and with actions that last and overlap. PDDL 2.2 Edelkamp and Hoffmann, 2004 and PDDL 3.0 A. Gerevini and Long, 2005 made the world and the requests richer: facts that follow from other facts, facts that change at fixed times, wishes, rules for the whole plan. PDDL 3.1 Kovacs, 2011 added a more compact way to write state. PDDL+ Fox and Long, 2006 lifted the static

assumption fully: in it the world has its own dynamics, which the agent can only start, stop or wait for.

No planner is obliged to support everything. Each model declares which features of the language it uses (its **requirements**), and each planner documents which ones it accepts. Most planners cover a part of the family: many are purely classical, some are numeric, some temporal, a few handle PDDL+. This matters when modeling, and it is the reason why *portability* is one of the five criteria with which we will judge a model.

The International Planning Competition

The first planning competition was organized by Drew McDermott at the AIPS conference of 1998 McDermott, 2000, and PDDL was written so that all the competitors could read the same problems. The competition, known as the IPC, has been held again in 2000, 2002, 2004, 2006, 2008, 2011, 2014, 2018 and 2023. Several editions shaped the language: the 2002 edition introduced PDDL 2.1, the 2004 edition PDDL 2.2, the 2006 edition PDDL 3.0 and the 2008 edition PDDL 3.1. The competitions also left a large public collection of domains and problems, which planners are still tested on today.

1.7. Meeting Kairos

The planner of this book is Kairos Martín Rico et al., 2026, an open-source planner that reads every version of PDDL listed above. We use it from the command line. Appendix A explains the ways to install it; the shortest one is the Python package:

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

The folder `examples/ch01` of the book holds the courier model as two files: `courier-domain.pddl`, the domain, and `courier-p01.pddl`, the problem of Figure 1.1. In this chapter we do not open them. We give them to the planner, the domain first and the problem second:

```
$ 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)
```

That is a plan, and it arrived in a few milliseconds. It is worth learning to read it now, because we will read hundreds.

Each line in parentheses is one **step** of the plan: the name of an action followed by the objects it is applied to. The line `(move r1 dock lobby)` says: robot `r1` moves from the dock to the lobby. The order of the objects is the order of the parameters of the action in Listing 1.1: `?r`, `?from`, `?to`. The steps are carried out from top to bottom. The first line, which starts

with a semicolon, is a comment written by the planner: this plan has cost 7, and in a model that says nothing about costs the cost of a plan is its number of steps.

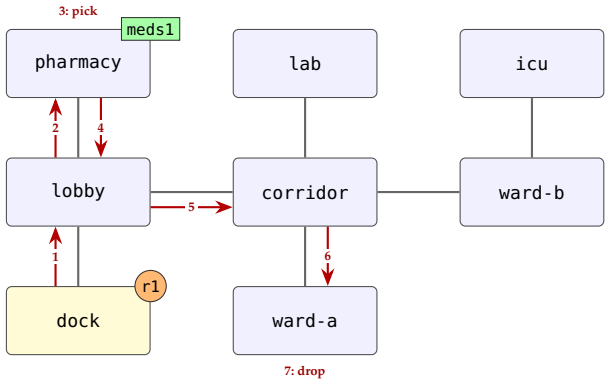


Figure 1.5.: The plan found by Kairos, drawn on the map. Follow the numbers: steps 1 and 2 take the robot away from ward A, because the medicines must be picked first (step 3). Steps 4 to 6 carry them to the ward, where step 7 leaves them.

Figure 1.5 traces the plan on the map. Checking it by hand takes a minute: each move joins two connected places and starts where the previous one ended, the robot is in the pharmacy when it picks the box, and it holds the box when it drops it in ward A. Nobody told the planner about pharmacies; it found the detour through the pharmacy because the model says that a robot picks an item only where the item is.

The planner can also check a plan for us. Kairos has a command for it, `validate`, which takes the domain, the problem and a file with a plan. The file `courier-p01.plan` contains the seven steps above:

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

Validating a plan means simulating it on the model, step by step, and checking that every step can be done when its turn comes and that the goal holds at the end. The word *makespan* is the time the plan takes; with instantaneous actions Kairos counts one unit per step. From now on, every plan printed in the book has passed this check.

Same domain, another problem

The file `courier-p02.pddl` describes another day on the same floor. The robot is in ward A, a blood sample waits in the ICU, and the lab needs it. The domain file is the same one:

```
$ 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)
```

Other objects, another start, another goal, and a new plan without touching the planner or the domain. This is the promise of Figure 1.2, kept.

A third problem, `courier-p03.pddl`, asks again for the medicines in ward A, but on a day when the ward is closed for cleaning: the problem file does not list the connection between the corridor and the ward.

```
$ 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)
```

There is no plan, and Kairos says so with the word **unsolvable**. This is an answer, and a valuable one: the planner has not given up, it has established that no sequence of actions, however long, delivers the medicines. The note above the verdict even explains why, in the vocabulary of the model: for the medicines to be in ward A somebody must drop them there, for that a robot must be in ward A, for that it must move there, and nothing is connected to ward A. Reading such explanations will become routine in Chapter 2.

A planner gives one of two useful answers: a plan, which can be validated, or the statement that no plan exists. Both come from the model, not from the world.

1.8. The MODEL method: five steps and five criteria

Running the planner was the easy part. Somebody wrote the domain file, and in doing so made decisions: that the floor is eight places and not a grid of square meters; that "carrying" is something to keep track of and "tired" is not; that picking needs the robot and the item in the same place. Each decision could have been taken differently. Writing a model is not translating a story into parentheses. *Modeling is choosing*, and this book teaches a method to choose well.

The method is called MODEL after the initials of its five steps (Figure 1.6). Every step answers a question and leaves something written.

M: Map the situation. What is the problem, who acts, what counts as solved, what is left out? The step produces the **model card**, a few lines with five headings: the *Story* in plain words, the *Agents* that act, what *Success* means, the *Scope* (what is in the model and what is out), and the *Assumptions* made about the world. No PDDL is written in this step. Most bad models are bad here: they solve another problem.

O: Objects and state. Which things exist, and what must the planner know about them to decide? The step produces the **state table**: every property that the model will track, what it is about, whether it can change, and how it is written.

D: Dynamics. What can be done or can happen, when, and what does it change? The step produces the **action table**: every action, what it acts on, when it can be done and what it changes.

E: Exercise the model. Does the planner do with the model what the world would allow, and nothing else? A model is a program and it has bugs. The step produces the **test set**: a trivial problem, a typical one, one that must have no plan, and one that tempts the planner to cheat. Every plan is read and validated.

L: Look again. Which other encoding would do, and which is better here? The step produces the **comparison**: two or three alternatives judged on five criteria, and a decision with its reason.

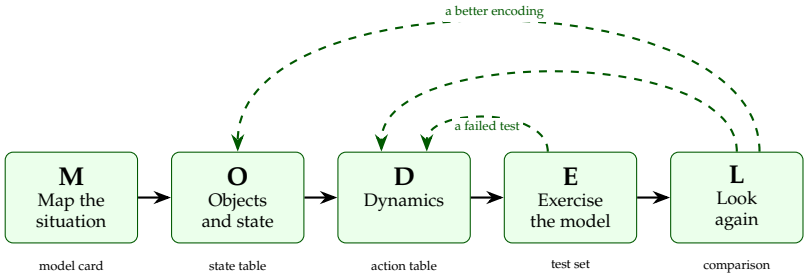


Figure 1.6.: The MODEL method. Solid arrows: the five steps in order, each with what it produces. Dashed arrows: the method is a loop. A failed test sends back to the dynamics; a better alternative found in step L sends back to the state or to the dynamics.

The first three steps go from the world to the model: decide what the problem is, then what exists, then what happens. The last two go from the model back to the world: check that the model behaves like the world, and ask whether a different model would behave better. The method is a loop. A test that fails in step E sends us back to step D, and the comparison of step L often sends us back to step O with a better idea of how to represent the state.

MODEL: Map the situation, Objects and state, Dynamics, Exercise the model, Look again. Each step answers a question and leaves a written result.

Step M on the courier world: a first model card

Here is the model card behind the files that we ran in this chapter.

Story	A robot carries items between the rooms of a hospital floor, on request.
Agents	One courier robot. It is the only one who acts.
Success	Each requested item lies in the room where it was asked for.
Scope	In: the rooms and their connections; where the robot and the items are; what the robot carries. Out: distances, time, battery, doors, people, how the arm grasps.
Assumptions	The five classical ones. The robot carries one item at a time.

Every line of the Scope is a decision. "Out: battery" means that this model will happily send the robot on a tour that empties it. If that matters, the card must change, and the model with it: Chapter 8 does exactly that.

The five criteria

Step L needs a way to say that one model is better than another. The method uses five criteria, always the same and always in this order.

1. **Fidelity**. The model has no impossible plans: every plan of the model can be carried out in the world. And it has no lost solutions: every reasonable way to solve the problem is a plan of the model. A

model that lets the robot carry ten boxes fails the first half; a model that forbids the robot to cross the lobby twice fails the second.
2. **Size**. How large the model becomes for the planner as the problem grows: how many concrete facts and actions result when the descriptions of the domain are applied to all the objects of the problem.
3. **Guidance**. How easy the model makes the search: whether it has dead ends, many equivalent ways to do the same thing, long detours, and how much of the problem the planner's estimates can see.
4. **Clarity**. How easy the model is to read, to change and to explain to somebody else, including yourself in six months.
5. **Portability**. Which planners accept the features of the language that the model uses.

Fidelity comes first because a model without it gives wrong answers quickly. The others are traded against one another: a compact model may be harder to read, a faithful one may need a feature that few planners support. No encoding wins on all five in every case, and that is why the last step ends with a decision *and its reason*.

Alternatives compared

Every chapter from now on ends its method section with a table like Table 1.4: two or three encodings of the same thing, written, run and judged on the five criteria. We cannot write PDDL yet, so this first comparison is made in words, on a decision that the model card already took: how to describe *where* things are. Encoding A is the one of this chapter: the floor is eight places. Encoding B divides the floor into square cells of one meter; for a floor of 40 by 25 meters that is a thousand cells, and the robot moves from a cell to a neighboring one.

Criterion	A: eight places	B: a thousand cells
Fidelity	Plans say which rooms to cross, not how; the way inside a room is left to the robot's navigation	Plans are exact routes; they are wrong as soon as a chair is moved
Size	8 places, 14 possible moves	1000 cells, about 4000 possible moves
Guidance	A delivery is 7 steps	The same delivery is dozens of steps, with a huge number of equivalent routes
Clarity	Places have the names that people use	Cell (17, 4) means nothing to a nurse
Portability	Classical	Classical

Table 1.4.: A first comparison, in words: two ways to describe where things are in the courier world.

Verdict. For deciding which item to fetch and in which order to visit the rooms, encoding A is better on every criterion that differs. B answers a different question, how to cross a room, and that question already has better tools: the robot's navigation software. Choose A when the planner decides *what* to do, and leave B's level of detail to the component that knows about geometry. The general lesson is the one of the model card: the right level of detail is the lowest one that still tells apart the decisions you need.

1.9. Common mistakes

The mistakes of a first contact with a planner are less about PDDL than about what to expect from the tool.

Giving the files in the wrong order

Kairos expects the domain first and the problem second. With the two files swapped, it reads the problem where it expects a domain and stops at the second line:

```
$ 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)
   |          ^
```

The message has the form of all the diagnostics of Kairos: file, line and column, what is wrong, a code in brackets, and the line with a mark under the place. Read the first error first.

Taking "unsolvable" for a failure of the planner

When the answer is `kairos: unsolvable (exhausted)`, as with `courier-p03.pddl`, the planner has not broken down, and running it again or for longer changes nothing: the model has no plan. Either the situation really has no solution (the ward is closed) or the model is wrong (somebody forgot to write a connection). The note that Kairos prints above the verdict says which fact cannot be reached; start there.

Expecting the planner to know what the model does not say

The plan of `courier-p01.pddl` ends with the robot standing in ward A. A person would send it back to its dock; the planner does not, because the goal asks only for the medicines in ward A. The planner has no common sense and no knowledge of hospitals: what is not in the model does not exist. If the robot must end at the dock, the goal has to say so.

Treating a plan as a program

A plan solves one problem: it is a sequence fitted to one initial state and one goal. The plan of the first problem is useless for the second one, and the validator rejects it; here it stops at the first step that names an object the second problem does not have:

```
$ 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
```

What is reused from one day to the next is the domain, never the plan. A new situation means running the planner again.

1.10. Summary

Planning is deciding, before acting, which actions to carry out and in which order so as to reach a goal. A planner is a general program that does this for any model it is given; the model is the part that we write, and its quality decides the quality of the plans. The ideas to keep from this chapter:

- A model has three ingredients: **states** (the initial one among them), **actions** with preconditions and effects, and a **goal**. A plan is a sequence of actions that leads from the initial state to a state that satisfies the goal.
- **Classical planning** assumes a finite, deterministic, fully known and static world with instantaneous actions. Parts III to VI of the book lift the first, the last two, and the limits on what a goal can be.
- Planning fits problems with clear rules and changing situations. Hand-written behavior, hand-coded search and learning fit others, and real systems combine them.
- A model comes in two files: the **domain** (a kind of world) and the **problem** (one situation and one goal). One domain serves many problems.
- **PDDL** is the standard language for models, born in 1998 with the planning competition. Its versions (1.2, 2.1, 2.2, 3.0, 3.1 and PDDL+) are the route of this book.
- **Kairos** takes a domain and a problem and answers with a plan or with "unsolvable". It also validates plans.
- Modeling is choosing. The **MODEL method** organizes the choices in five steps (Map the situation, Objects and state, Dynamics, Exercise the model, Look again) and judges alternatives on five criteria (fidelity, size, guidance, clarity, portability).

The next chapter opens the two files that we treated here as a black box, teaches the PDDL in them, and runs the five steps of the method on the courier world for the first time.

1.11. Further reading

The view of planning as a general solver plus a model, and its comparison with programming and with learning, follows the first chapter of Geffner and Bonet Geffner and Bonet, 2013, a short book that then explains how planners search. The standard textbook of the field is Ghallab, Nau and Traverso Ghallab et al., 2004; its first chapter discusses the classical assumptions (there called restrictive assumptions) one by one. The same authors' second book Ghallab et al., 2016 is about what this chapter left to Chapter 20: how planning connects with acting. Russell and Norvig Russell and Norvig, 2020 place planning inside artificial intelligence as a whole, next to search and learning.

On the language: the original PDDL manual McDermott et al., 1998 and McDermott's account of the first competition McDermott, 2000 tell why PDDL was made the way it is. Haslum, Lipovetzky, Magazzeni and Muise Haslum et al., 2019 give a compact reference of the whole language. The paper that started it all, by Fikes and Nilsson Fikes and Nilsson, 1971, is still readable today.

1.12. Exercises

Exercises for Section 1.1 — Deciding before acting

Exercise 1.1.1. For each task, say whether solving it needs planning as defined in this chapter (a sequence of interacting actions found before acting), and why: (a) a thermostat that switches the heating on below 19 degrees; (b) getting a car out of a crowded parking lot where other cars must be moved first; (c) sorting a list of numbers; (d) packing a suitcase so that the things needed first end on top; (e) braking when the car in front brakes.

Exercise 1.1.2. With the map of Figure 1.1 and paper only: the robot is at the dock, `meds1` is in the pharmacy and `sample1` is in the ICU. The medicines must go to ward B and the sample to the lab. Write a plan, as a list of steps (move, pick, drop) and count its steps. Then write the plan that makes the two deliveries in the other order, count again, and explain why one order is so much better than the other.

Exercises for Section 1.2 — A general solver plus a model

Exercise 1.2.1. Describe in words the three ingredients (what a state says, two or three actions with their condition and their change, and a goal) for: (a) making a cup of tea; (b) an elevator that serves the calls of a building; (c) the 8-puzzle, with eight sliding tiles in a 3 by 3 frame.

Exercise 1.2.2. The hospital of the courier world makes five changes. For each, say whether the planner, the description of what the robot can do, or the description of the situation has to change: (a) a new ward C opens next to the corridor; (b) the request of today is a meal for the ICU; (c) the robots receive a second tray and can carry two items; (d) a second robot is bought; (e) the hospital wants plans computed twice as fast.

Exercise 1.2.3. The goal "`meds1` is in ward A" is satisfied by many states. With one robot, one item and the eight places of the map, how many different states satisfy it? (A state says where the robot is and where the item is, or that the robot carries it.)

Exercises for Section 1.3 — The classical assumptions

Exercise 1.3.1. Each of these facts about a real hospital breaks one of the five classical assumptions. Say which one, and which part of the book deals with it according to Table 1.1: (a) crossing the corridor takes 40 seconds; (b) the pharmacy closes at 15:00; (c) one time in fifty the robot fails to grasp a box; (d) the battery level is a number between 0 and 100; (e) nobody knows on which shelf the sample was left; (f) two robots work at the same time.

Exercise 1.3.2. The concept box of the section claims that a wrong model can be useful. Give an example of a decision for which the eight-place model of the courier world is good enough although it ignores distances, and one decision for which ignoring distances gives a clearly bad answer.

Exercises for Section 1.4 — Planning and its alternatives

Exercise 1.4.1. Write, in pseudo-code, a hand-made program for the courier robot that serves requests of the form "take item *i* from place *a* to place *b*" on the map of Figure 1.1. You may assume a function `route(a, b)` that returns the list of places between two places. Then list what you would have to change in your program if (a) a request can name two items for the same destination and the robot has two trays; (b) some connections are closed on some days. Compare with what would change in a model.

Exercise 1.4.2. For each system, choose the approach of Table 1.2 that you would use first, and give your reason in one sentence: (a) a vending machine; (b) a robot arm that must recognize ripe tomatoes; (c) a warehouse where forty robots serve orders that arrive all day; (d) the solver of one fixed puzzle in a video game.

Exercises for Section 1.6 — Domain, problem and PDDL

Exercise 1.6.1. Say whether each statement belongs to the domain or to the problem of the courier world: (a) the lobby is connected to the corridor; (b) a robot can pick an item only if both are in the same place; (c) there are two robots; (d) `meal1` must end in ward B; (e) after a drop the robot carries nothing; (f) the robot starts at the dock.

Exercise 1.6.2. With Table 1.3, name the earliest version of PDDL that can express each requirement: (a) the robot has a battery that loses 2 units per meter; (b) the lab accepts samples only between 8:00 and 14:00; (c) it would be nice, but it is not required, that the robot ends at the dock; (d) moving takes time and two robots move at once; (e) a sample out of the fridge warms up by itself and spoils at 8 degrees; (f) the robot must never enter the ICU while it carries waste.

Exercises for Section 1.7 — Meeting Kairos

Exercise 1.7.1. At the keyboard. Run Kairos on `courier-domain.pddl` and `courier-p01.pddl` and check that you obtain the plan of the chapter. Copy the problem to a new file and, without reading any explanation of the syntax, change it so that the medicines must end in the ICU. Before running it, predict the number of steps of the plan from the map.

Exercise 1.7.2. At the keyboard. In a copy of `courier-p01.pddl`, delete the text `(free r1)` and run the planner. Read the answer of Kairos and explain, from it, what that piece of text was telling the planner.

Exercise 1.7.3. At the keyboard. Write by hand, in a file, a plan for `courier-p02.pddl` in the format that Kairos prints, and validate it. Then swap two lines of your file so that the robot picks the sample before reaching the ICU, validate again, and explain each part of the message.

Exercises for Section 1.8 — The MODEL method: five steps and five criteria

Exercise 1.8.1. Write a model card (Story, Agents, Success, Scope, Assumptions) for a robot vacuum cleaner that must clean the rooms of a flat and return to its base. Give at least three things that you

leave out of the scope, and for one of them describe a situation in which leaving it out produces a bad plan.

Exercise 1.8.2. Match each activity with the step of the method where it belongs: (a) deciding that the model will not track the battery; (b) writing a problem that should have no plan and checking that the planner agrees; (c) listing what moving changes; (d) deciding to record which item a robot carries; (e) rewriting the model without the pick action to see whether plans get shorter; (f) reading a plan line by line.

Exercise 1.8.3. A classmate proposes a third encoding for Table 1.4: the floor has only two places, "pharmacy side" (lobby, pharmacy, dock) and "ward side" (everything else). Judge it on the five criteria and give a verdict: is there a request for which it is good enough, and one for which it loses fidelity?

2. A first model: STRIPS

In Chapter 1 we gave two files to a planner and received a plan that took a box of medicines from the pharmacy to ward A. We could run the model, and we could not change it. Suppose the hospital now says that the robot has to come back to its dock after each delivery, or that the ICU is closed today, or that a second kind of robot arrives which carries two items. Each of these is a small change to the model, and a black box cannot be changed. To model we have to write.

This chapter teaches the PDDL needed to write a complete model, from an empty file: how the language is laid out, how to say what can be true in a world (predicates), how to say what can be done in it (actions, with their preconditions and effects), and how to describe one situation and one goal (objects, initial state, goal). The level of the language that we use is the oldest and simplest one, called STRIPS: conditions are lists of facts that must be true, and effects are lists of facts that become true or false. Everything that later chapters add is built on it.

The first four sections teach the syntax on the smallest domain that we could think of, a few lamps with their switches. With the syntax in hand, we run the whole MODEL method on the courier world for the first time: the model card, the state table and the action table lead to the domain file, which we then test with four problems designed to catch its mistakes. Two classic domains, the blocks world and gripper, show the same ideas away from the hospital. The method section collects what this first pass teaches and compares three encodings of the courier world, and the chapter ends, as all of them will, with the mistakes that everybody makes.

2.1 Lists inside lists: the shape of PDDL 20
2.2 Predicates, objects and facts 21
What can be said: predicates 21
What is true: facts and states 22
2.3 Actions: preconditions and effects 22
Applying an action 23
2.4 The problem: objects, initial state and goal . . . 24
2.5 Modeling the courier world 25
Step M: map the situation 25
Step O: objects and state . 26
Step D: dynamics 26
A problem for the domain 28
2.6 Exercising the model . . . 29
Check, solve, validate . . . 29
The test set 30
2.7 Two classics: blocks world and gripper 33
Blocks world 33
Gripper 35
2.8 The MODEL method: the first full pass 37
What this chapter adds . . 37
Alternatives compared . . 38
2.9 Common mistakes 40
2.10 Summary 42
2.11 Further reading 43
2.12 Exercises 43

2.1. Lists inside lists: the shape of PDDL

A PDDL file looks strange at first sight because of its parentheses. The strangeness goes away with one rule: everything in PDDL is a list in parentheses, and the first element of the list says what the list is. The elements are separated by spaces, and an element can be a word or another list.

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

The first line is a list of two words. We will read it as "`on` holds for `desk`": the desk lamp is on. The second is a list of three elements, the word `and` and two lists: both things hold. The third wraps the second in a list that starts with `:goal`: this is what is wanted. Lists written this way are called **s-expressions**, a notation that PDDL inherits from the programming language Lisp. Figure 2.1 draws the third one as a tree, which is how a planner sees it.

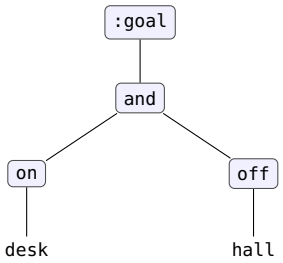


Figure 2.1.: The expression `(:goal (and (on desk) (off hall)))` as a tree. Each pair of parentheses is a node, named by the first word inside it; what follows the first word hangs below.

Four more rules complete the layout.

- **Layout is free.** Spaces, tabs and line breaks all count as one separator. We indent to show which list is inside which; the planner does not look at the indentation, it counts parentheses.
- **Comments** go from a semicolon to the end of the line.
- **Names** are made of letters, digits, hyphens and underscores, and start with a letter: `ward-a`, `meds1`, `robot-at`. PDDL does not tell upper case from lower case; this book writes everything in lower case, with hyphens between words.
- **Keywords** of the language that introduce a section start with a colon (`:goal`, `:action`), and **variables** start with a question mark (`?l`).

PDDL is written as lists in parentheses. The first word of a list says what the list is; the rest are its parts, which can be lists themselves.

A model is two files, as we saw: a domain and a problem. Each one is a single big list that starts with the word `define`:

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

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

The domain has a name, here `switch`. The problem has a name too, and says with `(:domain switch)` which domain it belongs to. The dots are the sections that the next three sections of the chapter fill in: predicates and actions in the domain; objects, initial state and goal in the problem.

2.2. Predicates, objects and facts

We need a first world, as small as possible. A room has some lamps: one in the hall, one on the desk, one on the porch. Each lamp is lit or dark, and it has a switch. That is all.

What can be said: predicates

The things that exist in a problem are its **objects**: here the three lamps, `hall`, `desk` and `porch`. A **predicate** is a property that an object can have, or a relation that several objects can be in. The domain declares its predicates in the section `:predicates`, each with a name and a list of variables, one per object that the predicate talks about:

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

This declares two predicates of one parameter. The variable `?l` stands for "any object": the declaration says only that `on` is said of one thing. Its name is free, and choosing a good one (`?l` for a lamp) is the documentation of the predicate, together with the comment. A predicate can have several parameters, like `(connected ?from ?to)` in the courier world, or none at all.

What is true: facts and states

A predicate applied to objects is a **fact**[1]: `(on desk)` is a fact, and so is `(off hall)`. A fact is true or false at each moment. With three lamps and two predicates there are six facts that can be written: `(on hall)`, `(on desk)`, `(on porch)`, `(off hall)`, `(off desk)`, `(off porch)`.

1: Other texts, and the messages of Kairos, say *atom* or *ground atom* for what we call a fact. *Ground* means "with objects, without variables".

In Chapter 1 a state was "a description of the world at one moment". Now we can be exact: a **state** is a set of facts, the ones that are true at that moment. The state in which only the hall lamp is lit is the set

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

Everything that is not in the set is false. This convention has a name, the **closed world assumption**: what is not stated to be true is false. There is no third value. A planner never thinks "I do not know whether the porch lamp is on": if `(on porch)` is not in the state, the porch lamp is not on.

A fact is a predicate applied to objects. A state is the set of facts that are true; every other fact is false (closed world assumption).

Two things about this definition deserve a second look.

First, the planner gives no meaning to names. For us `on` and `off` are opposites. For the planner they are two unrelated predicates, and the set `{(on hall),(off hall)}` is a perfectly good state in which the hall lamp is lit and dark at once. Nothing in the language forbids it. It is the model, through its initial state and its actions, that must never produce such a state. Keeping the facts of a model consistent with one another is a large part of the craft, and we will meet it in every chapter.

Second, a reasonable reader will ask why we need `off` at all, if a lamp that is not on is off. The answer is a limit of STRIPS, the level of PDDL of this chapter: a condition can ask for facts to be true, and it cannot ask for a fact to be false. To switch a lamp on we want the condition "the lamp is dark", and the only way to ask for it is a fact that is true exactly when the lamp is dark. Chapter 5 lifts this limit with negative preconditions, and compares both ways.

2.3. Actions: preconditions and effects

The lamps have switches. In PDDL what can be done is an **action**, declared in the domain with four parts: a name, its `:parameters`, its `:precondition` and its `:effect`. Listing 2.1 is the complete domain.

```
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)))))
```

Listing 2.1: The switch domain, complete: two predicates and two actions.

Line 3 is new. The section `:requirements` lists the features of the language that the file uses, so that a planner can tell at once whether it is able to read it. `:strips` is the basic level: every planner supports it. Each later chapter adds requirements to this list, and it will always be the first thing we write.

Take the action `switch-on`, in lines 9 to 13.

`:parameters` is the list of variables of the action: what it acts on. `switch-on` acts on one thing, `?l`. The action is a pattern: it describes at once the switching on of the hall lamp, of the desk lamp and of any lamp of any problem.

`:precondition` is what must be true for the action to be possible: here, that `?l` is off. With more than one fact, they are joined with `and`. In STRIPS a precondition is always this: a fact, or a list of facts that must all be true.

`:effect` is what the action changes. A fact written plainly becomes true: it is an **add effect**. A fact wrapped in `not` becomes false: it is a **delete effect**. After `switch-on`, `(on ?l)` is true and `(off ?l)` is false.

The variables of the precondition and of the effect must be parameters of the action. Replacing every parameter by an object gives a **ground action**, which is what a plan is made of: `(switch-on desk)` is `switch-on` with `desk` in the place of `?l`. With three lamps the two actions of the domain give six ground actions.

Applying an action

What an action does to a state is defined by two rules, and it is worth knowing them by heart.

1. A ground action is **applicable** in a state if every fact of its precondition is in the state.
2. **Applying** it gives a new state: take the old one, remove the facts of the delete effects, and put in the facts of the add effects. Every other fact stays as it was.

Figure 2.2 applies `(switch-on desk)` to the state in which only the hall lamp is lit. The precondition `(off desk)` is in the state, so the action is applicable. The delete effect removes `(off desk)`, the add effect puts in `(on desk)`, and the two facts about the other lamps pass untouched.

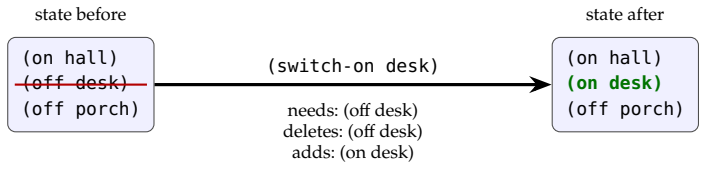


Figure 2.2.: Applying an action. The precondition is checked in the state before; the state after is the same set without the deleted fact (struck through) and with the added one (in green). The facts that the action does not mention are copied.

The last sentence of rule 2 is the reason why STRIPS actions are short. An action says only what changes. Switching on the desk lamp does not need to say that the hall lamp stays lit, that the porch lamp stays dark, or that the robot three rooms away is still where it was. In return, an action must say *everything* that changes: if `switch-on` forgot its delete effect, the desk lamp would end up on and off at once. We will call this double duty the frame rule and come back to it in the method section.

An action is applicable when its precondition facts are in the state. Applying it removes the delete effects and adds the add effects. Nothing else changes.

STRIPS, the planner and the level of PDDL

STRIPS was the planner of the robot Shakey Fikes and Nilsson, 1971, and its lasting idea was the way of describing an action that we have just seen: a precondition, a list of facts to add and a list of facts to delete. In PDDL the name survives as the requirement `:strips`, which means: preconditions and goals are conjunctions of positive facts, and effects are conjunctions of add and delete effects. It is the smallest useful subset of the language, and it is enough to write every model of this chapter.

2.4. The problem: objects, initial state and goal

The domain says what lamps are like. A problem says which lamps there are, how they are now and how we want them. Listing 2.2 is a problem for the switch domain.

```
;; 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))))
```

Listing 2.2: A problem of the switch domain: three lamps, one of them lit.

After its name and its domain, a problem has three sections.

`:objects` lists the names of the objects. They exist from this line on; nothing else exists.

`:init` is the initial state: the list of the facts that are true at the start. By the closed world assumption, every fact that is not listed is false. The initial state must be complete. If we forget `(off porch)`, the porch lamp is neither on nor off, and no action of the domain will ever touch it.

`:goal` is a condition that the last state must satisfy: in STRIPS, a fact or an `and` of facts that must be true. Unlike the initial state, the goal is partial. It says nothing about the porch lamp, so the porch lamp may end in any way.

The initial state is complete: what it does not list is false. The goal is partial: what it does not mention is free.

Before planning, we ask Kairos whether the two files are well written. The command `check` reads a domain, or a domain and a problem, and reports every mistake it finds without searching for a plan:

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

Valid means that the files are correct PDDL and agree with each other: every predicate used is declared, with the right number of arguments, and every object mentioned exists. *Plannable* means that Kairos supports all the features that the files use. Now the planner:

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

A plan is a sequence of ground actions, and we can now say exactly when it is right. Start from the initial state. `(switch-off hall)` needs `(on hall)`, which is there; applying it gives {`(off hall)`, `(off desk)`, `(off porch)`}. In that state `(switch-on desk)` needs `(off desk)`, which is there; applying it gives {`(off hall)`, `(on desk)`, `(off porch)`}. Both facts of the goal are in this last state. A **plan** is a sequence of ground actions in which each one is applicable in the state left by the previous ones, and whose final state contains the goal. The two steps could go in the other order here, and that would be another plan, as good as this one.

2.5. Modeling the courier world

We have all the syntax that a STRIPS model needs. Writing a good model takes more than syntax, and this is where the MODEL method starts to work. We now follow its five steps on the courier world, slowly, because this is the first time. The first three steps build the domain; the next section tests it.

Step M: map the situation

The model card of the courier world was written at the end of Chapter 1. We repeat it here, with the precision that we can now afford.

Step M: the model card of the courier world

Story	A robot carries items between the rooms of a hospital floor, on request.
Agents	The courier robots (one, in this chapter). Only they act.
Success	Each requested item lies in the room where it was asked for. Some requests also say where the robot must end.
Scope	In: the rooms and which ones are connected; where each robot and each item is; which item a robot carries. Out: distances, time, battery, doors, people, how the arm grasps, where exactly in a room things are.
Assumptions	The classical ones. A robot carries at most one item. Items do not move by themselves. Every connection can be crossed in both directions unless the problem says otherwise.

The card decides what the next two steps may talk about. “A robot carries at most one item” will turn into a predicate and two conditions. “Out: distances” means that no predicate will speak of meters, and that the planner will count every move as equal.

Step O: objects and state

Step O asks which things exist and what the planner must know about them to decide. The things of the card are of three kinds: robots, locations and items. For each kind we ask what can be said of it that matters for a delivery, and for each answer whether it can change.

A property that no action ever changes is **static**: the map is static, since moving, picking and dropping do not build corridors. A property that some action changes is **dynamic**: where the robot is. The difference matters to us, because static properties are data that we write once in the initial state, and it matters to the planner, which can treat static facts in a much cheaper way, as Chapter 3 will show. The answers go into the *state table* (Table 2.1).

Static properties never change: they are data of the problem. Dynamic properties are changed by some action, and every action must be checked against each of them.

Table 2.1: Step O: the state table of the courier world. Every row becomes a predicate of the domain.

Property	About	Kind	Encoded as
It is a robot / a location / an item	an object	static	`(robot ?x)` etc.
One can go from a location to another	two locations	static	`(connected ?from ?to)`
Where a robot is	a robot, a location	dynamic	`(robot-at ?r ?l)`
Where an item lies	an item, a location	dynamic	`(item-at ?i ?l)`
Which item a robot carries	a robot, an item	dynamic	`(carrying ?r ?i)`
The robot carries nothing	a robot	dynamic	`(free ?r)`

Three rows need a comment.

The first row exists because STRIPS, as we use it in this chapter, has no notion of kinds of objects. For the planner, `r1`, `lobby` and `meds1` are three names, and nothing prevents it from trying to move the lobby or to pick the robot. We tell the kinds apart with three predicates of one parameter, `(robot ?x)`, `(location ?x)` and `(item ?x)`, called **type predicates**. The problem states which object is of which kind, and each action asks for the kinds that it needs. Chapter 4 replaces them with real types.

The second row, `connected`, has a direction: `(connected lobby corridor)` says that one can go from the lobby to the corridor, and nothing about the way back. The planner does not know that corridors work both ways. A two-way connection is two facts, and the problem must list both.

The last row, `free`, is redundant for us: a robot is free exactly when it carries no item. We need it for the same reason as `off` for the lamps. The pick action must ask that the robot carries nothing, STRIPS can only ask for facts that are true, so "carries nothing" must be a fact. The price is that two predicates now tell overlapping stories, and the actions must keep them in step: whenever `carrying` appears, `free` must disappear, and the reverse.

Step D: dynamics

Step D asks what can be done, when, and what it changes. The card gives three verbs: a robot moves, picks and drops. For each one we fill a row of the *action table* (Table 2.2), in words first.

Action	Parameters	Applies when	Changes
`move`	robot, from, to	the robot is in *from*; *from* is connected to *to*	the robot is in *to*; it is no longer in *from*
`pick`	robot, item, location	robot and item are both in the location; the robot is free	the robot carries the item; the item no longer lies in the location; the robot is no longer free
`drop`	robot, item, location	the robot is in the location and carries the item	the item lies in the location; the robot is free; it no longer carries the item

Table 2.2.: Step D: the action table of the courier world. The column "Changes" lists, for each action, what becomes true and what stops being true.

The column of changes is where models go wrong, so we fill it with a fixed routine: for every dynamic row of the state table, ask whether the action makes it true or makes it false for some object. Take `pick`. Where the robot is: no change. Where the item lies: it stops lying in the location, since it is now on the robot. What the robot carries: the item. Free: no longer. Three of the four dynamic properties change, and the table shows the three.

The parameters come from the same routine: an action needs as parameters all the objects that its condition and its changes mention. That is why `pick` has the location as a parameter although "the robot picks the item" does not name it: the condition speaks of the place where both are, and the delete effect `(not (item-at ?i ?l))` must say which fact to remove.

Writing the table in PDDL is now almost mechanical. Listing 2.3 is the result: the file of Chapter 1, which we can finally read.

```
;; 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))))
```

Listing 2.3: The courier domain in STRIPS: the state table became the predicates, the action table became the actions.

Each action starts its precondition with the type predicates of its parameters, then the conditions of the action table. Each effect has the deletes

and the adds of the column "Changes". Compare `pick` and `drop`: one is the mirror image of the other, and what one adds the other deletes. Such symmetry is a good sign in a model, and its absence is worth a second look.

A problem for the domain

Listing 2.4 is the problem of Chapter 1: the medicines go from the pharmacy to ward A.

```
1  ;; Take the medicines from the pharmacy to ward A.
2  (define (problem courier-p01)
3    (:domain courier)
4    (:objects r1 meds1
5              lobby corridor pharmacy lab ward-a ward-b icu dock)
6    (:init
7      (robot r1) (item meds1)
8      (location lobby) (location corridor) (location pharmacy)
9      (location lab) (location ward-a) (location ward-b)
10     (location icu) (location dock)
11     ; the map: every connection in both directions
12     (connected lobby corridor) (connected corridor lobby)
13     (connected lobby pharmacy) (connected pharmacy lobby)
14     (connected lobby dock) (connected dock lobby)
15     (connected corridor lab) (connected lab corridor)
16     (connected corridor ward-a) (connected ward-a corridor)
17     (connected corridor ward-b) (connected ward-b corridor)
18     (connected ward-b icu) (connected icu ward-b)
19     ; where things are
20     (robot-at r1 dock) (free r1)
21     (item-at meds1 pharmacy))
22   (:goal (item-at meds1 ward-a)))
```

Listing 2.4: The first courier problem. Most of the initial state is static data: the kinds of the objects and the map.

The problem has ten objects, and its initial state has 27 facts. Of these, 24 are static: ten say the kind of each object and fourteen draw the map, each of the seven connections in its two directions. Only the last three, in lines 20 and 21, describe things that will change. This proportion is typical. A large part of a problem is the description of a fixed world, and the dynamic part, the one that a plan changes, is often small (Figure 2.3).

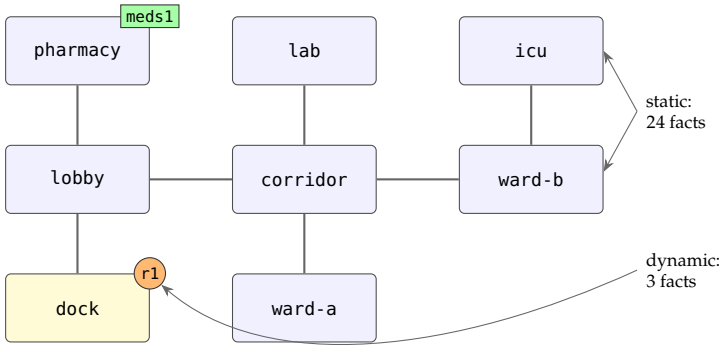


Figure 2.3.: Static and dynamic in `courier-p01.pddl`. The rooms and the lines between them never change: they are the 24 static facts. The robot and the item, drawn on top, are what the three dynamic facts describe and what a plan moves.

The goal is a single fact, `(item-at meds1 ward-a)`. It says nothing about the robot, so any final place of the robot is accepted.

2.6. Exercising the model

A model is a program, and a program that has only been written is probably wrong. Step E runs the model with one question in mind: does the planner do with it what the world would allow, and nothing else? The routine has three commands and four tests.

Check, solve, validate

First, `check`. It finds the mistakes of form before they turn into mysteries:

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

Second, solve. The plan is the one we know:

```
$ 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)
```

Third, read the plan. Reading means following it on the state, as we did with the lamps, and asking at each step whether the real robot could do it. Table 2.3 follows the dynamic facts; the static ones do not change and need not be written.

After	Dynamic facts of the state
(start)	`(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)`

Table 2.3.: The plan of `courier-p01.pddl`, read on the state. While the robot carries the medicines, they are in no room: `pick` deleted their `item-at` fact and `drop` adds a new one.

The trace shows a decision of the model that the listing hid: a carried item is nowhere. No fact `item-at` mentions it between the pick and the drop. That is a choice (another model could keep the item "in" the room that the robot crosses), and a good one, since nothing can then pick the item from the floor while the robot holds it.

The command `validate` does the same reading for us, on any plan in a file: one that a planner wrote, or one that we typed. The file `courier-p01.plan` holds the seven steps above, and `courier-p01-shortcut.plan` is a plan with a mistake: it goes from the pharmacy straight to the corridor.

```
$ 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
```

The message names the step, the fact of its precondition that is false and the line of the plan file. (The number after *at* is the time of the step: Kairos gives the steps of a sequential plan the times 0, 1, 2 and so on, and calls the total the *makespan*. Time becomes interesting in Chapter 10.) Validation is how we ask the model a precise question: "would you allow this?". We will use it in a moment to ask about a plan that should *not* be allowed.

The routine of step E: check the files, solve, read the plan on the state, and validate. A plan that is valid for the model can still be wrong for the world: that is what the tests look for.

The test set

One problem that gives a sensible plan proves little. The MODEL method asks for four kinds of tests, each aimed at a different kind of mistake.

Test 1: a trivial problem. A problem so small that we know its plan before running it. In `courier-p02.pddl` the robot starts in the pharmacy, next to the medicines, and must take them one room away, to the lobby.

```
    ; 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)
```

Three steps, one of each action, as expected. A trivial test catches the gross mistakes: an action that never applies, an effect on the wrong object. When it fails, the cause is easy to find because so little is involved.

Test 2: a typical problem. A problem like the ones the model is for. `courier-p01.pddl` is one. A richer one is `courier-p03.pddl`, with two deliveries and a robot that must go back to its dock:

```
    ; 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)
```

The plan delivers the medicines (steps 1 to 7), fetches the sample and takes it to the lab (steps 8 to 15) and returns to the dock (steps 16 to 18). The goal has three facts joined by `and`, and the third one, `(robot-at r1 dock)`, is what brings the robot home: in the first problem nobody asked, and the robot stayed in ward A. Reading a typical plan we look for steps that a person would not take. There are none here.

Test 3: a problem that must have no plan. A model can be wrong by allowing too much, and a test that expects a plan never notices. So we write a problem that is impossible in the world, and require the planner to say so. In `courier-p04.pddl` the ICU is closed: the initial state does not have the two `connected` facts that join it to ward B, and the goal wants the medicines there.

```
    (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)
```

The verdict is *unsolvable*: no plan exists. The note explains it by walking backwards from the goal, and with what we know now we can read every word. An *atom* is a fact. `(item-at meds1 icu)` is added only by `drop`, whose precondition includes `(robot-at ?r icu)` for some robot `?r`. (The text `- object` after a variable is its type, a notation of Chapter 4; here it means "any object".) A fact `(robot-at ?r icu)` is added only by `move`, which needs `(connected ?from icu)` for some place, and no action adds

a `connected` fact: it is static, and the initial state does not have one that ends in the ICU. This is the explanation that we would have given.

Test 4: a problem that tempts the planner to cheat. The most useful test, and the one that is specific to modeling. A planner looks for the shortest way to the goal that the model permits. If the model has a hole, the planner will find it and go through. So we write a problem in which breaking a rule of the world would pay off, and check that the plan does not break it. The rule to test in the courier world is "one item at a time". In `courier-p05.pddl`, two boxes of medicines wait in the pharmacy and both go to ward 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))))
```

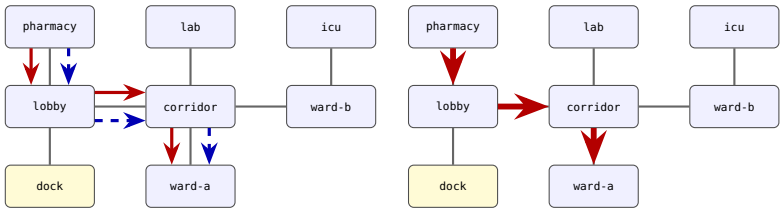
A robot that cheats makes one trip with both boxes: 2 moves to the pharmacy, 2 picks, 3 moves to the ward, 2 drops, 9 steps. An honest robot makes two trips: 15 steps (Figure 2.4). The length of the plan tells us which one the model allows.

```
$ 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)
```

Fifteen steps and two trips: the model holds. We can ask it directly, too. The file `courier-p05-cheat.plan` is the dishonest plan of 9 steps, typed by hand:

```
$ 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
```

The second pick is refused because `(free r1)` is false: the first pick deleted it. The predicate that looked redundant in the state table is what keeps the robot honest.



honest: two trips, 15 steps

cheating: one trip, 9 steps

Figure 2.4.: The cheat test. Left: the loaded trips of an honest plan, one per box (solid, then dashed). Right: the single loaded trip of a plan that carries both boxes. A model that allows the plan on the right has a hole, and the planner will find it because the plan is shorter.

A variant asks for the forbidden state itself. In `courier-p06.pddl` the goal is `(and (carrying r1 meds1) (carrying r1 meds2))`, and the right answer is again that no plan exists:

```
$ 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)
```

The reason is of a different kind than in test 3. There, a fact of the goal could never become true. Here each fact of the goal can become true, and the two never hold together; Kairos finds it out by visiting every state that the robot can reach.

Step E: the test set of the courier world

Test	File	Expected, and obtained
Trivial	`courier-p02`	3 steps: pick, move, drop
Typical	`courier-p01, p03`	7 and 18 steps; every step sensible
No plan	`courier-p04`	unsolvable: the ICU cannot be reached
Cheat	`courier-p05, p06`	15 steps (not 9); unsolvable

The test set stays with the model. Every time the domain changes, in this chapter and in the following ones, the four tests run again: `tools/check_examples.sh` does it for all the examples of the book.

2.7. Two classics: blocks world and gripper

The ideas of the chapter do not depend on hospitals. Two domains that every planning student meets show them again, each with a lesson of its own.

Blocks world

Some blocks lie on a table. A block can be on the table or on top of another block, and a hand moves them, one at a time: it can take a block only if nothing is on it. A problem gives the blocks in some arrangement and asks for another one (Figure 2.5).

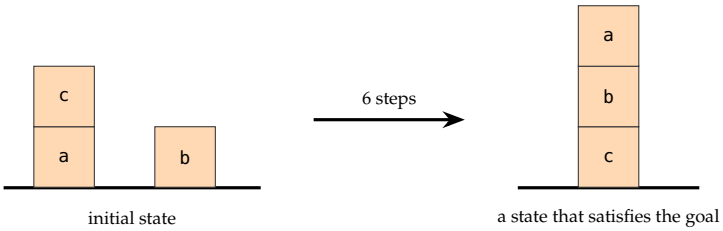


Figure 2.5.: The problem `blocks-p01.pddl`. The goal asks for `a` on `b` and `b` on `c`. Block `c` starts on top of `a` and must end at the bottom, so the first thing to do has nothing to do with either part of the goal: put `c` on the table.

Listing 2.5 is the usual STRIPS model, with four actions: two for the table (`pick-up`, `put-down`) and two for towers (`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))))

  (:action put-down             ; leave the block on the table
    :parameters (?x)
    :precondition (holding ?x)
    :effect (and (on-table ?x) (clear ?x) (hand-empty)
                 (not (holding ?x))))

  (:action unstack              ; take ?x from the top of ?y
    :parameters (?x ?y)
    :precondition (and (on ?x ?y) (clear ?x) (hand-empty))
    :effect (and (holding ?x) (clear ?y)
                 (not (on ?x ?y)) (not (clear ?x))
                 (not (hand-empty))))

  (:action stack                ; leave ?x on top of ?y
    :parameters (?x ?y)
    :precondition (and (holding ?x) (clear ?y))
    :effect (and (on ?x ?y) (clear ?x) (hand-empty)
                 (not (holding ?x)) (not (clear ?y)))))
```

Listing 2.5: Blocks world in STRIPS with four actions.

Three things to notice. The predicate `(hand-empty)` has no parameters: there is one hand, so the fact speaks about the world and not about any object. The predicate `(clear ?x)` is another fact about absence, like `free` and `off`: nothing is on `?x`. And the domain has no type predicates, because every object of a problem is a block: when there is only one kind of thing, there is nothing to tell apart.

```
;; 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))))
```

Listing 2.6: A blocks problem with three blocks.

```
$ 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)
```

Follow `clear` through the plan, because it is the delicate part of the model. At the start `b` and `c` are clear. `(unstack c a)` makes `a` clear and `c` not clear (the hand holds it; a held block cannot receive another). `(put-down c)` makes `c` clear again. `(stack b c)` takes it away for good. Each action that puts a block somewhere or takes it from somewhere must update `clear` for the block below, and every one of the four does. The blocks world is small, and it has more bookkeeping per action than the courier world: this is why it is a favorite for showing what a forgotten effect does.

The Sussman anomaly

The problem of Figure 2.5 has a name. It is the *Sussman anomaly*, after Gerald Sussman, who studied it at MIT in the early 1970s. The planners of that time solved a goal with two parts by solving one part and then the other. Here both orders fail: building `a` on `b` first leaves `b` unable to go onto `c` without undoing the work, and putting `b` on `c` first buries `a` deeper under `c`. The only solutions interleave the two parts. A three-block puzzle showed that the parts of a goal cannot in general be planned one by one.

The domain also gives us a third kind of unsolvable problem. The file `blocks-p02.pddl` keeps the same blocks and asks for `(and (on a b) (on b a))`: each block on top of the other.

```
$ 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)
```

A STRIPS model has no place to write the laws of its world. They hold because no action, applied to a sensible state, produces a state that breaks them.

Nothing in the domain says "a block cannot be on a block that is on it". The model does not need to say it: the actions are written so that such a state is never produced. This is how a STRIPS model expresses the laws of its world, through what its actions can and cannot do.

Gripper

A robot with two grippers carries balls from a room to another. The domain comes from the first planning competition McDermott, 2000. It looks like the courier world on purpose: we use it to see the same problem modeled with other decisions.

```
;; 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)))))
```

Listing 2.7: The gripper domain in STRIPS.

Compare it with Listing 2.3.

- There is one robot, so the robot is not an object: `(at-robby ?r)` has only the room as a parameter, where the courier world has `(robot-at ?r ?l)`. A model with fewer parameters is smaller, and it cannot grow to two robots without being rewritten.
- The hands are objects, `left` and `right`. "Carrying at most two" is expressed by two grippers that hold at most one ball each: `(empty ?g)` plays for a gripper the part that `(free ?r)` plays for a courier robot. A capacity can be modeled by giving objects to the places where things are held.
- `move` has no `connected` in its precondition: the robot goes from any room to any room. There is no map, and so no static relation in the problem besides the type predicates.

The problem `gripper-p01.pddl` has two rooms and four balls that must go from `room-a` to `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)
```

Two plans came out. Kairos prints a first plan as soon as it finds one, then keeps searching, and prints every plan that is better than the previous one; the last plan printed is the best one found. The first plan, of 13 steps, carries the first two balls one at a time. The second, of 11 steps, always fills both grippers. Chapter 3 explains why a planner works this way; with the option `--no-anytime` it stops at the first plan. For now, the rule is to read the last plan.

The 11-step plan uses the two grippers in every trip, and nobody told the planner to do so. In the courier world the cheat test checked that the robot cannot carry two items. Here carrying two is legal, the model says so through its two grippers, and the planner takes advantage of it. The cheat test of gripper is the same idea one step further: three balls in one trip must be impossible.

2.8. The MODEL method: the first full pass

We have gone once through steps M, O, D and E on the courier world. This section states what the pass has added to the method and then takes the last step, L.

What this chapter adds

The artifacts exist. The model card, the state table, the action table and the test set are no longer names from Chapter 1: they are the method box of step M, Table 2.1, Table 2.2 and the method box of step E. Later chapters will not repeat them whole. They will say what changes in them.

Static versus dynamic. Step O now asks, of every property: can any action change it? If not, it is static: it is written once in the initial state, it appears in preconditions and never in effects. If yes, it is dynamic, and step D owes it an answer for every action. A quick test of a finished domain is to list the predicates that appear in some effect: these are the dynamic ones, and the list should match the column "Kind" of the state table. In the courier domain they are `robot-at`, `item-at`, `carrying` and `free`, as planned.

The frame rule. The effects of an action must *say everything that changes, and nothing that does not*. The second half is a gift of the language: what an effect does not mention stays as it was. The first half is the modeler's duty, and it is easy to fail for the changes that are not the point of the action. The point of `pick` is that the robot carries the item. That the item is no longer on the shelf, and that the robot is no longer free, are consequences that the world gives for free and the model does not. The routine of step D (go through every dynamic property, for every action) exists to catch them.

The frame rule: the effects of an action say everything that changes and nothing that does not.

Questions that this chapter adds to the method

- **M** Can I write the five lines of the model card before any PDDL? Which rules of the world ("one item at a time") does the card state?
- **O** Is this property static or dynamic? Is it about absence (`free`, `clear`, `off`) and so needs a fact of its own? Do two predicates overlap, and who keeps them in step?
- **D** For each action and each dynamic property: does the action make it true, make it false, or leave it? Does every fact that becomes true replace one that must be deleted?
- **E** Do `check`, solve and `validate` pass on a trivial problem, a typical one, one with no plan, and one that tempts the planner to cheat?
- **L** Which other encoding would do? (Next subsection.)

Alternatives compared

Step L asks which other encoding would do. We compare the courier domain, encoding A, with two others that a reasonable modeler could have written. All three are files of `examples/ch02`, and all three were run on the cheat test, the problem of two boxes for ward A.

Encoding B: no pick and no drop. If all that matters is where the items end, the grasping can be left out. `courier-carry-domain.pddl` has no `carrying` and no `free`. Besides `move`, it has one action that moves the robot and an item together:

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

The rule "one item at a time" needs no predicate now: each `carry` takes one item along, by construction.

```
$ 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)
```

Eleven steps for the two trips. The plan is honest and has a surprise: it leaves the first box in the corridor, goes back for the second, and finishes both from there. In this model putting an item down and taking it again costs nothing, because those actions do not exist, so relaying the boxes through the corridor is as good as any other way.

Encoding C: one predicate for every position. A robot in a room and an item in a room are both "something somewhere". `courier-at-domain.pddl` is encoding A with the predicates `robot-at` and `item-at` merged into one, `(at ?x ?l)`. Its plan for the cheat test is, step for step, the plan of encoding A. The difference shows when the type predicates are taken away. In encoding A they are almost a formality: only a robot can appear in a `robot-at` fact, so `move` could hardly be applied to anything else. In encoding C they carry the weight. The file `courier-at-notypes-domain.pddl` is encoding C without them, and this is its plan for the delivery of `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)
```

The medicines walk to the ward by themselves. `(at meds1 pharmacy)` looks to `move` exactly like `(at r1 dock)`, and nothing says that `meds1` is not a robot.

Table 2.4 judges the three encodings on the five criteria. The sizes are for the cheat test; the number of ground actions is the one that Kairos reports with the option `-v`, which Chapter 3 teaches to compute by hand.

Criterion	A: `pick`, `drop`	B: `carry`	C: one `at`
Fidelity	One item at a time, enforced by `free`. Plans say when to grasp and release	One item at a time, by construction. Grasping is invisible: the executor must work it out; no fact says that a robot holds an item	As A, but only while every action keeps its type predicates
Size	8 predicates, 3 actions; 46 ground actions	6 predicates, 2 actions; 42 ground actions; grows with items times connections	7 predicates, 3 actions; 46 ground actions
Guidance	Plan of 15 steps	Plan of 11 steps; many equivalent relays	As A
Clarity	Reads like the story	Shortest file; plans need interpreting	One name less; the kinds of things are harder to see
Portability	`:strips`	`:strips`	`:strips`

Table 2.4.: Step L: three STRIPS encodings of the courier world, compared on the cheat test (`courier-p05` and its versions for B and C).

A word on the row Size. In encodings A and C each of `pick` and `drop` yields one ground action for each robot, item and location. In B, `carry` yields one for each robot, item and connection. On our map, with 8 locations and 14 connections in the two directions, the totals are close. On a map where every room has many neighbors, B grows faster.

Verdict. Choose B when the only question is which items go where and in how many trips, and nothing in the world depends on holding: it is the smallest model and its plans are the shortest. Choose C when many kinds of things share the property of being somewhere and the actions treat them alike; with the real types of Chapter 4 its weakness disappears. For the courier world of this book we keep A. Its plans say everything that the robot must do, grasping included, and the fact `(carrying ?r ?i)` is one that later chapters need: a sample that is contaminated while it is

carried, a load that weighs on the battery, a cart that is pushed. B would have to be thrown away at the first of them.

2.9. Common mistakes

Each mistake of this section is a file of `examples/ch02` that is wrong on purpose, with what Kairos does when it meets it.

A forgotten delete effect

The most frequent mistake of STRIPS modeling, and a silent one. In `courier-nodelete-domain.pddl` the action `move` adds the new position and forgets to delete the old one:

```
  (:action move
    :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` finds nothing wrong, because nothing is wrong with the form. The cheat test gives it away:

```
$ 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)
```

Eight steps where 15 are needed. After its moves the robot is in the dock, the lobby, the pharmacy, the corridor and ward A at once: it leaves a copy of itself in every room. It then picks in the pharmacy and drops in the ward without moving. *Symptom*: plans that are too short, in which something is in two places. *Fix*: for every add effect of a dynamic property that has one value at a time, write the delete of the old value.

A missing precondition

In `courier-nofree-domain.pddl` the action `pick` does not ask for `(free ?r)`. Again `check` is satisfied, and again the cheat test is not:

```
$ 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)
```

This is the 9-step plan that Figure 2.4 warned about. On the typical problems `courier-p01` and `courier-p03` this wrong domain gives the same plans as the right one: only a test built to tempt the planner

shows the hole. *Symptom*: a plan that does something that the model card forbids. *Fix*: find the step that should not have been possible, and add to its action the precondition that the world imposes.

A misspelled predicate

In `courier-typo-domain.pddl` the precondition of `move` says `robot-in` where it should say `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 warns and goes on: for it, a predicate that is not declared is a fact that is never true, so `move` is never applicable and every delivery is unsolvable. Other planners may treat the same file as an error and stop. *Symptom*: a warning from `check`, and problems without a plan whose explanation mentions an action that "never" applies. *Fix*: read the warnings of `check`, not only its verdict.

Unbalanced parentheses and stray variables

A missing parenthesis at the end of `move` (`courier-paren-domain.pddl`) is detected where the next section begins, not where the parenthesis is missing:

```
$ 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
```

Look *above* the line of the message. An editor that highlights matching parentheses saves most of these. A variable that is used and is not a parameter (`courier-var-domain.pddl` writes `?loc` for `?l` in an effect) is reported at its place:

```
$ 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
```

A problem that does not match its domain

`courier-bad-p01.pddl` has three mistakes in its initial state: a place that is not in `:objects`, a fact with too few arguments and a predicate that does not exist.

```
$ 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
```

The unknown object is an error. The other two are warnings, and their text says what Kairos does: *the fact is ignored*. An ignored fact of the initial state is, by the closed world assumption, a false fact: the robot is nowhere and the medicines are nowhere. *Symptom*: an unsolvable problem that looks solvable. *Fix*: run `check` with the domain *and* the problem, and leave no warning unread.

A goal that is already true

When the goal holds in the initial state, the plan is empty. `courier-p07.pddl` asks for the medicines in the pharmacy, where they already are:

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

A plan of cost 0 with no steps is a correct answer, and usually the sign of a goal copied from the wrong line or of a test that tests nothing. *Fix*: for every test, know the expected length of the plan before running it.

2.10. Summary

A STRIPS model says what can be true with predicates and what can be done with actions; a problem gives the objects, the initial state and the goal. Table 2.5 collects the PDDL of the chapter.

Construct	Meaning
`(define (domain d) ...)`	A domain file, named `d`
`(:requirements :strips)`	The features of the language that the file uses
`(:predicates (p ?x ?y) ...)`	What can be said: each predicate with its parameters
`(:action a ...)`	Something that can be done
`:parameters (?x ?y)`	The variables of the action
`:precondition (and ...)`	Facts that must be true for the action to apply
`:effect (and f (not g))`	Facts that become true (add) and false (delete)
`(define (problem p) ...)`	A problem file, named `p`
`(:domain d)`	The domain that the problem belongs to
`(:objects a b c)`	The objects that exist
`(:init ...)`	The facts true at the start; all others are false
`(:goal (and ...))`	Facts that must be true at the end; the rest is free

Table 2.5.: The PDDL of this chapter: all of a STRIPS model.

The ideas behind the syntax:

- A **fact** is a predicate applied to objects; a **state** is the set of the facts that are true. What is not in the state is false: the **closed world assumption**.
- An action is **applicable** when the facts of its precondition are in the state. Applying it removes its delete effects and adds its add effects; everything else stays.
- A **plan** is a sequence of ground actions, each applicable after the previous ones, that ends in a state containing the goal.
- STRIPS conditions ask only for true facts. Absence needs its own fact (`free`, `off`, `clear`), and the actions must keep it in step with the rest.
- **Static** facts are data: in the initial state, in preconditions, never in effects. **Dynamic** facts are what plans change.
- The **frame rule**: an effect says everything that changes and nothing that does not.
- Step E is a routine: `kairos check`, solve, read, `kairos validate`, on a **test set** with a trivial problem, a typical one, one without a plan and one that tempts the planner to cheat.

We have used the planner without asking how it finds its plans or why it sometimes prints two. The next chapter opens that box: what a planner does with a model decides which models are easy for it, and so it is part of modeling.

2.11. Further reading

The description of actions by preconditions, add lists and delete lists comes from the STRIPS paper of Fikes and Nilsson Fikes and Nilsson, 1971. The syntax is that of the first PDDL manual McDermott et al., 1998, where `:strips` opens the list of requirements. Kovacs Kovacs, 2011 gives the complete grammar of the current language in a few pages, useful when a doubt about syntax arises. Haslum et al. Haslum et al., 2019 devote their second chapter to the same subset of the language, with a formal definition of states, actions and plans that complements the informal one given here, and Ghallab, Nau and Traverso Ghallab et al., 2004 present it as the "classical representation" next to two equivalent ones. Gripper and the other domains of the first competition are described by McDermott McDermott, 2000. The validation of plans as a tool of its own started with VAL Howey et al., 2004, the validator of the planning competitions, whose plan format Kairos reads and writes.

2.12. Exercises

Exercises for Section 2.1 — Lists inside lists: the shape of PDDL

Exercise 2.1.1. Each of these lines is meant to be a piece of PDDL and has one mistake of form. Find it and write the line correctly: (a) `(and (on desk) (off hall);` (b) `(on, desk);` (c) `(:goal and (on desk) (off hall));` (d) `(on desk) ; (off hall))`. Then draw the tree of the corrected line (a) in the style of Figure 2.1.

Exercises for Section 2.2 — Predicates, objects and facts

Exercise 2.2.1. The switch domain with three lamps has six possible facts. (a) How many different sets of facts are there? (b) How many of them describe a situation that makes sense, with each lamp either lit or dark? (c) Give one set that makes no sense and say which line of a problem file could produce it.

Exercise 2.2.2. A state of a courier problem is {`(robot-at r1 lab)`, `(carrying r1 sample1)`, `(item-at meds1 lobby)`} plus the static facts of `courier-p01.pddl`. Under the closed world assumption, say whether each fact is true, false or unknown: (a) `(free r1)`; (b) `(item-at sample1 lab)`; (c) `(connected lab corridor)`; (d) `(connected lab icu)`; (e) `(robot-at r1 corridor)`.

Exercises for Section 2.3 — Actions: preconditions and effects

Exercise 2.3.1. By hand, with the courier domain of Listing 2.3. The dynamic facts of a state are `(robot-at r1 lobby)`, `(free r1)`, `(item-at meds1 lobby)` and `(item-at sample1 lab)`; the static ones are those of `courier-p01.pddl`, with `sample1` declared as an item. (a) Which of these ground actions are applicable: `(move r1 lobby lab)`, `(pick r1 meds1 lobby)`, `(pick r1 sample1 lab)`, `(drop r1 meds1 lobby)`, `(move r1 lobby dock)`? For each one that is not, name the precondition fact that fails. (b) Write the state after applying `(pick r1 meds1 lobby)` and then `(move r1 lobby corridor)`.

Exercise 2.3.2. A student writes `pick` with the effect `(and (not (free ?r)) (carrying ?r ?i))`. Apply this action by hand to the initial state of `courier-p02.pddl` and then finish the delivery with a move and a drop. What is strange in the final state? Which test of the test set would notice, and how?

Exercises for Section 2.4 — The problem: objects, initial state and goal

Exercise 2.4.1. At the keyboard. Write a problem for the switch domain with the initial state of `switch-p01.pddl` in which the three lamps must end lit. Predict the plan, then run it.

Exercise 2.4.2. At the keyboard. Remove `(off porch)` from the initial state of `switch-p01.pddl` and change the goal to `(on porch)`. Predict the answer of the planner, run it, and explain the note that Kairos prints.

Exercise 2.4.3. The lamps hang from a master switch: a lamp can be switched on only while the power is on. Extend the switch domain with what is needed (the power can be on or off, and an action turns it on), and write a problem in which the power starts off and two lamps must end lit. Hint: a predicate may have no parameters, and an action too: `:parameters ()`.

Exercises for Section 2.5 — Modeling the courier world

Exercise 2.5.1. At the keyboard. Write a problem for the courier domain: the robot starts in the corridor; `sample1` is in the lab and must go to the ICU; `meal1` is in the lobby and must go to ward B. Before

running it, find by hand the shortest plan that you can and count its steps. Compare with the last plan that Kairos prints.

Exercise 2.5.2. The door of ward A is changed for one that opens only inwards: the robot can enter the ward and cannot leave it. Which file changes, the domain or the problem, and which facts? Make the change in a copy of `courier-p01.pddl`, ask also that the robot ends at the dock, and explain the answer of the planner.

Exercise 2.5.3. Add to the courier domain an action `hand-over`: a robot that carries an item gives it to a free robot that is in the same location. Fill its row of the action table first, going through the four dynamic properties, and then write the PDDL. Test it with a problem in which `r1` holds `meds1` in the pharmacy, `r2` is free in the lobby, and the goal is `(carrying r2 meds1)`.

Exercises for Section 2.6 — Exercising the model

Exercise 2.6.1. At the keyboard. Write in a file a plan for `courier-p02.pddl` that is valid and has more steps than the one Kairos finds, and validate it. Then write the plan `(move r1 pharmacy lobby)`, `(drop r1 meds1 lobby)`, predict what the validator will say and at which step, and check.

Exercise 2.6.2. Design a test set for the switch domain: a trivial problem, a typical one, one that must have no plan and one that tempts the planner to cheat. For each, state the expected result before running it. (For the last two, think about what is impossible for a lamp.)

Exercises for Section 2.7 — Two classics: blocks world and gripper

Exercise 2.7.1. At the keyboard. Write a blocks problem with four blocks: `d` on `c`, `c` on `b`, `b` on `a`, `a` on the table. The goal is the same tower upside down. Predict the number of steps and run it.

Exercise 2.7.2. In a copy of the blocks domain, remove the delete effect `(not (clear ?y))` from `stack`. (a) Predict a goal that becomes reachable and that no real set of blocks can satisfy. (b) Write it as a problem and run it with the wrong domain and with the right one. (c) Which of the four kinds of tests is your problem?

Exercise 2.7.3. Model from this story, following steps M to E: *a ferry carries cars between ports. It can sail from any port to any other. It takes one car at a time: a car boards the ferry in the port where both are, and leaves it in the port where the ferry is.* Write the model card in three lines, the state table, the action table, the domain, and a problem in which two cars in different ports must swap ports.

Exercises for Section 2.8 — The MODEL method: the first full pass

Exercise 2.8.1. For the blocks domain and for the gripper domain, list the static and the dynamic predicates, using the test of the method section (does the predicate appear in some effect?). Then, for `unstack`, go through every dynamic predicate of the blocks domain and say whether the action makes it true, false, or leaves it, and for which objects.

Exercise 2.8.2. Step L on the blocks world. An alternative encoding has no hand: three actions move a clear block in one step, from a block

to another block, from a block to the table, and from the table to a block. Write it, solve the problem of Figure 2.5 with it, and compare it with the four-action domain on the five criteria. Use `kairos -v` to see the number of ground actions of each.

Exercises for Section 2.9 — Common mistakes

Exercise 2.9.1. Find the bug. A courier domain is identical to Listing 2.3 except for the effect of `drop`, which is `(and (not (carrying ?r ?i)) (item-at ?i ?l))`. (a) Without running anything, say which of the tests `courier-p01` to `courier-p05` still give the right answer and which do not. (b) What does the planner answer for the ones that fail? (c) This mistake and the missing precondition of the pitfall box are both about `free`. Why does one produce plans that are too short and the other no plans at all?

What a planner does with your model

3.

In the previous chapter we wrote a model of the courier world, gave it to Kairos and got a plan back in a few milliseconds. The planner was a black box, and for one robot and one box of medicines a black box is enough. It stops being enough very soon. Give the same domain four deliveries and two robots: the first plan still comes in milliseconds, but the planner then works for three seconds, on more than half a million states, before it says that no better plan exists. Remove one precondition by mistake, and the number of actions the planner handles goes from 30 to 80 while the plans become impossible to carry out. A modeler who does not know what happens inside the box cannot tell why, and cannot choose between two models that look equally correct.

This chapter opens the box. It first makes precise what a model *means*: which states exist, when an action can be applied, what it produces, and what counts as a plan. Then it follows the model through the planner: how the actions with variables become actions about concrete objects, how many states that creates, how a search finds a path among them, and how a heuristic, computed from the model itself, tells the search where to look. Nothing here is proved; everything is shown on problems small enough to draw or to count by hand, and then measured with Kairos.

The chapter has nine sections. The first three define the meaning of a model, the grounding of its actions and the size of its state space. The next two explain forward search and heuristics. Two more deal with what the planner answers: a plan, a better plan, the best plan, or no plan at all. The last two sections turn all of it into advice for the modeler and into two additions to the MODEL method: estimating the size of a model before running it, and giving numbers to the criteria Size and Guidance. The chapter uses no new PDDL: only the STRIPS of Chapter 2, on the courier world, the sliding puzzle and the towers of Hanoi.

3.1 What a model means . . . 47
A problem small enough to draw . . . 47
States . . . 48
Applicable actions and successors . . . 48
Plans . . . 49
The state space . . . 50
3.2 Grounding: from variables to objects . . . 50
Counting by hand . . . 51
What the planner says . . . 51
The whole floor, the puzzle and the towers . . . 52
3.3 Why the space explodes . 54
Counting states . . . 54
3.4 Forward search . . . 55
The loop . . . 55
Searching blind . . . 55
3.5 Heuristics: asking a simpler problem . . . 56
A number for each state . . 56
Ignoring the deletes . . . 56
The heuristic over the whole space . . . 57
When the relaxation sees little . . . 58
3.6 A plan, a better plan, the best plan . . . 59
Anytime: the first plan, then better ones . . . 59
Choosing how much to ask 60
3.7 No plan, or no plan found 60
Proved by grounding . . . 61
Proved by search . . . 61
Not found . . . 62
3.8 What this means for the modeler . . . 62
Parameters multiply . . . 62
Static preconditions prune 63
Dead ends . . . 63
Symmetry . . . 64
3.9 The MODEL method: size and guidance in numbers 65
Alternatives compared . . 66
3.10 Common mistakes . . . 67
3.11 Summary . . . 68
3.12 Further reading . . . 69
3.13 Exercises . . . 69

3.1. What a model means

A PDDL model is a text, and a planner is a program that reads it. For the two to agree, and for two planners to agree with each other, the text must have one meaning. This section states it for STRIPS. Later chapters extend it, but never change it.

A problem small enough to draw

The floor of the hospital has eight rooms, too many for a first drawing. We cut it down to three rooms in a row, with the robot in the middle one (Figure 3.1). The domain is the one of Chapter 2, unchanged; the problem is Listing 3.1.

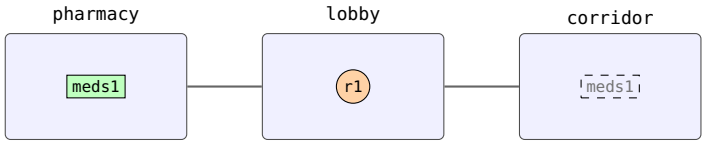


Figure 3.1.: The tiny problem of this chapter. The robot `r1` starts in the lobby with empty hands; the medicines are in the pharmacy and must end in the corridor (dashed).

```
;; 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)))
```

Listing 3.1: The tiny problem: three rooms, one robot, one item.

States

A **fact** is a predicate applied to objects, such as `(robot-at r1 lobby)`. A **state** is a set of facts: the ones that are true. Every fact that is not in the set is false; this is the closed world assumption of Chapter 2. The initial state of the tiny problem has 16 facts, the ones listed in `:init`.

Thirteen of those facts can never change, because no action has them in its effect: the three type facts, such as `(location lobby)`, and the map. They are the same in every state, so we leave them out when we write a state down. What remains is short:

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

Applicable actions and successors

An action of the domain has parameters. Replacing each parameter by an object gives a **ground action**, such as `(move r1 lobby pharmacy)`. A ground action a has three sets of facts: its preconditions $pre(a)$, the facts it makes true $add(a)$, and the facts it makes false $del(a)$. For `(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)`

A ground action a is **applicable** in a state s when all its preconditions are in s: $pre(a) \subseteq s$. Applying it gives the **successor** state: remove the deleted facts, then put in the added ones.

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

Everything else stays as it was. That is the whole meaning of an action in STRIPS: three sets and one formula.

A state is the set of true facts. An action applies when its preconditions are in the state, and it changes the state by deleting some facts and adding others.

In s_0 two ground actions are applicable: `(move r1 lobby pharmacy)` and `(move r1 lobby corridor)`. The action `(pick r1 meds1 pharmacy)` is not,

because `(robot-at r1 pharmacy)` is not in s_0. Applying the first move gives

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

and in s_1 the pick is applicable.

Plans

In STRIPS the goal is a set of facts too. A state s **satisfies** the goal G when $G \subseteq s$: the goal facts are true, and the goal says nothing about the rest. A **plan** is a sequence of ground actions $a_1, a_2, \ldots, a_n$ such that a_1 is applicable in the initial state, each next action is applicable in the state that the previous one produced, and the last state satisfies the goal. Nothing more, and nothing less: a plan may be longer than needed, and it may walk in circles.

The command `kairos validate` checks exactly this definition. We write a plan by hand in the file `courier-p01.plan`, one ground action per line, and ask:

```
$ 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
```

The word *makespan* belongs to plans with time (Chapter 10); for a sequence it is the number of steps. A sequence can fail to be a plan in two ways, and the validator tells them apart. In `courier-p01-shortcut.plan` we remove the third action, so that the robot tries to leave a room where it is not:

```
$ 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
```

In `courier-p01-halfway.plan` every action is applicable, but the robot drops the medicines in the lobby:

```
$ 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
```

The state space

Take the initial state, apply every applicable action, do the same in each new state, and go on until nothing new appears. The result is a graph: its nodes are the **reachable states**, and there is an arrow from s to s', labeled with a ground action, when that action is applicable in s and produces s'. This graph is the **state space** of the problem. A plan is a path in it from the initial state to any state that satisfies the goal.

For the tiny problem the whole graph fits in Figure 3.2. A state is decided by two things: where the robot is (three choices) and where the medicines are (in one of the three rooms, or in the hands of the robot: four choices). That makes $3 \times 4 = 12$ states.

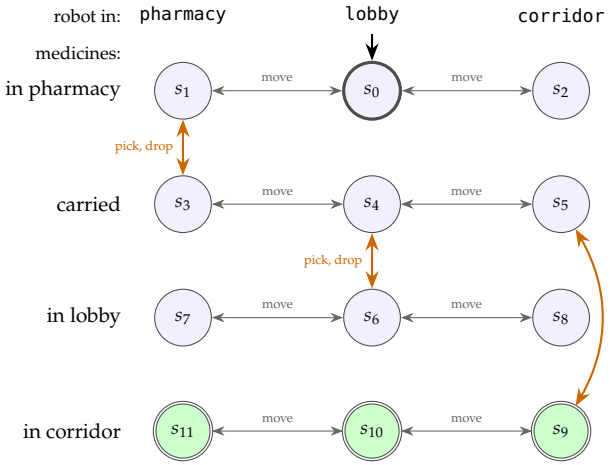


Figure 3.2.: The state space of the tiny problem: 12 states. Each double arrow stands for two ground actions, one in each direction (thick arrows: `pick` and `drop`). The initial state is s_0; the three states of the bottom row satisfy the goal. The plan written by hand is the path $s_0, s_1, s_3, s_4, s_5, s_9$.

Three things in the figure are worth a second look. First, there are several goal states: the goal says where the medicines are, not where the robot ends. Second, every arrow is double, so nothing in this problem is irreversible; it will not always be so. Third, the state space is not in the PDDL files. It is *implied* by them, and the planner has to build the part of it that it needs.

3.2. Grounding: from variables to objects

The search works on ground actions, but the domain has actions with parameters. Most planners start by producing every ground action that could ever be needed, a step called **grounding**; Kairos does so too, like most planners. Figure 3.3 shows where it sits.

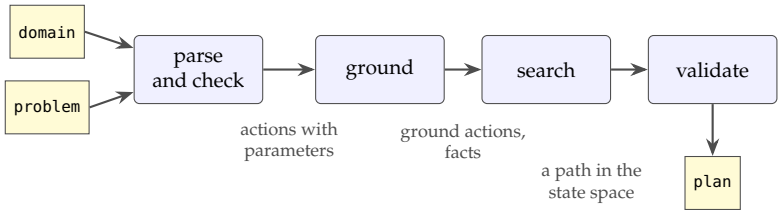


Figure 3.3.: The stages a model goes through. Grounding turns the few actions of the domain into the many ground actions of one problem; the search looks for a path among the states they create.

Counting by hand

How many ground actions does the tiny problem have? The first answer is: every way of replacing parameters by objects. The problem has 5 objects (`r1`, `meds1` and three rooms) and each action has 3 parameters, so each action has $5^3 = 125$ instances, and the three actions have 375.

Most of the 375 are nonsense: `(move meds1 r1 r1)` is one of them. The planner discards an instance as soon as one of its preconditions is a fact that can never be true. The type facts and the map decide most cases, because they never change:

- `move ?r ?from ?to` needs `(robot ?r)` and `(connected ?from ?to)`. There is 1 robot, and the map has 4 `connected` facts: $1 \times 4 = 4$ instances.
- `pick ?r ?i ?l` needs a robot, an item and a room: $1 \times 1 \times 3 = 3$ instances.
- `drop ?r ?i ?l`, the same: 3 instances.

That is $4 + 3 + 3 = 10$ ground actions, out of 375 candidates. We count the facts that can change in the same way: `robot-at` for 1 robot and 3 rooms (3), `item-at` for 1 item and 3 rooms (3), `carrying` (1) and `free` (1): 8 facts.

What the planner says

The option `-v` makes Kairos report what it does, on the standard error. In this chapter we always add `--threads 1`: with one thread the runs are reproducible, and the numbers printed here are the ones the reader will get. (The times will differ; they depend on the machine.)

```
$ 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
```

The `[ground]` line confirms the count: 8 atoms (**atom** is another name for a ground fact) and 10 actions. We will come back to the other lines along the chapter. To see the ground actions themselves, `--dump=ground` writes them to the file `ground.txt`. Two of the ten:

```
$ 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))
...
```

Two details show how the grounder thinks. The move has lost four of its five preconditions: `(robot r1)`, the two `location` facts and `(connected lobby corridor)` were checked once, during grounding, and need no more checking. A fact that no action changes is a **static** fact, and a precondition on a static fact costs nothing during the search. And `(pick r1 meds1 lobby)` is there although the medicines do not start in the lobby: they could be dropped there later, so the action may be needed.

Grounding replaces parameters by objects in every way that the static facts allow, and keeps the actions that could become applicable at some point.

The grounder decides "could be needed" with a quick, optimistic analysis: it assumes that a fact, once true, stays true. That keeps a few actions that can never be applied, as the towers of Hanoi will show in a moment, but it never throws away one that a plan needs.

Facts that exclude each other

The `[sas]` line says "3 variables". The robot is always in exactly one room, so of the three facts `(robot-at r1 pharmacy)`, `(robot-at r1 lobby)` and `(robot-at r1 corridor)` exactly one is true in any reachable state. Planners find such groups of mutually exclusive facts (*mutex groups*) by analyzing the actions, and store one variable with several values in place of several true-or-false facts: one variable for where the robot is, one for where the medicines are (three rooms, or carried), one for `(free r1)`. The file written by `--dump=sas` shows them. This representation comes from the Fast Downward planner (Helmert, 2006, 2009); we meet it again in Chapter 16, where PDDL lets the modeler write such variables directly.

The whole floor, the puzzle and the towers

The same count works for the problem of Chapter 2, which is `courier-p02` here. It has one robot, one item and the eight rooms, and its map has 7 connections, so 14 `connected` facts. Moves: $1 \times 14 = 14$. Picks: $1 \times 1 \times 8 = 8$. Drops: 8. Total: 30 ground actions, and $8 + 8 + 1 + 1 = 18$ atoms. Kairos agrees:

```
$ 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)
...
```

The other two domains of the chapter are classics. In the **sliding puzzle** (Listing 3.2) eight numbered tiles sit on a 3×3 board, and a tile next to the empty cell can slide into it. Its only action, `slide ?t ?from ?to`, needs a tile and two adjacent cells. The board has 12 pairs of neighbors, so 24 `adjacent` facts: $8 \times 24 = 192$ ground actions. The atoms are `tile-at` for 8 tiles and 9 cells (72) and `empty` for 9 cells: 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))))
```

Listing 3.2: The sliding puzzle.

In the **towers of Hanoi** (Listing 3.3) a tower of discs must go from one peg to another, moving one disc at a time and never putting a disc on a smaller one. The encoding treats pegs and discs alike, as things a disc can rest on, and lists in the static predicate `smaller` what may rest on what. Listing 3.4 is the problem with three discs.

```
;; 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))))
```

```
;; 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))
```

Listing 3.3: The towers of Hanoi.

```
18  (:goal (and
19    (on d1 d2) (on d2 d3) (on d3 peg3))))
```

Listing 3.4: Three discs, from `peg1` to `peg3`.

Count `move ?d ?from ?to` for three discs. The smallest disc, `d1`, may rest on 5 things (two discs and three pegs), so both `?from` and `?to` have 5 choices: 25 instances. For `d2` there are 4 things: 16. For `d3`, the three pegs: 9. Total: 25 + 16 + 9 = 50, and that is what Kairos reports. The count includes `(move d1 peg2 peg2)`, which can never be applied: it needs `d1` on `peg2` and `peg2` clear at the same time. The optimistic analysis of the grounder does not see the contradiction. Such actions waste a little memory and are otherwise harmless.

3.3. Why the space explodes

Ground actions grow with the problem, but gently. States do not. This section counts both as the courier gets more work.

Counting states

Take the whole floor with one robot and n items. A state is decided by the room of the robot (8 choices) and by where each item is. Either the robot carries nothing, and each item is in one of 8 rooms (8^n ways); or it carries one of the n items and the others are in rooms ($n \cdot 8^{n-1}$ ways). All of these states can be reached, since the robot can take any item anywhere. So:

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

For one item that is $8 \cdot 9 = 72$. Each new item multiplies the number by more than 8. Table 3.1 puts the formula next to what Kairos reports for the problems `courier-p02` to `courier-p05`, and for `courier-p06`, which adds a second robot to the four deliveries (for two robots the same reasoning gives $64 \cdot (8^4 + 8 \cdot 8^3 + 12 \cdot 8^2)$).

Problem	Atoms	Ground actions	Reachable states
`p02`: 1 item	18	30	72
`p03`: 2 items	27	46	640
`p04`: 3 items	36	62	5 632
`p05`: 4 items	45	78	49 152
`p06`: 4 items, 2 robots	58	156	573 440

Table 3.1.: Size of the courier problems. Atoms and ground actions are the ones Kairos reports with `-v`; the reachable states are counted with the formula of the text. Each item adds 9 atoms and 16 actions, and multiplies the states by about 9.

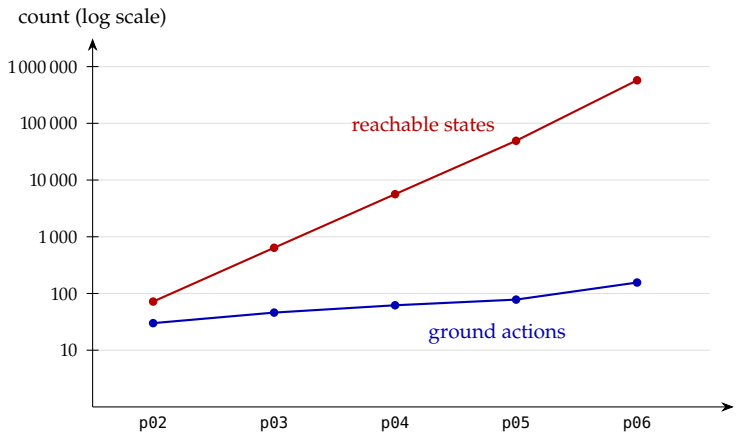


Figure 3.4.: The numbers of Table 3.1 on a logarithmic scale. The model grows by addition (the lower line is almost flat); the state space grows by multiplication.

What the modeler writes grows slowly with the problem; the state space it implies is multiplied by every new object that can be in several places.

The puzzle and the towers tell the same story. A board of the sliding puzzle is an arrangement of 8 tiles and a hole in 9 cells: 9! = 362 880 arrangements, of which exactly half, 181 440, can be reached from a given one (the other half would need two tiles to jump over each other). In the towers of Hanoi each disc is on one of three pegs and the order on a peg is forced, so n discs have 3^n states: 27 for three discs, 59 049 for ten, 531 441 for twelve. In all three domains the PDDL text has the same few lines whatever the size.

How hard is planning?

The explosion is not a defect of today's planners that better engineering will remove. Bylander proved in 1994 that deciding whether a STRIPS problem, already grounded, has a plan is PSPACE-complete (Bylander, 1994). Problems of that class are believed to be harder than the famous NP-complete ones: a plan may need a number of steps that is exponential in the size of the model (the towers of Hanoi with n discs need $2^n - 1$ moves), so even writing the answer down can take exponential time. No planner can be fast on every model. The practical consequence is the subject of this chapter: planners are fast on models that give their search something to hold on to, and the modeler decides part of that.

3.4. Forward search

A planner cannot build a graph of half a million states to find a path of 27 steps in it, at least not as its first move. It builds the graph while it explores it, starting from the initial state. This is **forward search**, the way most current planners work, and the only one we describe.[1]

1: Planners have also searched backward from the goal, in the space of partial plans, and through translations to logic. Ghallab et al. (2004) cover them all.

The loop

The search keeps two collections: the **open list**, with the states it has met and not yet explored, and the set of all the states it has met, so that it explores none twice. It repeats four steps:

1. Take a state out of the open list. If the list is empty, stop: there is no plan.
2. If the state satisfies the goal, stop: the path that led to it is a plan.
3. **Expand** the state: find the ground actions applicable in it and compute the successor of each.
4. Put in the open list the successors never met before, each with a note of the state and the action it came from.

Everything that distinguishes one search from another is in step 1: *which* state to take. Two numbers measure the work done: the states **expanded** (step 3) and, when a heuristic is used, the states **evaluated** with it. Kairos prints both with `-v`.

Searching blind

The simplest choice is to take the oldest state of the open list. This is breadth-first search: it explores all the states one action away from the

start, then all those two actions away, and so on. Follow it on Figure 3.2. It expands s_0, and finds s_1 and s_2. It expands s_1 (finding s_3) and s_2 (nothing new). Then s_3 (finding s_4), then s_4 (finding s_5 and s_6), then s_5 (finding s_9), then s_6 (finding s_7 and s_8). The next state it takes out of the list is s_9, which satisfies the goal. Seven expansions, and the plan it returns, 5 actions long, is as short as a plan can be, because every shorter path was tried before.

Breadth-first search uses no knowledge of the problem: it spreads in all directions like water. On the tiny problem that hardly matters, because there are few directions. On `courier-p05`, whose shortest plan has 28 actions, it would have to go through most of the 49 152 states before meeting a goal state. A search that knows nothing about where the goal is pays for the whole explosion.

Forward search builds the state space as it explores it. What makes a search good or bad is the order in which it picks the next state to expand.

3.5. Heuristics: asking a simpler problem

A person who looks at Figure 3.1 does not spread like water: going to the corridor first is plainly a waste, because the medicines are on the other side. The planner needs the same kind of sight, and it must get it from the model alone, since it knows nothing of hospitals.

A number for each state

A **heuristic** is a function h that gives each state an estimate of how many actions are still needed to reach the goal from it. With a heuristic, step 1 of the loop becomes: take the state of the open list with the smallest h. This is **greedy best-first search**. If the estimates were exact the search would walk straight to the goal, expanding only the states of the plan. They are not exact, and the art is in getting estimates that are cheap to compute and good enough.

Ignoring the deletes

The idea that changed planning around the year 2000 is to get the estimate by solving a simpler version of the same problem, a **relaxation**, and the simplification that works is surprisingly crude: *ignore every delete effect*. In the relaxed problem a fact, once true, stays true forever. The robot that moves from the lobby to the pharmacy is then in both rooms; an item that is picked up is in the hand and still on the shelf.

The relaxed problem is easy because nothing is ever lost: the set of true facts only grows, so there are no wrong moves and no need to search. A plan for it, a **relaxed plan**, can be built in two passes. Going forward, we collect in layers the facts that become true. From s_0 of the tiny problem (Figure 3.5):

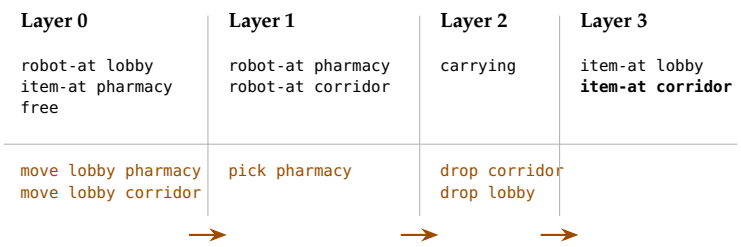


Figure 3.5.: The relaxed problem from s_0. Each layer shows the new facts (object names `r1` and `meds1` left out); below the line, in color, the actions applied in that layer, which produce the next one. Nothing is ever removed, so the robot is in three rooms from layer 1 on. The goal fact appears in layer 3.

Going backward from the goal, we pick for each fact needed one action that produces it, and then ask for the preconditions of that action. `(item-at meds1 corridor)` comes from `(drop r1 meds1 corridor)`, which needs `(carrying r1 meds1)` and `(robot-at r1 corridor)`. The first comes from the pick, which needs the robot in the pharmacy, which comes from `(move r1 lobby pharmacy)`. The second comes from `(move r1 lobby corridor)`; the robot is still in the lobby, because nothing is deleted. The relaxed plan has four actions:

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

The heuristic value is the length of the relaxed plan: $h(s_0) = 4$. The real answer is 5. The relaxation has forgotten the walk back from the pharmacy to the lobby, since a robot that never leaves a room never needs to return to it.

The delete relaxation pretends that facts never become false. Its plans are easy to find, and their length estimates how far the goal is in the real problem.

Planning as heuristic search

Estimating distances with the delete relaxation was introduced by Bonet and Geffner in the planner HSP (Bonet & Geffner, 2001). Hoffmann and Nebel's FF (Hoffmann & Nebel, 2001) extracted an explicit relaxed plan, as we just did, used its length as the heuristic and its actions as hints of what to try first; FF was the most successful automatic planner of the planning competition of 2000. The planners that followed kept the scheme and refined it: Fast Downward (Helmert, 2006) and LAMA (Richter & Westphal, 2010), which adds *landmarks*, facts that every plan must make true at some point. The heuristics that Kairos uses by default for a problem like ours are the two of LAMA: the relaxed plan of FF and a count of the landmarks still to reach.

The heuristic over the whole space

We can repeat the exercise in each of the 12 states. Figure 3.6 shows, for each state, the length of its relaxed plan and, after the bar, the true distance to the goal.

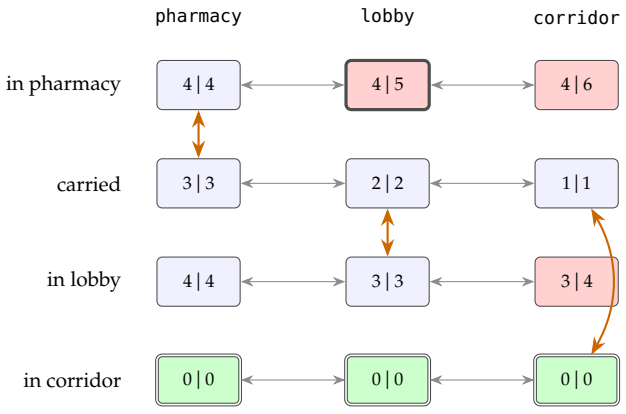


Figure 3.6.: The state space of Figure 3.2 with, in each state, the length of its relaxed plan and the true distance to the goal. They differ in three states (in red), always because the relaxation forgets that the robot has to come back. The initial state is the one with the thick border.

The estimate is exact in nine states and too low in three. More important than its accuracy is its shape: along the plan the values go 4, 4, 3, 2, 1, 0, steadily downhill, so a greedy search is led to the goal. It expands s_0 and

sees two successors, both with value 4. This is a **plateau**: the heuristic does not say which way is better, and the search has to try. If it tries s_1 first it then goes straight down: s_3, s_4, s_5 and the goal.

Kairos shows this descent with `--debug=search`, which prints a line each time the search meets a state with a lower value than any before. On the whole floor (`courier-p02`, whose shortest plan has 7 actions):

```
$ 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
...
```

The estimate of the initial state is 6 for a true distance of 7 (the forgotten step is, once more, the return from the pharmacy to the lobby), and the search reaches the goal after expanding 8 states of the 72. On `courier-p05` the first plan comes after 79 expansions, in a space of 49 152 states. That is the whole reason why planners work.

When the relaxation sees little

The delete relaxation is good at problems where the difficulty is *getting* things, and poor where the difficulty is that things *get in the way*, because getting in the way is done by deletes. The towers of Hanoi are the extreme case. Take three discs. In the relaxed problem, moving `d1` away makes `d2` clear, and `d1` never comes back to cover anything; a peg, once clear, stays clear. So the relaxed plan moves each disc once: `d1` to `peg2`, `d2` to `peg3`, `d3` to `peg3`, and it is done, because `(on d1 d2)` and `(on d2 d3)` were true at the start and nothing deletes them. The estimate is 3; the true answer is 7. Table 3.2 shows that the gap grows without limit: the estimate for n discs is n, the truth is $2^n - 1$.

Discs	States (3^n)	Initial h	First plan	Shortest plan
3	27	3	14 actions, 23 expanded	7
4	81	4	18 actions, 28 expanded	15
6	729	6	77 actions, 284 expanded	63
8	6 561	8	357 actions, 3 170 expanded	255
10	59 049	10	1 322 actions, 22 333 expanded	1 023

Table 3.2.: The towers of Hanoi with Kairos (problems `hanoi-p02` to `hanoi-p06`). The estimate of the initial state is the `ff` value printed by `--debug=search`. With so little guidance the first search expands a good part of the space: more than a third of it for ten discs.

Compare the last row with the courier: 79 expansions in a space of 49 152 states there, 22 333 in a space of 59 049 here. The two spaces have about the same size. What differs is how much the heuristic sees of each, and that depends on the problem and on how it is modeled.

3.6. A plan, a better plan, the best plan

Greedy search goes where the heuristic points and returns the first plan it meets. Nothing makes that plan short. Planning distinguishes two ambitions. **Satisficing** planning wants a plan, a reasonable one if possible, found fast. **Optimal** planning wants a plan with the guarantee that no better one exists, and pays for the guarantee. Until Chapter 9, "better" means "with fewer actions".

Anytime: the first plan, then better ones

Kairos takes a middle road, called **anytime** search: it returns the first plan as soon as it has one, and keeps searching for better ones while there is time. Each better plan is printed when found, so the last plan printed is the best. A hard board of the sliding puzzle shows it (the file `puzzle-p02.pddl`; the plans themselves are cut here):

```
$ 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)
...
```

Four plans: 95 slides, then 51, 39 and 31. With `-v` we see when each one came and how the run ended:

```
$ 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
```

Read the times. The first plan took 6 milliseconds and 440 expansions. The plan of 31 slides was found after 50 milliseconds. Then the planner worked for almost a second and a half more without printing anything, and ended with `proven optimal`: it had looked for a plan of 30 slides or fewer everywhere, and there is none. To be sure of that it had to meet

181 440 states: the whole reachable space, the number we computed in Section 3.3.

How the improving searches work is beyond this chapter; in short, each one may only return a plan cheaper than the best so far, and trusts the heuristic less than the one before, so that it explores more widely.[2] The last one trusts it so little that it ends up looking everywhere. Table 3.3 shows the same three jobs on the courier problems.

Finding a plan, finding a good plan and proving that a plan is the best are three different jobs, each far more expensive than the one before.

2: The documentation of Kairos gives the details: restarts with weighted A* of decreasing weight, as LAMA does (Richter & Westphal, 2010).

	First plan		Proof of the best plan	
Problem	Actions	Expanded	Actions	States met
`courier-p02`	7	8	7	33
`courier-p03`	13	21	13	397
`courier-p04`	19	47	19	5 054
`courier-p05`	28	79	28	48 048
`courier-p06`	31	393	27	544 039

Table 3.3.: The first plan and the proof that a plan is the shortest, with Kairos and one thread. The first plan costs a few dozen expansions; the proof makes the planner meet most of the reachable states of Table 3.1.

On `courier-p06`, the problem with two robots, the first plan (31 actions) comes after 4 milliseconds, the best one (27 actions) after 10, and the proof that 27 is the minimum after 3.3 seconds on our machine. For a robot that waits for its orders, the first two numbers matter and the third one does not.

Choosing how much to ask

Three options say when to stop. `--no-anytime` returns the first plan and ends. `--timeout S` gives the search S seconds (30 if nothing is said) and keeps the best plan found in them. `--target-cost C` stops at the first plan of cost C or less:

```
$ 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
...
```

The words in parentheses after `solved` say why the search ended: `proven optimal`, `target reached`, `timeout`, or `no improvement` (which is also what `--no-anytime` reports). Only the first one is a guarantee about the plan.

3.7. No plan, or no plan found

When the planner returns without a plan, the modeler must know which of two very different things happened: the problem *has* no plan, or the planner *found* none in the time it had. The first is a fact about the model, and usually reveals a mistake in it. The second is a fact about the search.

Proved by grounding

In `courier-p07.pddl` the medicines must go to the ICU, and the map has lost the two facts that connect `ward-b` and `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 search was needed. The optimistic analysis of grounding, the one that assumes facts never become false, could not make the goal fact true even so. If the goal cannot be reached when nothing is ever lost, it cannot be reached at all. The note reads as a chain of reasons, from the goal back to the missing fact, and it is the first thing to read when a model that should have a plan has none.

Proved by search

In `courier-p08.pddl` the door of the ICU opens only from outside: the map has `(connected ward-b icu)` but not the opposite fact. A sample waits in the ICU and must go to the lab.

```
$ 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)
```

Grounding sees nothing wrong here: with deletes ignored, the robot enters the ICU, picks the sample and is still in the corridor to carry it to the lab. Only the search finds out. It meets 8 states, the robot in each of the eight rooms. The one inside the ICU is thrown away without being expanded: from there even the relaxed problem has no plan, since no move starts in the ICU. The other seven are expanded, nothing new appears, and the open list becomes empty. An empty open list is a proof: every state that could lead somewhere was examined.

A proof by search costs the whole state space. The board `puzzle-p03.pddl` is the goal with tiles 7 and 8 swapped, one of the arrangements that cannot be reached:

```
$ 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)
```

It takes a second and a half. A heuristic cannot shorten this: it says where to look first, and here the planner has to look everywhere.

Not found

The towers of Hanoi with twelve discs have a plan, of 4 095 moves. With two seconds:

```
$ 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
```

The status is `timeout`, not `unsolvable`, and the exit code is 11, not 10. The planner claims nothing about the problem. (With the default 30 seconds it finds a first plan in about 5; the number of states in the note depends on the speed of the machine.)

"Unsolvable" is a proof that no plan exists, by grounding or by exhausting the search. "Timeout" only says that the search was stopped; the problem may well have a plan.

Table 3.4 collects the answers met in this chapter. Appendix A lists them all.

Status line	Exit code	Meaning
`solved (...)`	0	at least one plan; the parentheses say why the search ended
`unsolvable (exhausted)`	10	proved: no plan exists
`timeout (timeout)`	11	out of time before the first plan; nothing is known

Table 3.4.: What Kairos answers to a STRIPS problem. Scripts read the exit code; people read the status line and the note above it.

3.8. What this means for the modeler

The planner is general, but it is not indifferent to how a model is written. Four effects follow directly from the mechanisms of this chapter. Each is measured below on the courier world.

Parameters multiply

Each parameter of an action multiplies its instances by the number of objects that fit in it. An action with k parameters over n objects has up to n^k instances; preconditions on static facts are what cuts the number down.

A parameter that nothing constrains is not cut down at all. The domain `courier-extra-domain.pddl` is the courier domain with one change, the kind of thing that survives a copy and paste: `drop` has a fourth parameter, `?to`, that its precondition and effect never mention.

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

Listing 3.5: A parameter too many (from `courier-extra-domain.pddl`).

With the three deliveries of `courier-p04` the correct domain has 62 ground actions. This one:

```
$ 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)
...
```

The 24 drops have become $24 \times 12 = 288$, one for each of the 12 objects of the problem in the place of `?to`, and $14 + 24 + 288 = 326$. The plans are still correct, and `kairos check` finds nothing to object to. The symptoms are the jump in the `[ground]` line and an object with no business in the action, `dock`, at the end of every drop of the plan.

Static preconditions prune

A precondition on a static fact is checked during grounding and costs nothing afterwards, so it is the cheapest way to tell the planner what cannot happen. Leaving one out has two costs. In the domain of courier-anywhere-domain.pddl, the action `move` does not ask for `(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)
...
```

The moves went from 14 to $8 \times 8 = 64$ (the count includes moves from a room to itself, which nothing forbids now), the model from 30 ground actions to 80, and the robot walks through walls. This is the cheat test of Chapter 2 seen from the planner's side: the model that allows too much is both wrong and larger.

Dead ends

A **dead end** is a state from which the goal cannot be reached. A model has dead ends when some action cannot be undone. The courier world of Figure 3.2 has none; the one-way door of the ICU creates them (Figure 3.7). In `courier-p09.pddl` the robot must take the medicines into the ICU and a sample from the lab to ward A. Once inside the ICU it never leaves, so every state with the robot in the ICU and the sample not yet delivered is a dead end.

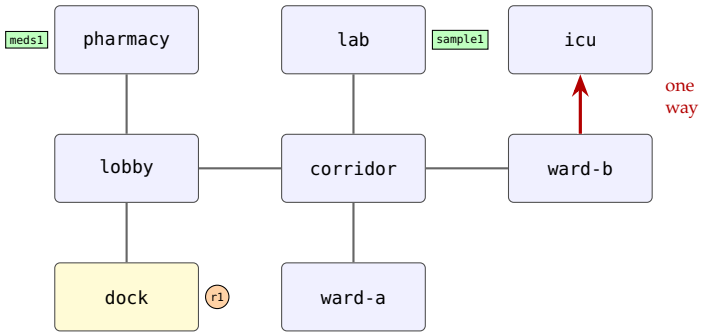


Figure 3.7.: The floor of `courier-p09`: the door of the ICU opens only from outside. `meds1` must go to the ICU and `sample1` to ward A. Entering the ICU before the sample is delivered is a dead end.

```
$ 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)
```

The plan leaves the ICU for the end, as it must: it sets the medicines down in the corridor, deals with the sample, and comes back for them. Here the planner was helped by its heuristic. From a state with the robot inside the ICU and the sample undelivered, even the relaxed problem has no solution (no move starts in the ICU, deletes or no deletes), so the state is recognized as a dead end at once and thrown away.

Not every dead end is recognized. In `courier-p08` of the previous section every state was a dead end, since the problem has no plan, but the relaxation recognized only the one inside the ICU: from the other seven it saw a plan, and the search had to exhaust the space to learn the truth. A model with many unrecognized dead ends is hard for a greedy search, which walks into them one after another. When a model is slow and its world has irreversible actions (doors that lock, fuel that is spent, things that break), this is the first suspect.

Irreversible actions create dead ends. Some are seen by the heuristic and cost nothing; the others are found only by searching all that lies behind them.

Symmetry

Two objects are **symmetric** when swapping them everywhere turns the problem into itself. In `courier-p10.pddl` four boxes of medicines, `meds1` to `meds4`, all start in the pharmacy and must all go to ward A. Nothing tells them apart, and for the hospital "the robot has delivered `meds1` and `meds3`" and "the robot has delivered `meds2` and `meds4`" are the same situation. For the planner they are different states, because they are different sets of facts. Table 3.5 shows the price as the boxes increase.

Problem	First plan	Best plan	States (proof)
1 box (`courier-p02`)	7	7	33
2 boxes	15	15	464
3 boxes	25	23	4 670
4 boxes (`courier-p10`)	33	31	43 595
1 box, 3 robots (`courier-p11`)	7	7	607

Table 3.5.: Identical objects, with Kairos and one thread: actions of the first and of the best plan, and states met to prove the best one (the problems with two and three boxes are in the `solutions` folder, for Exercise 3.8.2). Each identical box multiplies the states by about ten, although the only real question is how many boxes have arrived. Three identical robots in the place of one multiply them by eighteen.

The first plan is not much affected: the heuristic picks one box and goes on. The damage is done to everything that explores widely: the search for better plans, the proof of the best one and, above all, the proof that a problem has no plan. In STRIPS there is little the modeler can do about identical objects that are really needed. Chapter 7 shows how to count without naming ("three boxes in the pharmacy"), and Chapter 8 does it with numbers. What the modeler can do already is not to create symmetry without need: no spare robots "just in case", no objects that the goal does not require.

3.9. The MODEL method: size and guidance in numbers

Chapter 2 ran the MODEL method for the first time. Step E, Exercise the model, asked whether the plans were right; step L, Look again, compared encodings on five criteria, two of which, Size and Guidance, could only be argued in words. This chapter gives them numbers, and adds to each step a question.

- **Step E, a new question: how large is the model, and is that what I expected?** Before running, estimate the ground actions (for each action, multiply the objects that fit each parameter, then keep what the static preconditions allow) and the reachable states (multiply the choices of each thing that can change). After running, read the `[ground]` line of `kairos -v`. A number far above the estimate is a bug in the model until proved otherwise.
- **Step L, a new question: what do the numbers say?** Size is the atoms and ground actions of the `[ground]` line, and how they grow when the problem grows. Guidance is the states expanded until the first plan, the length of that plan against the best one, and the states met when the planner has to look everywhere.

Step E on the courier world: estimate, then run

We are about to run `courier-p05`: one robot, four items, eight rooms, 14 `connected` facts.
Estimate. Moves: $1 \times 14 = 14$. Picks: 1 robot, 4 items, 8 rooms: 32. Drops: 32. Ground actions: 78. Atoms: $8 + 4 \times 8 + 4 + 1 = 45$. States: $8 \cdot (8^4 + 4 \cdot 8^3) = 49\,152$. A space of this size can be exhausted in well under a second, so we may even ask for the proof of the best plan.
Run. `kairos courier-domain.pddl courier-p05.pddl -v --threads 1` reports 45 atoms and 78 actions, a first plan of 28 actions after 79 expansions, and `proven optimal` after meeting 48 048 states.
Compare. Everything matches. Had the `[ground]` line said 326

actions, as in Section 3.8, we would have looked for a stray parameter before looking at any plan.

Alternatives compared

Three encodings of the courier world, all in STRIPS, run on the same three deliveries (`meds1` to ward A, `sample1` to the lab, `meal1` to the ICU).

Encoding A, the base (`courier-domain.pddl`): `move`, `pick`, `drop`. An item that is carried is in no room.

Encoding B, loaded moves (`courier-loaded-domain.pddl`, Listing 3.6). Many first models reason like this: an item in the hands of the robot is still somewhere, so `item-at` should stay true and follow the robot. Then moving with an item changes the place of the item, and needs the item as a parameter.

```
(: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)))
```

Listing 3.6: Encoding B: the moves of `courier-loaded-domain.pddl`.

Encoding C, go anywhere (`courier-goto-domain.pddl`). The robot's route does not matter to the hospital, so one action, `go`, takes the robot to any room it can reach, and the problem lists in a static predicate, `reachable`, the 56 pairs of different rooms.

```
(: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)))
```

Running B brings a surprise that no table of sizes would have shown:

```
$ 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)
```

The plan has 18 actions where encoding A needs 19, and it ends without a drop. In encoding B `(item-at sample1 lab)` becomes true when the robot *enters* the lab with the sample in its hands, and the goal asks for nothing

else. The robot stands in the lab holding the sample, and the model calls that a delivery. The goal must say more (Exercise 3.9.2).

In C the first plan wanders (it begins with `(go r1 dock icu)`, for no reason), and the second one, the best, has 12 actions: four per delivery.

Table 3.6.: Three encodings of the courier world on the same three deliveries (problems `courier-p04`, `courier-loaded-p01` and `courier-goto-p01`; Kairos, one thread).

	A: base	B: loaded moves	C: go anywhere
Fidelity	plans can be carried out as they are	a carried item counts as delivered: the goal must also ask for free hands	correct only if any route will do; the route is not in the plan
Size	36 atoms, 62 actions; per robot, 14 moves	36 atoms, 104 actions; 14 more moves per item	36 atoms, 104 actions; 56 moves per robot, growing with the square of the rooms
Guidance	first plan: 19 actions, 47 expanded; proof: 5 054 states	first plan: 18 actions, 34 expanded; proof: 4 567 states	first plan: 15 actions, 26 expanded; best: 12; proof: 4 937 states
Clarity	three actions, one meaning for `item-at`	four actions; `item-at` means two things	three actions, but the problem must list what the map implies
Portability	plain STRIPS	plain STRIPS	plain STRIPS

Verdict. B is the one to avoid: it is larger than A, no easier to search, harder to read, and its extra fact let a wrong plan through. Its lesson is general: a parameter added to keep a redundant fact up to date is paid in ground actions and in chances to go wrong. C is a real alternative. It makes plans shorter and the search shallower, and it is the right choice when another program, a navigation system for instance, will find the route anyway. It is the wrong choice as soon as the route matters to the plan: the doors of Chapter 5 and the battery of Chapter 8 are about what happens on the way. It also grows with the square of the rooms, so on a large map it is A that stays small. We keep A for the rest of the book.

3.10. Common mistakes

Reading "timeout" as "no plan exists"

A script runs the planner, gets no plan, and reports that the mission is impossible. But the status line said

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

with exit code 11: the search was stopped, and nothing was proved. Only `unsolvable`, exit code 10, means that no plan exists. Check the exit code, not the absence of a plan, and give more time (`--timeout`) before blaming the model.

Taking the first plan for the best

With `--no-anytime`, or when only the first plan of the output is read, the hard board of the sliding puzzle gets a plan of 95 slides; the best has 31. The first plan of a greedy search is *a* plan. When quality matters, let the anytime search run and take the *last* plan

printed, and look at the status line: `solved (proven optimal)` is a guarantee, `solved (timeout)` is the best found in the time given.

Waiting for a proof nobody needs

The opposite mistake. On `courier-p06` the best plan is on the screen after 10 milliseconds, and the planner keeps running for more than three seconds to prove that it is the best; on larger problems that proof would never end, and the run stops at the timeout. If a good plan is enough, say so: `--no-anytime` for the first plan, `--timeout` or `--target-cost` for a bounded improvement.

A parameter that nothing constrains

Chapter 7 gives this mistake a name, the *idle parameter*. The plan shows actions with an object that has no business there:

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

and the `[ground]` line of `kairos -v` reports 326 actions where 62 were expected. An action has a parameter that its precondition and effect do not use, and it is instantiated once per object. `kairos check` accepts the domain, since it is valid PDDL. Remove the parameter. If the parameter is used but only in effects, look for the missing static precondition that should tie it to the others.

A missing static precondition

The plan is shorter than it has any right to be:

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

although the dock and the pharmacy are not connected, and the model has 80 ground actions where 30 were expected. A precondition that relates two parameters, here `(connected ?from ?to)`, is missing. The model is larger *and* wrong. This is what the estimate of step E is for: a count that does not match is noticed before the plans are read.

Judging a model on a tiny problem

A model that answers in a millisecond with one item is declared finished. With four items and two robots, the same model makes the planner meet 544 039 states to prove its best plan. Size and guidance show only when the problem grows. Test every model on a problem two or three times larger than the one it was written for, and watch how the numbers of `-v` grow.

3.11. Summary

A model has a precise meaning: states are sets of facts, an action is applicable when its preconditions are in the state, it produces a successor by deleting and adding facts, and a plan is a sequence of applicable actions that ends in a state with the goal facts. The planner grounds the actions, which gives a number of ground actions that grows gently with the problem, and then searches a state space that grows explosively. It survives the explosion thanks to a heuristic computed from the model itself, most often by ignoring deletes. The heuristic is not equally good

on every model, and what the planner is asked for (a plan, a good plan, the best plan, a proof that none exists) changes the cost by orders of magnitude.

Idea	What to remember
State, action, plan	$s' = (s \setminus del(a)) \cup add(a)$; `kairos validate` checks a plan against this definition
Grounding	parameters multiply, static preconditions prune; read the `[ground]` line of `kairos -v`
State space	implied by the model, never written; multiplied by each new object
Heuristic	the length of a plan that ignores deletes; blind to what gets in the way
Anytime	first plan fast, better ones later, the proof last and dearest
Unsolvable	a proof (exit code 10), by grounding or by exhausting the search; a timeout (11) proves nothing
For the modeler	no needless parameters, every static precondition, beware of dead ends and of identical objects
MODEL	step E: estimate the size, then compare; step L: Size and Guidance in numbers

3.12. Further reading

The semantics of STRIPS given here goes back to the paper that named it (Fikes & Nilsson, 1971); Ghallab et al. (2004) develop it formally, together with the kinds of search that this chapter left out. Geffner and Bonet (2013) is the shortest serious account of planning as heuristic search, and its chapters on classical planning cover the delete relaxation, landmarks and the search algorithms of this chapter with their proofs. For search in general (breadth-first, greedy best-first, A*), see Russell and Norvig (2020).

The original sources are readable. Bonet and Geffner (2001) introduce heuristics from the delete relaxation; Hoffmann and Nebel (2001) describe FF, with the relaxed plan and the actions it suggests; Helmert (2006) and Helmert (2009) explain grounding and the translation to variables with several values; Richter and Westphal (2010) add landmarks and the anytime scheme that Kairos follows. Lipovetzky and Geffner (2012) propose a different source of guidance, the novelty of a state, which Kairos also uses. The complexity result is in Bylander (1994). Planning without grounding, for problems whose ground actions do not fit in memory, is described by Corrêa et al. (2020). What Kairos itself does is in Martín Rico et al. (2026) and in Appendix A.

3.13. Exercises

The files named in the exercises are in `examples/ch03`. Run Kairos with `--threads 1` to get the numbers of the solutions.

Exercises for Section 3.1 — What a model means

Exercise 3.1.1. In the tiny problem, consider the state in which the robot is in the lobby and carries the medicines (s_4 in Figure 3.2). Write it as a set of facts, without the static ones. List the ground actions applicable in it and the successor state of each.

Exercise 3.1.2. Write by hand a plan for `courier-p01` with exactly seven actions, and check it with `kairos validate`. Is a plan that is longer than necessary still a plan?

Exercise 3.1.3. Without running anything, say what `kairos validate` will answer for this sequence on `courier-p01`, and at which step: `(move r1 lobby pharmacy)`, `(drop r1 meds1 pharmacy)`, `(pick r1 meds1 pharmacy)`. Then check.

Exercises for Section 3.2 — Grounding: from variables to objects

Exercise 3.2.1. Count by hand the atoms and the ground actions of `courier-p03` (one robot, two items, eight rooms). Check with `kairos -v`.

Exercise 3.2.2. Give a formula for the number of ground actions that Kairos reports for the towers of Hanoi with n discs, and check it for 4 and 6 discs (`hanoi-p03` and `hanoi-p04`). How many of the ground actions for 2 discs can never be applied?

Exercise 3.2.3. The *gripper* domain: a robot with two grippers, `left` and `right`, carries balls between two rooms; it moves between rooms, picks a ball with a free gripper and drops it. Write it in STRIPS without types, with a problem that takes four balls from `rooma` to `roomb`. Estimate its ground actions before running, then compare.

Exercises for Section 3.3 — Why the space explodes

Exercise 3.3.1. Derive the number of reachable states of the courier world with two robots and four items on the eight rooms, the last row of Table 3.1. How many would there be with three robots?

Exercise 3.3.2. The 15-puzzle is the sliding puzzle on a 4×4 board. How many ground actions and atoms does our encoding give it? How many reachable states, knowing that, as on the small board, half of the arrangements can be reached? At 52 bytes per state, the figure Kairos reported for the 8-puzzle, how much memory would it take to store them all?

Exercises for Section 3.4 — Forward search

Exercise 3.4.1. Draw the state space of the towers of Hanoi with two discs (`hanoi-p01`): it has 9 states. Run breadth-first search on it by hand, from the tower on `peg1` to the tower on `peg3`. In which order are the states expanded, and how many are expanded before a goal state is found?

Exercises for Section 3.5 — Heuristics: asking a simpler problem

Exercise 3.5.1. In the tiny problem, build by hand the relaxed plan from the state where the robot is in the corridor with empty hands and the medicines are in the pharmacy (s_2). Compare its length with the true distance, and explain the difference.

Exercise 3.5.2. Explain why the relaxed plan of the towers of Hanoi with n discs has n actions. Check the value for 6 discs with `--debug=search`.

Exercise 3.5.3. In the sliding puzzle, with deletes ignored, a cell that becomes empty stays empty. Describe in words what a relaxed plan looks like, and say whether you expect the heuristic to estimate too low or too high. For `puzzle-p02`, compare the initial value printed by `--debug=search` with the length of the best plan.

Exercises for Section 3.6 — A plan, a better plan, the best plan

Exercise 3.6.1. Run `hanoi-p05` (8 discs) three times: with `--no-anytime`, with `--target-cost 300`, and with no option. For each run, give the length of the plan returned and the reason why the search ended.

Exercise 3.6.2. Write the problem of the sliding puzzle for the board "1 3 6 / 5 · 2 / 4 7 8" (the dot is the empty cell) and the usual goal. How long is the first plan, how long the best one, and how many states did the planner meet to prove it?

Exercises for Section 3.7 — No plan, or no plan found

Exercise 3.7.1. Copy `courier-p02` and remove `(free r1)` from its initial state. Predict what Kairos will answer, whether it will need to search, and what its note will say. Then run it.

Exercise 3.7.2. A colleague says: "the planner ran for the 30 seconds and printed no plan, so the goal is impossible; I will tell the nurses". Which two things should be checked first, and what would you try next?

Exercises for Section 3.8 — What this means for the modeler

Exercise 3.8.1. A student writes a `handover` action for the courier world, with which a robot gives the item it carries to another robot in the same room: parameters `?from ?to ?i ?l`. With 2 robots, 4 items and 8 rooms, how many ground instances does it have at most? Which of them are useless, and can STRIPS, as we know it so far, exclude them?

Exercise 3.8.2. Using `courier-p02`, `solutions/courier-boxes2.pddl`, `solutions/courier-boxes3.pddl` and `courier-p10`, reproduce Table 3.5. Then compare the row of three boxes with `courier-p04`, which has three *different* deliveries. Which needs more states for the proof, and what do you conclude?

Exercise 3.8.3. In the world of the one-way door, the robot starts *inside* the ICU and must bring `meds1` from the pharmacy to the ICU. Predict the answer of Kairos and how many ground actions it will report. Then run `solutions/courier-trapped.pddl`.

Exercises for Section 3.9 — The MODEL method: size and guidance in numbers

Exercise 3.9.1. Apply the new question of step E to encoding C with two robots and the four deliveries of `courier-p06`: estimate atoms and ground actions, then run `solutions/courier-goto-p02.pddl` and compare. Compare also the best plan and the time of the proof with those of encoding A on `courier-p06`.

Exercise 3.9.2. Repair the use of encoding B so that a delivery is not complete while the robot still holds the item, changing only the problem. Run it and fill again column B of Table 3.6. Does the verdict change?

Exercise 3.9.3. Somebody proposes a fourth encoding, D: keep `move` as in A and add to the domain an action `deliver ?r ?i ?from ?to` that picks the item in `?from`, carries it and drops it in `?to` in one step, when `?to` is reachable from `?from`. Without writing it, judge D on the five criteria against A and C.

Part II. Classical planning: PDDL 1.2

4. Types, constants and equality

4.1 **What type predicates cost** 74
4.2 **Types** 75
What the checker now catches 77
What the planner gains . . 77
4.3 **Type hierarchies** 78
A second domain: logistics 81
4.4 **A parameter of several types: `either`** 83
The alternative: name the union 84
4.5 **Constants** 84
4.6 **Equality** 86
When the inequality is needed 88
4.7 **The MODEL method: kinds of things** 88
Alternatives compared . . 89
4.8 **Common mistakes** 91
4.9 **Summary** 92
4.10 **Further reading** 93
4.11 **Exercises** 93

The courier model of Chapter 2 works, but it trusts its author too much. Every object of a problem is just a name, and the only thing that says that `r1` is a robot and `pharmacy` a room is a fact in the initial state, `(robot r1)` or `(location pharmacy)`, that the actions check in their preconditions. Suppose that we write `(item-at pharmacy meds1)` instead of `(item-at meds1 pharmacy)`. Nothing complains. The planner looks for a plan, finds none and says that the problem has no solution, and we are left reading the whole initial state to find two swapped words. Chapter 3 showed a second cost: an action with three parameters is instantiated with every combination of three objects, and most of those combinations (a room that carries a robot from an item to another item) are nonsense that the planner has to throw away.

This chapter gives the model a vocabulary for the *kinds* of things. With **types** every object and every parameter declares what it is, a checker rejects the facts and the actions that mix kinds up, and the planner only tries the combinations that make sense. Types can be arranged in a hierarchy (a medicine is an item; a robot is a carrier), so that one action serves a whole family of objects and another serves only a part of it. With **constants** a domain can name an object that exists in every problem, such as the pharmacy. With **equality** an action can demand that two of its parameters are different objects, so that a robot is not its own witness.

We first measure what the type predicates of Chapter 2 cost, and then type the courier world and see which mistakes Kairos now catches. Hierarchies come next, with a hospital team of robots and porters and with the logistics domain, followed by `either`, a construct for a parameter that accepts several types, and its alternative. Constants and equality close the language part. The method section then adds its questions to steps O and D of the MODEL method and compares, with numbers, type predicates against types and a constant against a parameter.

4.1. What type predicates cost

In Chapter 2 the kinds of things were ordinary predicates. We called them **type predicates**: `(robot ?x)`, `(location ?x)` and `(item ?x)`. The initial state listed one such fact per object, and every action started its precondition with one of them per parameter:

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

This is correct, and it runs on any planner. It has three costs.

It is noise. A third of each precondition and ten facts of the initial state of `courier-p01` say nothing about the situation; they only repeat what the names already suggest to a human reader.

It checks nothing. A predicate such as `item-at` accepts any two objects. The file `courier-untyped-p01-swapped.pddl` is the first problem of

Chapter 2 with one slip, the one of the introduction: its initial state says `(item-at pharmacy meds1)`. We ask Kairos to check it and then to solve it:

```
$ 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)
```

The check passes: for the language, the fact is as good as any other. The planner then proves, correctly, that the problem *as written* has no plan. The note helps (nobody can pick `meds1` because it is nowhere), but the model gave the planner no means to say what is really wrong: that a room was put where an item belongs.

It multiplies candidates. `courier-p01` has 10 objects. Each of the three actions has three parameters, so there are $10^3 = 1000$ ways to instantiate each one, 3000 in total. Only 30 of them can ever be applied: 14 moves along the 7 connections in both directions, 8 picks and 8 drops. A good grounder finds those 30 without building the other 2970, because it notices that `(robot pharmacy)` never becomes true; Chapter 3 showed how. But the work of discarding is real, it grows with the cube of the number of objects, and it is done again for every problem.

Type predicates are correct but they add noise, they let wrong facts through, and they leave the planner to discover which combinations of objects make sense.

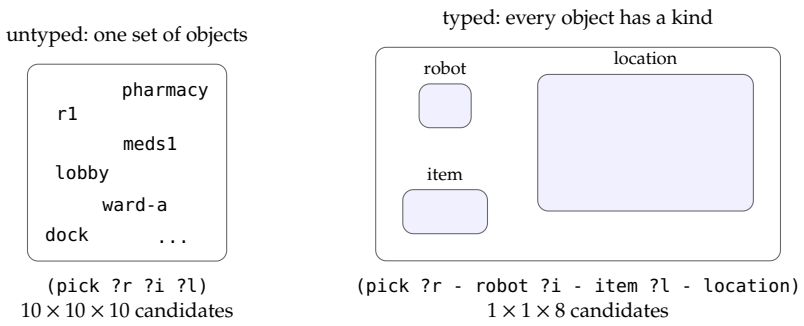


Figure 4.1.: The objects of `courier-p01` without and with types. On the left, any object can fill any parameter. On the right, each parameter draws only from the box of its type.

4.2. Types

A **type** is a name for a set of objects. A typed model says three things: which types exist, which type each object has, and which type each parameter of a predicate or of an action accepts. An object can fill a parameter only if its type fits. The requirement is `:typing`.

Listing 4.1 is the courier domain with types. It is the version that the rest of the book starts from.

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

```

```
5   (:types robot location item)
6
7   (:predicates
8     (connected ?from ?to - location)
9     (robot-at ?r - robot ?l - location)
10    (item-at ?i - item ?l - location)
11    (carrying ?r - robot ?i - item)
12    (free ?r - robot))
13
14  (:action move
15    :parameters (?r - robot ?from ?to - location)
16    :precondition (and (robot-at ?r ?from) (connected ?from ?to))
17    :effect (and (not (robot-at ?r ?from))
18                 (robot-at ?r ?to)))
19
20  (:action pick
21    :parameters (?r - robot ?i - item ?l - location)
22    :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
23    :effect (and (not (item-at ?i ?l)) (not (free ?r))
24                 (carrying ?r ?i)))
25
26  (:action drop
27    :parameters (?r - robot ?i - item ?l - location)
28    :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
29    :effect (and (not (carrying ?r ?i))
30                 (item-at ?i ?l) (free ?r))))
```

Listing 4.1: The typed courier domain (`examples/ch04/courier-domain.pddl`).

Four things changed with respect to Chapter 2.

- **The requirement.** Line 3 adds `:typing` to `:strips`.
- **The section `:types`.** Line 5 declares the three types. It comes after the requirements and before the predicates.
- **Typed variables.** A variable is followed by a hyphen and its type: `?r - robot`. The hyphen has a space on both sides, which tells it apart from the hyphen inside a name such as `robot-at`. A type applies to all the variables listed before it that have no type yet: in `(connected ?from ?to - location)` both `?from` and `?to` are locations.
- **No type predicates.** The predicates `robot`, `location` and `item` are gone, from the declarations and from every precondition. What they said is now in the parameter lists.

The problem changes in the same way: `:objects` gives each object its type, with the same hyphen notation, and the initial state loses the ten facts that only named kinds.

```
;; 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)))
```

Listing 4.2: The first courier problem, typed (`examples/ch04/courier-p01.pddl`).

```
$ 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)
```

The plan is the one of Chapter 2, step by step: types do not change what the robot can do. They change what the *modeler* can write.

What the checker now catches

We now make the same slip in the typed problem and check it:

```
$ 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
```

The message gives the file, the line and the column, says what was expected and what was found, and adds a guess that happens to be right. The mistake that cost a search and a reading of the whole file now costs a second. An object of the wrong kind, as in `courier-p01-wrongtype.pddl`, where the initial state says `(robot-at meds1 dock)`, is reported in the same way:

```
$ 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
```

The same check runs inside the domain. Every use of a predicate in a precondition or an effect is compared with its declaration, so an action that writes `(carrying ?i ?r)` with an item first and a robot second does not pass.

With types, a fact or a condition that mixes kinds is an error found by the checker, with its line, before any search.

What the planner gains

With types the candidates of an action are counted by multiplying the sizes of the types of its parameters, not the number of objects. In `courier-p01` there are 1 robot, 1 item and 8 locations:

Action	Untyped	Typed
`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

Of the 80 typed candidates, 30 survive: the 50 moves between rooms that are not connected are discarded, as before, because `connected` is static. Kairos reports the same 30 ground actions for both models (run them with `-v` and compare the `[ground]` lines). So for a planner with a careful grounder the gain is in the work of grounding, not in the search that follows: the ground problem is the same. For a planner that instantiates naively, the difference is between 3000 and 80 actions, and it widens fast: with 2 robots, 3 items and the same 8 rooms (`courier-p02`) it is 6591 against 224.

Types came with the first PDDL

PDDL was defined in 1998 by Drew McDermott and the committee of the first planning competition, held at the AIPS-98 conference McDermott, 2000; McDermott et al., 1998. That first version, known today as PDDL 1.2, already had `:typing`, `:equality` and domain constants: everything in this chapter is as old as the language. The requirement flags were there from the start too, so that a planner could say which parts of the language it supported and refuse the rest.

4.3. Type hierarchies

The hospital hires human porters to work beside the robots. A porter walks the same floor, picks and drops the same items. And the items are no longer all alike: there are medicines, samples and meals, and a meal has to be *served* to the patient, which only a person does.

With the types of the previous section we would need a type `porter` and, because a parameter has one type, a second copy of everything: `porter-at`, `move-porter`, `pick-porter`... The copies would differ in one word. What we want to say is that robots and porters are two kinds of the same thing.

A **type hierarchy** says so. A type can be declared as a **subtype** of another, its **supertype**, with the hyphen notation used for objects:

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

This reads: `robot` and `porter` are subtypes of `carrier`; `medicine`, `sample` and `meal` are subtypes of `item`; `location` stands alone. The rule that gives hierarchies their use is short: *an object of a type is also an object of every supertype of that type*. A parameter declared `?c - carrier` accepts robots and porters; a parameter declared `?p - porter` accepts porters only.

The type `object`

Every hierarchy has the same root, the predefined type `object`. A type declared without a supertype, such as `location` above, is a subtype of `object`. A variable or an object written without a type has type `object`: a parameter `?x` with no type accepts anything, and an object with no type fits only parameters of type `object`. An untyped model is therefore a typed model in which everything is an `object`; this is why Kairos wrote `?r - object` in the note of Section 4.1.

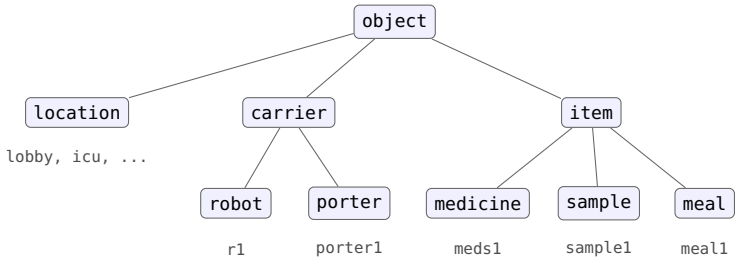


Figure 4.2.: The type hierarchy of `courier-team`, with some objects under their types. An object fits a parameter if the type of the parameter is its own type or lies above it: `meal1` fits `?m - meal`, `?i - item` and an untyped `?x`.

Listing 4.3 uses the hierarchy. The predicates and the three base actions are declared over the supertypes, so they serve every carrier and every item. Since they are no longer about robots only, `robot-at` is renamed `carrier-at` in this domain.[1] The new action `serve` is declared over two subtypes: its first parameter is a porter, its second a meal.

1: Later chapters go back to a floor with robots only and to the name `robot-at` of Listing 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))))
```

Listing 4.3: A hierarchy of carriers and items (`examples/ch04/courier-team-domain.pddl`).

In the problem every object is declared with its most specific type: `r1 - robot`, `porter1 - porter`, `meds1 - medicine`, `meal1 - meal`. The robot and the meal start in the lobby, the porter in ward B, the medicines in the pharmacy. The goal is `(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)
```

We read the plan. The same `move` and `pick` appear with `r1` and with `porter1` as first argument: one action, two kinds of carrier. The robot takes the medicines to the ICU. The porter walks to the lobby for the meal and serves it in ward A; the robot stands next to the meal at the start, but the plans in which it carries the meal to the ward have more steps in total, and serving is not something the robot can do. To be sure of that last point we write a plan in which the robot serves (`courier-team-p01-robot-serves.plan`) and validate it:

```
$ 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
```

Declare a predicate or an action over a supertype when it holds for the whole family, and over a subtype when it holds for a part of it.

Step O on the team: kinds before properties

Step O of the MODEL method asks which things exist and what the planner must know about them. With types, the first half of the question gets its own artifact, a list of kinds that is written before the state table:

Kind	Is a	Why it is a kind of its own
`location`		things are *in* it
`carrier`		it moves, picks and drops
`robot`, `porter`	`carrier`	only a porter serves
`item`		it is picked and dropped
`meal`	`item`	only a meal is served
`medicine`, `sample`	`item`	goals and later rules name them

The third column is the test for a type: *some predicate or some action treats these objects differently from the rest*. A kind that passes no such test ("red items", "items on the second shelf") is a property and belongs in the state table as a predicate. So does anything that changes while the plan runs: an object never changes its type.

A second domain: logistics

Logistics is one of the domains of the 1998 competition and has been a standard test ever since. Packages have to travel between places in several cities. Inside a city they go by truck; between cities they go by airplane, and an airplane lands only at airports. Figure 4.3 shows the small problem that we solve.

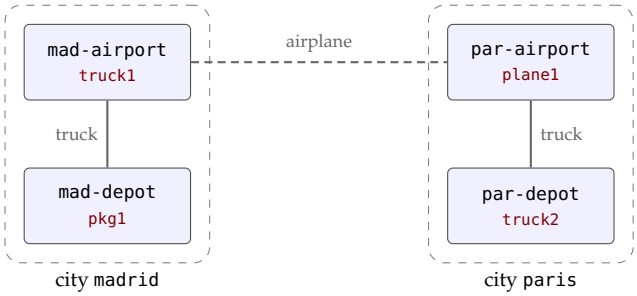


Figure 4.3.: The problem `logistics-p01`: `pkg1` must go from the depot of Madrid to the depot of Paris. Each place shows, in red, what is there at the start. Solid lines are driven by trucks, the dashed one is flown.

The kinds, with the test of the method box: trucks and airplanes are both loaded and unloaded, but they move differently, so they are two subtypes of a type `vehicle`. Airports and depots both hold packages and trucks, but only airports take airplanes, so they are two subtypes of `place`. Packages and vehicles are both *at* places: they share a supertype, `physobj`, so that one predicate `at` serves both.

```
;; 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
    truck airplane - vehicle
    package vehicle - physobj)

  (:predicates
    (in-city ?p - place ?c - city)
    (at ?x - physobj ?p - place)        ; packages and vehicles
    (in ?k - package ?v - vehicle))

  ;; One load and one unload serve trucks and airplanes.
  (:action load
    :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))))
```

Listing 4.4: Logistics with a three-level hierarchy (`examples/ch04/logistics-domain.pddl`).

A hierarchy can have any depth: `truck` is a `vehicle`, a `vehicle` is a `physobj`, a `physobj` is an `object`. The lines of `:types` can come in any order, and a type may be used as a supertype before the line that declares it (here `vehicle`, in lines 9 and 10).

Each parameter is typed as high in the hierarchy as the rule it states allows, and no higher. `load` takes any `vehicle` at any `place`; the classic untyped encodings of this domain need two actions, `load-truck` and `load-airplane`, with the same body. `drive` takes a `truck`, and `fly` an `airplane` and two parameters of type `airport`: the rule "an airplane lands only at airports" is written in the parameter list and costs no precondition.

```
$ 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)
```

The package changes vehicle twice, at the two airports, and the single `load` appears with a truck and with the airplane. A shorter plan would fly the package straight to the depot of Paris. The file `logistics-p01-fly-to-depot.plan` tries it:

```
$ 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. A parameter of several types: `either`

A public library lends books and DVDs. Magazines stay in the building: a member takes one from the shelf, reads it and puts it back. So `on-shelf` is about all three kinds, and `borrowed` about two of them. If the types are `book`, `dvd` and `magazine`, with no hierarchy, no single type fits the parameter of `borrow`.

PDDL has a construct for this case. Where a type is expected, `(either t1 t2 ...)` stands for the union of several types: the parameter accepts an object of any of them. It is part of `:typing` and needs no requirement of its own.

```
;; 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))))
```

Listing 4.5: A library with `either` (`examples/ch04/library-domain.pddl`).

In the problem `library-p01`, Ana has `book1` on loan and wants `dvd1`; Bob wants `book1` and wants to read `mag1`. A member has one loan at a time.

```
$ 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 must give the book back before she borrows the DVD (one loan at a time) and before Bob can take the book. A goal that asks for a borrowed magazine, as in `library-p02-wrong.pddl`, is rejected by the checker, which prints the union as the expected type:

```
$ 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
```

The alternative: name the union

`(either book dvd)` appears three times in Listing 4.5, and `(either book dvd magazine)` once. A union that the domain keeps repeating is a concept of the domain that has no name yet. Here the names are easy to find: what can be borrowed is *loanable*, and everything on the shelves is *media*. A hierarchy states it once:

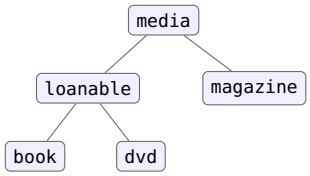
```
(:types
  member
  loanable magazine - media
  book dvd - loanable)

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

The file `library-hier-domain.pddl` is the whole domain written this way. It solves `library-p01` with the same plan and rejects `library-p02-wrong` with "must be loanable" in the message. Figure 4.4 draws both.



flat types;
the union is written at each use



the union has a name

Figure 4.4.: Two ways to say "a book or a DVD". With `either`, the union is built where it is needed. With a hierarchy, it is a type declared once.

Which one to choose? The hierarchy reads better, and a new kind of loanable thing (an audiobook) is one word in `:types`, against an edit at every `either`. It is also more portable: among the planners that this book compares, Fast Downward (and so LAMA) rejects `either` in action parameters and ENHSP does not accept it at all, while all of them read hierarchies. `either` keeps one use: unions that cut *across* a hierarchy that is already right for everything else. A type has one supertype, so a hierarchy is a tree, and two groupings that overlap (things that can be borrowed, things that need a player) do not both fit in it. Then one grouping becomes the tree and the other an `either`, or a predicate such as `(needs-player ?x)`.

Prefer a named supertype to `either`. Keep `either` for a union that crosses the hierarchy, and remember that some planners reject it.

4.5. Constants

So far a domain has never mentioned an object. Actions speak of `?r` and `?l`, and only the problem knows that there is a `pharmacy`. This is what

makes a domain reusable on any floor of any hospital. But some rules of the world are about one particular thing. In our hospital, medicines are not left lying in a room for a robot to pick up: they are in the stock of the pharmacy until the pharmacist hands them to a robot. The rule names the pharmacy. An action for it cannot have a location parameter, because the location is not a choice.

A **constant** is an object declared in the domain. It exists in every problem of that domain, and actions can use its name wherever they could use a variable. Constants are declared in a section `:constants`, after `:types` and before `:predicates`, in the notation of `:objects`. They need no requirement.

```
;; 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))))
```

Listing 4.6: The pharmacy as a constant (`examples/ch04/courier-pharmacy-domain.pddl`).

(The listing skips `move`, `pick` and `drop`, which are those of Listing 4.1.) The action `dispense` has two parameters. Its precondition `(robot-at ?r pharmacy)` mixes a variable and a constant, which is allowed in any position of any predicate, in preconditions and in effects. The new predicate `(in-stock ?m - medicine)` says that a medicine has not been handed out yet; while it is in stock it is not `item-at` any room, so `pick` cannot take it.

The problem uses the constant like any other object, in the initial state and in the goal, but does *not* declare it:

```
;; 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)
```

Listing 4.7: A problem of the domain with a constant: `pharmacy` appears in `:init` but not in `:objects` (`examples/ch04/courier-pharmacy-p01.pddl`).

The steps `(dispense r1 meds2)` and `(dispense r1 meds1)` carry no location: the plan says only what was chosen. The robot goes to the pharmacy twice, because it holds one item at a time; in between (the part cut from the output) it leaves `meds2` in the ICU and brings `sample1` from there to the lab, with the ordinary `pick`.

A constant is an object that the domain itself declares because one of its rules names it. It belongs to every problem, and no problem declares it again.

Typical constants are a distinguished place (the pharmacy, the charging dock, the depot of a delivery company) and the values of a small fixed scale that the actions name, such as `(:constants low medium high - level)`. The test is always the same: *does a rule of the domain mention this object by name?* If the answer is no, the object belongs in the problems. The lobby is in every problem of this book too, but no action treats it differently from the corridor, so it is not a constant.

4.6. Equality

The hospital now has a rule for medicines: before a medicine is delivered, a second robot must verify it. The robot that carries the medicine and a robot that acts as witness meet in some room, and the medicine becomes `verified`. A first version of the action:

```
(: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))
```

Following step E of the method, we test it with a problem that should have *no* plan: `courier-check-p01` has a single robot, and its goal is `(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)
```

There is a plan, and its fourth step shows the hole: `(verify r1 r1 meds1 pharmacy)`. The robot is its own witness. Nothing in the action said that `?r` and `?w` are different robots, and the planner does not assume it: *two parameters of an action can always be bound to the same object, if its type fits both*. Different variable names promise nothing.

To demand two different objects PDDL offers the predicate `=`, which needs the requirement `:equality`. `(= ?a ?b)` is true when both variables are bound to the same object.[2] It is not declared in `:predicates`, it is never in the initial state and no effect can change it. It is almost always used negated, as the condition $a \neq b$:

2: In PDDL two different names never denote the same object (the *unique name assumption*), so `(= r1 r2)` is false whatever the state.

```
;; 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)))
```

Listing 4.8: The double check: the requirements and the action `verify`, with the inequality as its last precondition (`examples/ch04/courier-check-domain.pddl`).

The `not` around the equality is the first condition of this book that is not a plain fact. Conditions with `not` are the subject of Chapter 5 and have their own requirement, `:negative-preconditions`, which the domain declares beside `:equality`. Many planners accept `(not (= ?a ?b))` with `:equality` alone; Kairos accepts it too, with a warning that asks for the second requirement.

Now the one-robot problem has no plan, as it should:

```
$ 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)
```

With a second robot waiting in the corridor (`courier-check-p02`, Figure 4.5) the planner finds three plans, each better than the last. The best one:

```
$ 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)
```

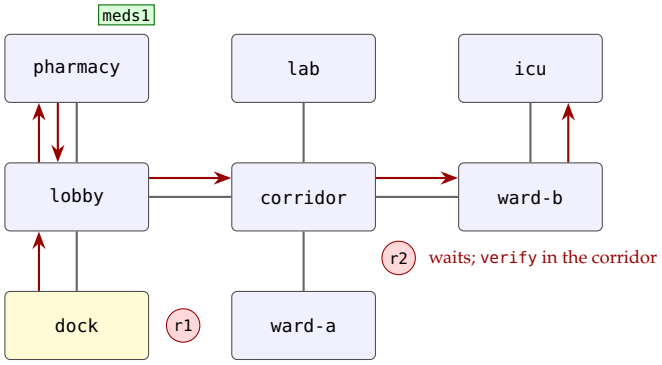


Figure 4.5.: The best plan of `courier-check-p02`. The route of `r1` passes through the corridor, where `r2` is waiting, so the verification costs one step and no detour.

The first plan that Kairos printed sent the witness to fetch the medicine and the second made `r2` walk to ward B; the third uses the fact that `r1` crosses the corridor anyway.

Step E: the cheat test for parameters of the same type

The test set of step E has one problem that tempts the planner to cheat. For an action with two parameters of the same type, or of types where one lies above the other, the cheat is always the same: *bind both to one object*. The recipe: write a problem with a single object of that type, and ask for something that should need two. If a plan comes back, read the step where the same name appears twice.

When the inequality is needed

Not every pair of parameters of the same type needs an inequality. In `(move ?r ?from ?to)` nothing stops `?from` and `?to` from being the same room, but the precondition `(connected ?from ?to)` does the work: no problem states `(connected lobby lobby)`, so that move is never applicable. The inequality is needed when *no other precondition* separates the two parameters. The `drive` of Listing 4.4 is such a case: its preconditions say that both places are in the same city, which a place is with itself, so `(drive truck1 mad-depot mad-depot madrid)` is a legal step. It is harmless (it changes nothing and no sensible plan contains it) but it is one more ground action for each truck and place. An inequality that only removes useless actions is good housekeeping; one that removes impossible plans, as in `verify`, is part of the meaning of the model.

Two parameters may be bound to the same object. Add `(not (= ?a ?b))` when the action makes no sense with one object in both roles and no other precondition prevents it.

4.7. The MODEL method: kinds of things

The chapter changes little in the model card of the courier world (step M): the stories added a porter, a pharmacist and a rule of double check, and each of them went into the Scope line as “in”. What it changes is how steps O and D are carried out. Each feature brings a question.

Step	New questions
O	**Which kinds of things are there?** A kind is a type if some predicate or action treats its objects differently from the rest and no object ever changes kind.
O	**Is there a hierarchy?** Look for kinds that share predicates or actions (a common supertype) and for actions that apply to a part of a kind (a subtype).
O	**Does an object belong to every problem?** If a rule of the domain names it, it is a constant.
D	**May two parameters be the same object?** For each pair of parameters whose types overlap: if the answer is no and no precondition already prevents it, add an inequality.

Table 4.1.: What this chapter adds to the MODEL method.

The artifacts change accordingly. The state table of step O is preceded by the list of kinds (as in the method box of Section 4.3), and its column "what it is about" now holds types: `carrying` is about a `carrier` and an `item`. The action table of step D gives a type to each parameter and gains a line "distinct" for the pairs that must differ. The test set of step E gains, for each such pair, the cheat test of the previous section, and two cheap habits: run `kairos check` on every file before solving, and write at least one wrong plan per subtype rule and see it rejected by `kairos validate`.

Alternatives compared

Type predicates or types. Both models exist: the untyped one of Chapter 2 and Listing 4.1. We run both on the same problem with two robots and three items (`courier-untyped-p02` and `courier-p02`). Both return the same plans, a first one of 21 steps and then the optimal one of 20, in the same time.

Criterion	Type predicates	Types
Fidelity	The same plans. A fact with swapped arguments is accepted and gives "unsolvable" or, worse, a wrong plan.	The same plans. Swapped or ill-kinded arguments are errors of `check`.
Size	6591 candidate actions; 124 ground actions after pruning with the static type facts.	224 candidates; the same 124 ground actions.
Guidance	The same ground problem, the same search.	The same.
Clarity	13 extra facts per problem and one extra condition per parameter. Hierarchies by hand: `(item meds1)` *and* `(medicine meds1)`.	Kinds are read in the parameter lists; a hierarchy is one line.
Portability	Plain `:strips`: every planner.	`:typing`: every planner of the comparison of Appendix B that reads classical domains.

Table 4.2.: Type predicates against types, measured on the two-robot courier problem.

Verdict. Use types. They cost nothing in portability and nothing in search, and they turn a class of silent mistakes into error messages. Type predicates keep two uses. One is a kind that *changes*: a sample that becomes "analyzed" has a property, not a new type. The other is a classification that crosses the tree, where an object would need two supertypes; a static predicate such as `(fragile ?i)` says it without disturbing the hierarchy.

A constant or a parameter. The pharmacy rule can be written without a constant. In `courier-dispensary-domain.pddl` the room is a parameter of the action, a static predicate marks the rooms where medicines are handed out, and the stock says where it is:

```
(: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)))
```

The problem declares `pharmacy` among its locations and adds `(dispensary pharmacy)` to the initial state. Both domains solve the same deliveries with plans of 21 steps.

Table 4.3.: A constant against a parameter with a static predicate, for the pharmacy rule.

Criterion	Constant `pharmacy`	Parameter and `dispensary`
Fidelity	Right while every hospital has exactly one pharmacy. A second one cannot be modeled without editing the domain.	Any number of dispensaries, each with its stock.
Size	`dispense`: $1 \times 2 = 2$ candidates. 64 ground actions in all.	`dispense`: $1 \times 2 \times 8 = 16$ candidates, 2 after pruning. 64 ground actions in all.
Guidance	The same ground problem.	The same.
Clarity	The rule is read in the domain. Plan steps are short: `(dispense r1 meds2)`.	The domain is general; the rule is split between the action and a fact of each problem. Steps repeat what is known: `(dispense r1 meds2 pharmacy)`.
Portability	Constants: every planner of the comparison that reads classical domains. A problem that declares the constant again may be rejected.	Plain typed STRIPS.

Verdict. A constant is right when the object is part of the *rules*: there is exactly one in every problem, and the domain would be a different domain without it. A parameter with a static predicate is right when the object is part of the *situation*: there may be one, several or none. The courier world of the following chapters takes the second road for the charging dock (a predicate `has-charger`, in Chapter 8), because a larger floor may have two chargers. When in doubt, choose the parameter: turning a predicate into a constant later means deleting things, while the opposite means editing every action and every problem.

4.8. Common mistakes

A type that swallows the variable before it

A type applies to *all* the untyped variables before it. Deleting "`- robot`" by accident does not leave `?r` without a type; it gives it the next one:

```
(:action pick
  :parameters (?r ?i - item ?l - location)
```

Here `?r` is an item. Kairos reports every use of `?r` that needs a 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
```

The same happens in `:objects`, in the other direction: a name written *after* the last type of the list has no type and is a plain `object`. When a message says that a variable or an object "is" of a type you did not intend, look at the hyphens of its declaration, not at the line of the message.

Types without `:typing`

A domain that uses types but declares only `:strips` is not standard PDDL. Kairos reads it and warns (`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
```

Other planners stop with a syntax error at the first hyphen, far from the cause. The warning only appears with `check` or with `-v`: one more reason to check before solving. The same holds for `:equality`.

An object declared with the supertype

An object fits the parameters of its type and of the types *above* it, never below. If the problem declares `meds1 meds2 - item`, the medicines are items that are not known to be medicines, and the predicates and actions that ask for a `medicine` reject them (`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
```

When no fact of the problem mentions the object in a subtype position there is no message: the object is simply never chosen for the subtype actions, and the symptom is a plan that is missing or longer than expected. Declare every object with its most specific type.

A constant declared again in the problem

A constant already exists in every problem. Listing it in `:objects` declares the same name twice (`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 keeps one object and goes on; a stricter planner refuses the problem. The mistake is common when a domain gains a constant late and its old problems are not updated.

Trusting that different variables are different objects

The planner binds `?r` and `?w` to the same robot whenever that helps, as in `(verify r1 r1 meds1 pharmacy)` of Section 4.6. There is no error message for this, because it is legal PDDL: the symptom is a plan that is shorter than it should be, or a plan for a problem that should have none. Only the cheat test finds it. The fix is `(not (= ?r ?w))` and the requirement `:equality`.

4.9. Summary

Types say what kind of thing each object is and what kind each parameter accepts. They remove the type predicates and their facts, they let `kairos check` and `kairos validate` reject what mixes kinds, and they reduce the candidates of each action from a power of the number of objects to a product of the sizes of its types. A hierarchy lets a rule be stated once, at the level where it holds. Constants give the domain a name for the objects that its rules mention. Equality lets an action demand that two parameters differ, which the planner never assumes.

Construct	Requirement	What it says
`(:types a b - c)`	`:typing`	`a` and `b` are types, subtypes of `c` (of `object` if no supertype is given)
`?x - t, name - t`	`:typing`	the variable accepts, or the object is, a `t` or any subtype of it
`(either t1 t2)`	`:typing`	in place of a type: an object of any of them
`(:constants n - t)`	none	an object of every problem, usable in the actions
`(= ?a ?b)`	`:equality`	both are the same object
`(not (= ?a ?b))`	`:equality`*	they are different objects

Table 4.4.: The constructs of this chapter. (*) Together with `:negative-preconditions`, for the `not`.

For the method: list the kinds before the properties; type each parameter as high in the hierarchy as its rule allows and no higher; make a constant only of what the rules name; and for every two parameters whose types overlap, ask whether they may be the same object.

4.10. Further reading

The manual of the first PDDL McDermott et al., 1998 defines `:typing`, with `either`, `:constants` and `:equality` in a few pages; the article on the 1998 competition McDermott, 2000 tells how the language was used there and describes its domains, logistics among them. The textbook of Haslum, Lipovetzky, Magazzeni and Muise Haslum et al., 2019 presents typed PDDL with more formal detail than this chapter. How a planner grounds a typed task efficiently, and why the static type facts make most candidates vanish, is explained in the description of the translator of Fast Downward Helmert, 2009. Planners that do not ground at all, and work on the typed, lifted actions directly, are a current research topic Corrêa et al., 2020.

4.11. Exercises

Exercises for Section 4.1 — What type predicates cost

Exercise 4.1.1. In the untyped problem `examples/ch02/courier-p01.pddl`, replace `(robot-at r1 dock)` by `(robot-at dock r1)`. Before running anything, predict what `kairos check` and `kairos` will answer. Then run them and explain the note that Kairos prints.

Exercise 4.1.2. An untyped courier problem has 3 robots, 5 items and 12 rooms joined by 15 connections (each in both directions). How many candidate instantiations do the three actions have in total? How many of them can be applied in some reachable state? Give both numbers again for the typed domain.

Exercises for Section 4.2 — Types

Exercise 4.2.1. The gripper domain: a robot with two grippers, `left` and `right`, carries balls between rooms. Predicates `at-robby` (the room of the robot), `at` (the room of a ball), `empty` (a gripper holds

nothing) and `holding` (a gripper holds a ball); actions `move`, `pick` and `drop`. Write it with types and solve a problem that takes three balls from `rooma` to `roomb`. How many candidates does `pick` have, and how many would it have without types?

Exercise 4.2.2. A problem of the typed courier domain declares its objects like this:

```
(:objects
  r1 - robot
  lobby corridor pharmacy lab ward-a ward-b dock - location
  meds1 - item
  icu)
```

What type does `icu` have? Which facts of the usual initial state will the checker reject, and with which message? Check your answer with Kairos.

Exercise 4.2.3. Is this parameter list legal? What type does each variable have? `(?a ?b - robot ?c ?d)`. Rewrite it so that `?a` is a robot and `?b` an item, leaving `?c` and `?d` as they are.

Exercises for Section 4.3 — Type hierarchies

Exercise 4.3.1. With the hierarchy of Listing 4.3 and the objects of `courier-team-p01`, say which of these are legal ground actions or facts (legal means that the types fit, not that the action is applicable): (a) `(pick porter1 meds1 lobby)`; (b) `(serve porter1 meds1 icu)`; (c) `(carrying meal1 porter1)`; (d) `(serve porter1 meal1 dock)`; (e) `(move meal1 lobby corridor)`.

Exercise 4.3.2. Model a zoo from this story. *Keepers and vets work in a zoo and walk between its enclosures. Animals live in enclosures. Any keeper can feed a herbivore (zebras, giraffes) in the enclosure where it lives. Big cats (lions, tigers) are fed only by a senior keeper; a senior keeper can do everything that a keeper does. Only a vet treats an animal, of any kind.* Draw the hierarchy first, write the domain, and solve a problem with a keeper, a senior keeper, a vet, a lion, a zebra and a giraffe in which every animal must be fed and the giraffe treated. Write a plan in which the plain keeper feeds the lion and see what `kairos validate` says.

Exercise 4.3.3. In Listing 4.4, what would change in the plans if the two location parameters of `fly` were declared `place` instead of `airport`? Which criterion of step L suffers? And what if `drive` were declared with `?t - vehicle`?

Exercises for Section 4.4 — A parameter of several types: `either`

Exercise 4.4.1. Rewrite the team domain of Listing 4.3, without `serve`, so that a single predicate `at` says where carriers *and* items are. Do it twice: with `either`, and with a new supertype. Solve the same problem with both. Which of the two would you hand to a colleague who uses another planner?

Exercise 4.4.2. The library buys audiobooks, which can be borrowed, and decides that DVDs and audiobooks (but not books) need a player, which the member must have before borrowing them. Explain why the hierarchy `loanable` of Section 4.4 cannot express both groupings with types alone, and propose an encoding.

Exercises for Section 4.5 — Constants

Exercise 4.5.1. Add to the typed courier domain a constant `dock` and an action `park`, with which a robot that is in the dock and carries nothing ends its shift: `(parked ?r)`. Write a problem whose goal is to deliver `meds1` to ward A and to be parked, with the robot starting in the dock. Run the cheat test: read the plan. Then repair the model using only what you know so far (no negative conditions).

Exercise 4.5.2. For each of these objects, say whether you would make it a constant of its domain, and why: (a) the table of the blocks world; (b) the city `madrid` in logistics; (c) the grippers `left` and `right` of the gripper domain; (d) the lab, in a courier world where samples are analyzed with an action `analyze`.

Exercises for Section 4.6 — Equality

Exercise 4.6.1. Two robots that are in the same room and carry one item each can exchange them. Write the action `swap` for the typed courier domain, first without an inequality. Is `(swap r1 r1 meds1 meds1 lab)` applicable when `r1` carries `meds1` in the lab? What does it change? Count the ground actions with `kairos -v` with and without `(not (= ?a ?b))`, on a problem where the swap is useful.

Exercise 4.6.2. Add inequalities to `drive` and `fly` of Listing 4.4. Kairos grounds `logistics-p01` to 24 actions without them. Predict the number with them, and say which actions disappear. Does the plan change?

Exercise 4.6.3. A student adds `(not (= ?from ?to))` to `move` in the courier domain "to be safe". Does it change the plans? The ground actions? Give a variation of the courier world in which it would matter.

Exercises for Section 4.7 — The MODEL method: kinds of things

Exercise 4.7.1. Apply step O to this story and produce the list of kinds, with the reason for each type, and the state table. *A warehouse has shelves and packing stations joined by aisles. Forklifts and hand carts move pallets; only a forklift can take a pallet from a high shelf. Some pallets hold frozen goods and may only be left at a cold station.* Which of the words of the story become types, which predicates, and is any a constant?

Exercise 4.7.2. A colleague proposes to model "robot `r2` is out of order" by declaring `r2 - broken-robot`, a new type that no action accepts. Judge the proposal with the five criteria of step L against a predicate `(working ?r)` required by every action.

5. Richer conditions

Every precondition and every goal written so far is a list of facts that must all be true. That is enough for a floor of open rooms, and it stops being enough at the first door. A robot may open a door only if the door is *not* open, and our preconditions can only ask for facts that are present. A robot may enter the ICU if it has a keycard *or* another robot with a keycard lets it in, and our preconditions can only say "and". The night shift ends when *every* item has been delivered and *no* item is left in the corridor, and our goals can only name items one by one, so a goal written for three items is wrong for four.

This chapter extends what a condition can say. Besides facts joined by `and`, a precondition or a goal can contain `not` (the fact is absent), `or` and `imply` (alternatives), `exists` (some object satisfies it) and `forall` (every object does). Each of them has its own requirement flag, and each can be avoided with plain facts and some extra work, so for each one we also ask when it is worth using. Effects stay as they are: actions still add and delete facts. Effects that depend on the situation are the subject of Chapter 6.

The courier world gains doors, which can be open, closed or locked (negative preconditions and negative goals), restricted rooms that need a keycard (`or` and `imply`), an escort for robots without one (`exists`), and goals about all the items and all the doors (`forall`). The blocks world and a puzzle of keys and chests show the same constructs outside the hospital. A short section looks at what a planner does with these conditions, the method section adds five questions to steps O and D and compares three pairs of encodings, and the chapter ends with the usual mistakes.

5.1 Conditions about absence: negative preconditions . 96
Three states with two predicates 97
Exercising the doors . . . 98
5.2 Negative goals 99
5.3 Alternatives in a condition: `or` and `imply` 100
5.4 Some object: `exists` . . . 101
A puzzle of keys and chests 102
5.5 Every object: `forall` . . 103
Blocks world: `clear` is the absence of `on` 103
One `move` for all passages 104
Quantified goals 105
5.6 What a planner does with rich conditions 106
5.7 The MODEL method: absence, any, every . . . 106
Alternatives compared . 107
5.8 Common mistakes 109
5.9 Summary 111
5.10 Further reading 111
5.11 Exercises 112

5.1. Conditions about absence: negative preconditions

Three passages of the hospital floor get a door: the pharmacy, the lab and the ICU (Figure 5.1). A robot can go through a door only when it is open, and it can open a door that is closed. So the model needs to know, for each door, whether it is open, and it needs an action whose precondition is that the door is *not* open.

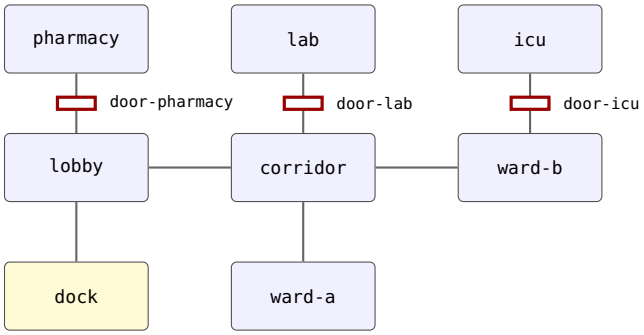


Figure 5.1.: The floor with doors. The four passages without a mark stay as `connected` facts; the three with a door become `joins` facts, one per direction.

In STRIPS a precondition can only require facts to be true. With the requirement `:negative-preconditions`, a precondition can also contain

`(not (p ...))`, a **negative precondition**, which holds in a state when the fact `(p ...)` is *not* in it. The `not` of an effect deletes a fact; the `not` of a precondition tests that a fact is missing. They are written alike and mean different things.

```
;; 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)))
```

Listing 5.1: The courier world with doors; `move`, `pick` and `drop` are those of Chapter 4 (`examples/ch05/courier-doors-domain.pddl`).

The domain uses the names that the rest of the book keeps: a type `door`, `(joins ?d ?a ?b)` for a door between two rooms, `(open ?d)` and `(locked ?d)`. A passage is now of one of two kinds. Without a door it is a `connected` fact, crossed with the old `move`. With a door it is a `joins` fact, crossed with `move-through`, which asks for the door to be open. Two negative preconditions appear in `open-door`: the door is not open and not locked.

Three states with two predicates

A door is in one of three states, and no predicate is called `closed`. "Closed" is what we call a door that is neither open nor locked. Figure 5.2 shows the states, the actions that go from one to another, and how two predicates encode them.

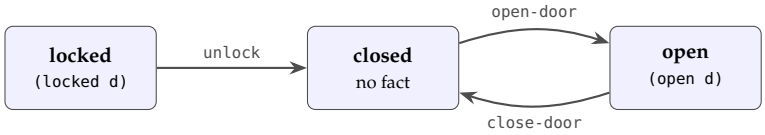


Figure 5.2.: The three states of a door and the facts that encode each. The closed door is the one about which the state says nothing.

This is the closed world assumption of Chapter 2 put to work. A state lists the facts that are true; everything else is false. So `(not (open door-lab))` needs nothing in the state to hold: it holds because `(open door-lab)` is not there. The initial state of `courier-doors-p01`, where every door is closed, simply has no fact about the state of the doors.

```
$ 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)
```

The plan of Chapter 4 had seven steps; this one has eight, because the door has to be opened once. Nothing asks the robot to close it again, so it stays open.

A negative precondition holds when the fact is absent from the state. With the closed world assumption, one predicate encodes both sides of a yes or no property.

Exercising the doors

Step E on the doors: four problems

The test set of step E, for the doors. *Trivial*: `p01` above, one closed door on the way. *Typical*: `p03`, a sample from the ICU to the lab, with the ICU door locked and a robot that has a keycard. *No plan*: `p04`, the same without the keycard. *Cheat*: `p04` again, on every earlier version of the domain: does any of them get a robot through a locked door?

The typical problem is solved in twelve steps:

```
$ 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)
```

The locked door needs two steps, `unlock` and then `open-door`, in that order: `open-door` requires `(not (locked ?d))`. The lab door is opened on the way out, as the robot passes by it. Any other moment would do; a plan is a correct order, not the order a person would choose.

Without the keycard, Kairos proves that there is no 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)
```

The cheat test earns its place here. A first version of the domain (`courier-doors-v0-domain.pddl`) had an `open-door` with a single negative precondition, `(not (open ?d))`. It solves `p01` with the same plan, and it also "solves" the problem that must have no 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)
...
```

A locked door is not open, so the robot opened it. With two predicates there are four combinations, and the fourth, a door that is open *and* locked, had become reachable. Each time a negative precondition defines a state by what is absent, check which other states have the same absence.

5.2. Negative goals

The pharmacist complains: the robot leaves the door open. The goal has to say that, at the end, the pharmacy door is not open. A goal is a condition like a precondition, and the same requirement, `:negative-preconditions`, allows `not` in it. The problem `courier-doors-p02` is `p01` with a **negative goal**:

```
(:goal (and (item-at meds1 ward-a)
            (not (open door-pharmacy))))
```

The plan gains one step, right after the robot comes out of the pharmacy:

```
$ 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)
...
```

A positive goal asks the planner to make something true, and somebody has to do it. A negative goal is satisfied by a fact that is missing, and a fact can be missing for reasons the modeler did not think of. The negative goal of `p02` is already true in the initial state; it costs a step only because the delivery forces the door open on the way. A problem whose goals are all negative often has the empty plan as its solution. We meet this again, as a cheat, in Section 5.5.

A negative goal asks that a fact is absent at the end. It is often true from the start, and it can become true in ways that were not intended.

Negative conditions are not accepted everywhere. Among the planners that this book compares, POPF and OPTIC have neither negative preconditions nor negative goals; the other classical planners of the comparison read both. Section 5.7 shows the encoding that such planners need.

5.3. Alternatives in a condition: `or` and `imply`

The ICU and the lab become **restricted** rooms: only a robot with a keycard may enter them. The rule concerns the room that a robot moves *into*, so it belongs in the precondition of `move`. In words: "the destination is not restricted, or the robot has a keycard". A list of facts joined by `and` cannot say it, because there are two different ways of being allowed in.

The requirement `:disjunctive-preconditions` adds two connectives. `(or a b ...)` holds when at least one of its parts holds. `(imply a b)` holds when `a` is false or `b` is true: "if `a`, then `b`". They can be nested freely with `and` and `not`. To keep this section about one thing, the domain `courier-access` has no doors: it is the typed domain of Chapter 4 with two static predicates, `(restricted ?l)` and `(has-keycard ?r)`, and this `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)))
```

The same rule reads more naturally as an implication, "if the room is restricted, the robot has a keycard" (`courier-access-imply-domain.pddl`):

```
                     (imply (restricted ?to)
                            (has-keycard ?r))
```

Both mean the same: `(imply a b)` is an abbreviation of `(or (not a) b)`. Choose the one that reads like the rule in the story. In `courier-access-p01` the medicines are in the pharmacy with `r1` beside them, and they must reach the ICU; only `r2`, which is in ward A, has a keycard.

```
$ 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)
```

The robot that is next to the medicines does nothing: it could carry them as far as ward B, but handing them over there would cost more steps than letting `r2` make the whole trip. If nobody has a keycard (`courier-access-p02`), there is no plan, and the note of Kairos shows the disjunction from the inside:

```
$ 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)
```

There are two lines about `move`, one for each way of satisfying the `or`. The first way needs the ICU not to be restricted, which never holds; the second needs a keycard, which nobody has. The planner has split the action in two, one per alternative. We come back to this in Section 5.6.

`(or a b)` holds when some part holds; `(imply a b)` is "if a then b". Both need `:disjunctive-preconditions`.

5.4. Some object: `exists`

The hospital relaxes the rule. A robot without a keycard may enter a restricted room if it is escorted: some robot that has a keycard is beside it, in the room it comes from. The precondition of `move` gets a third alternative, and that alternative is not about the parameters of the action. It is about *some* robot, whichever it is.

One way is to add a parameter `?e` for the escort. The other is to say "there is one" inside the condition. PDDL has a construct for that, with a requirement flag of its own, `:existential-preconditions`: `(exists (?x - t) c)` holds when the condition `c` holds for at least one object of type `t`. The variable is declared like a parameter, with its type, and it exists only inside the parentheses of the `exists`. The domain `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)))
```

Listing 5.2: The `move` of `examples/ch05/courier-escort-domain.pddl`: three ways to be allowed into a room.

The condition inside the `exists` mixes its own variable, `?e`, with a parameter of the action, `?from`. In the same situation as before, the plan changes:

```
$ 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)
```

Now `r1` carries the medicines all the way, and `r2` only walks to ward B and waits there. The step `(move r1 ward-b icu)` is legal because, at that moment, a robot with a keycard is in ward B. The plan does not say which robot: the escort is not an argument of the step. To see it one has to follow the state. That is the price of `exists`, and also its use: the action keeps its three parameters, and the planner does not have to choose an escort when any will do.

Is the inequality of Chapter 4 needed here, so that a robot does not escort itself? No. If `?e` is bound to `?r`, the condition asks that `?r` has a keycard, and then the second alternative of the `or` already lets it in.

A puzzle of keys and chests

A row of chests. Each chest opens with certain keys, and holds more keys. You start with one key and want to open the treasure chest. "You have some key that fits" is an `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))))
```

Listing 5.3: A puzzle with `exists` (`examples/ch05/chests-domain.pddl`).

In `chests-p01` (Figure 5.3) the iron key opens the oak and the pine chest. The oak chest holds the brass key, which opens the cedar chest, which holds the gold key of the treasure. The pine chest is a distraction: its silver key only fits the pine chest itself.

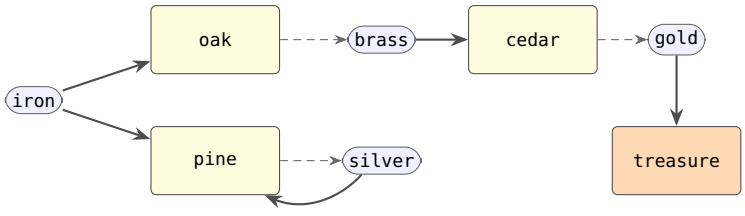


Figure 5.3.: The problem `chests-p01`. A solid arrow from a key to a chest means that the key fits; a dashed arrow from a chest to a key means that the key is inside.

```
$ kairos chests-domain.pddl chests-p01.pddl
; cost = 5
(open oak)
(take brass oak)
```

```
(open cedar)
(take gold cedar)
(open treasure)
```

Now change one rule: a key breaks when it is used. The effect of `open` must delete `(have ?k)` for the key that was used, and an effect cannot mention the variable of an `exists`, which does not exist outside the condition. The key has to become a parameter (`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))))
```

The puzzle is harder now, because opening the pine chest first wastes the iron key, and the plan names every key: `(open oak iron)`, `(open cedar brass)`, `(open treasure gold)`.

`(exists (?x - t) c)` holds when `c` holds for some object of the type. Use it when the action needs somebody and does nothing to it; when an effect must name the object, make it a parameter.

5.5. Every object: `forall`

The last connective speaks about all the objects of a type. With the requirement `:universal-preconditions`, `(forall (?x - t) c)` holds when the condition `c` holds for *every* object of type `t`. `exists` and `forall` are the two **quantifiers**, and the requirement `:quantified-preconditions` declares both at once.

The objects of a problem are a finite list, so a quantifier is an abbreviation. With three doors, Figure 5.4 shows what each one stands for.

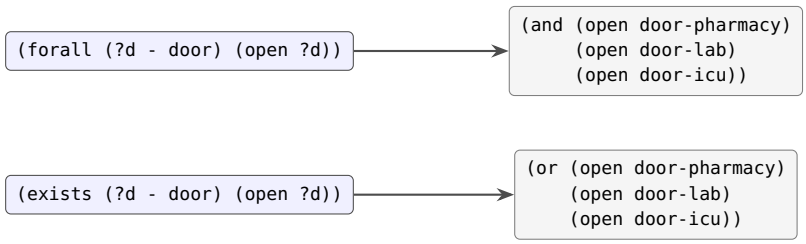


Figure 5.4.: With the objects of a problem known, `forall` is a conjunction and `exists` a disjunction with one part per object of the type. The domain is written once; the expansion is different in each problem.

Two consequences follow from the expansion. If the type has no objects, the `forall` is an empty conjunction, which is true, and the `exists` an empty disjunction, which is false. And the two are linked by negation: "no door is open" can be written `(not (exists (?d - door) (open ?d)))` or `(forall (?d - door) (not (open ?d)))`.

Blocks world: `clear` is the absence of `on`

The blocks world of Chapter 2 keeps a predicate `(clear ?x)`: nothing is on block `?x`. Every action that moves a block adds one `clear` fact and deletes another to keep it true to the towers. But `clear` says nothing new. A block is clear when no block is on it, and that can be asked directly:

```
(: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)))
```

Listing 5.4: Two actions of the blocks world without `clear` (`examples/ch05/blocks-domain.pddl`).

The effects are shorter than the classic ones, because there is one predicate less to keep up to date, and the initial state of a problem only describes the towers (Figure 5.5). The inequality in `stack` was hidden in the classic domain: there, a block in the hand is not `clear`, so it could not be stacked on itself. Here nothing is on the held block, and without `(not (= ?x ?y))` the action `(stack a a)` would be applicable.

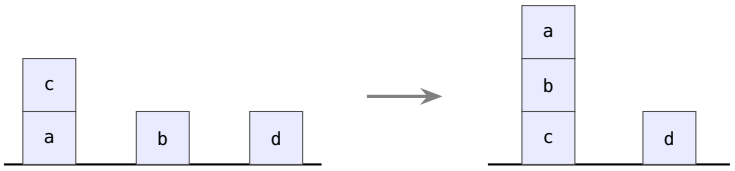


Figure 5.5.: The problem `blocks-p01`. Its initial state has five facts: `(on c a)`, three `on-table` facts and `(hand-empty)`. Which blocks are clear is read from the absence of `on` facts.

```
$ 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)
```

The classic domain (`blocks-clear-domain.pddl`, with a problem that lists three `clear` facts) returns the same six steps. Section 5.7 compares the two styles on the doors.

One `move` for all passages

The doors of Section 5.1 needed two actions for moving, one per kind of passage. A quantifier can merge them. Let every passage be a `connected` fact again, and let `joins` only say which passages have a door. Then a robot can move when *every door on that passage is open*:

```
(: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)))
```

Listing 5.5: A single `move` (`examples/ch05/courier-ward-domain.pddl`).

The pattern `(forall (?x - t) (imply a b))` is how one says "every `x` that satisfies `a` also satisfies `b`". The quantifier ranges over all the doors of the floor; the `imply` discards those that are not on this passage. For a passage with no door, every implication is true because its first part is false, and the robot moves freely. Writing `and` where `imply` belongs is the most common mistake with `forall`, and it has its box in Section 5.8.

Quantified goals

Goals take quantifiers too, with the same requirements. This is where `forall` is used most, because it makes a goal independent of the objects of the problem. The night shift of `courier-ward-p01`:

```
(:goal (and (forall (?m - medicine) (item-at ?m icu))
            (forall (?d - door) (not (open ?d)))))
```

Every medicine in the ICU, every door closed. The type of the variable selects the objects: the problem also has a sample, and the first `forall` does not touch it. The best plan has 22 steps; its end:

```
$ 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)
```

The robot closes the pharmacy door after its second and last visit, and closes the ICU door from the inside. Nothing in the goal says that the robot should come out.

The other goal of the introduction, "no item left in the corridor", is a `forall` over a negative condition. We give it the cheat test. In `courier-ward-p02` a meal lies in the corridor, the robot is there too, and the goal is `(forall (?i - item) (not (item-at ?i corridor)))`.

```
$ kairos courier-ward-domain.pddl courier-ward-p02.pddl
; cost = 1
(pick r1 meal1 corridor)
```

The robot picks the meal up and stops. An item in the gripper is not `item-at` any room, so the corridor is "empty". The goal said where the items should not be and forgot to say where they may be. In `courier-ward-p03` a second `forall` is added, `(forall (?r - robot) (free ?r))`, and the plan becomes `pick`, `move`, `drop`: the meal ends in ward B.

`(forall (?x - t) c)` holds when c holds for every object of the type, and for none if there is none. A goal about absence must also say where things may end up.

From STRIPS to ADL

The conditions of this chapter and the effects of the next one entered planning through ADL, the Action Description Language that Edwin Pednault proposed in 1989 Pednault, 1989 as a middle ground between the plain facts of STRIPS Fikes and Nilsson, 1971 and full logic. When PDDL was defined in 1998 McDermott et al., 1998, it took those constructs and gave each one a requirement flag, plus a flag `:adl` that stands for all of them together (Chapter 6).

5.6. What a planner does with rich conditions

Search algorithms and heuristics are simplest when every ground action has a plain list of facts as its precondition. Most planners therefore rewrite rich conditions before they search, and the rewriting is done on the ground problem, where the objects are known Helmert, 2009. Knowing its outline helps to predict what a condition costs.

- **Quantifiers are expanded** over the objects, as in Figure 5.4. The size of the condition grows with the number of objects of the type, and a quantifier inside another multiplies both.
- **Static parts are evaluated.** In the `move` of Listing 5.5, `(joins ?d ?from ?to)` never changes, so each implication of the expansion either disappears (no such door there) or shrinks to `(open d)`. The ground action from the lobby to the pharmacy ends with one precondition about doors, the one a person would have written.
- **Disjunctions split actions.** An action whose precondition is `(or a b)` becomes two actions with the same effects, one that requires `a` and one that requires `b`. An `exists` is a disjunction, so it also splits, into one action per object.
- **Negation** of a fact is handled directly by many planners; others add a second fact that is kept opposite to the first, the trick that Section 5.7 does by hand.

The split can be watched with `kairos -v`. On `courier-access-p01` the domain with `or` has 64 ground actions. A second file, `courier-access-split-domain.pddl`, writes the two cases by hand as two actions that exclude each other (`move` for rooms that are not restricted, `move-with-keycard` for restricted rooms and robots with a keycard) and has 52. The 12 extra actions are moves of `r2`, the robot with the keycard, into rooms that are not restricted: both parts of the `or` hold, so both copies of the action are kept, and they do the same thing. The searches are identical (the same 352 states), because duplicated actions lead to the same successor. The escort tells the same story: 98 ground actions with `exists`, 74 when the escort is a parameter of a separate action (`courier-escort-param-domain.pddl`).

Planners expand quantifiers over the objects and split disjunctions into several actions. A rich condition is as cheap as what it expands to.

These numbers are small. They stop being small when alternatives multiply: a conjunction of ten conditions with two alternatives each, all about facts that change, expands into 2^{10} combinations in a planner that splits them all. The warning sign is a `forall` whose body contains an `or`, an `imply` or an `exists` over facts that are not static.

5.7. The MODEL method: absence, any, every

The model card of the courier world changes in its Scope line: doors and access rules are now “in”; who holds the door for whom, and how long a door takes to open, stay “out”. Steps O and D each gain questions, which are asked of every rule of the story as it is turned into a condition.

Step	New questions
O	**Is the property yes or no, or does it have several values?** For yes or no, store one side and let the closed world assumption give the other; choose the side that is true less often, so that initial states are short. For three values or more, list the combinations of the predicates and check that the meaningless ones cannot be reached.
D	**Is the condition about absence?** "Not open", "nobody inside": `not`.
D	**Is it about any object?** "Some robot with a keycard": `exists` if the action does nothing to that object, a parameter if it does or if the plan should name it.
D	**Is it about every object?** "All doors closed": `forall`, with `imply` to select which objects the rule is about.
D	**Are there several ways to satisfy it?** `or`, or one action per way.

Table 5.1.: What this chapter adds to the MODEL method.

Step E gains two cheat tests, both met above. For every state defined by absence, look for another state with the same absence (the locked door that was "not open"). For every goal about absence or about "every", ask how it could be true for a silly reason: the item in the gripper that is in no room, the type that has no objects.

Alternatives compared

A negative precondition or a complementary predicate. The doors can be modeled without `not`. The domain `courier-closed` has three predicates, `open`, `closed` and `locked`, and each action deletes one and adds another, so that exactly one of them holds for each door:

```
(: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` is a **complementary predicate**: it is true exactly when another condition is false, and the effects keep it that way. The initial state must now say `(closed door-lab)` for every closed door. On the problems `p01` and `p02` both domains return the same plans, after visiting the same number of states.

Criterion	Negative preconditions	Complementary predicate
Fidelity	The same plans. A meaningless combination (open and locked) can be written and, with a careless action, reached.	The same plans. A door about which the initial state says nothing is in no state at all, and no action applies to it.
Size	21 facts and 42 ground actions on `p01`.	24 facts, 42 ground actions.
Guidance	102 states visited.	102 states visited.
Clarity	One predicate per property; short effects and initial states. Three states are read from two predicates.	One predicate per state, each with a name. Every action maintains the group; every problem lists every door.
Portability	Not accepted by POPF and OPTIC.	Plain typed STRIPS: every planner.

Table 5.2.: Negative preconditions against a complementary predicate, for the doors.

Verdict. For a yes or no property, use the negative precondition: it cannot go out of step, because there is nothing to keep in step. Use complementary predicates when the model must run on a planner without negative preconditions (the temporal planners POPF and OPTIC are the usual case), and consider them when a property has three or more values that deserve names. The cost of forgetting differs. With negation, a forgotten fact means a closed door, which is a legal state. With complementary predicates it means a door that is nothing: in `courier-closed-p01-forgot` the pharmacy door has no fact, and Kairos answers "unsolvable" for a problem whose author sees nothing wrong.

`or` or two actions. Section 5.6 already ran both encodings of the restricted rooms, the domain with `or` and the split one.

Criterion	One action with `or`	Two actions
Fidelity	The same plans.	The same plans, if the cases cover everything. A case that is forgotten is a lost solution.
Size	64 ground actions (12 duplicates).	52 ground actions.
Guidance	352 states visited.	352 states visited.
Clarity	The rule is in one place. Steps are all `move`.	The rule is spread over two actions that must be changed together. Steps say which case was used: `move-with-keycard`.
Portability	Not accepted by POPF and OPTIC.	Needs only what each case needs (here, a negative precondition).

Table 5.3.: A disjunction against one action per alternative, for the restricted rooms.

Verdict. Use `or` (or `imply`) when the alternatives are a detail of one rule and the effects are the same. Write two actions when the planner at hand has no disjunction, when the alternatives will soon differ in their effects or costs, or when the plan should tell them apart. With three independent rules of two alternatives each, the split needs eight actions; that is the point where the disjunction wins on Clarity alone.

A `forall` goal or a list. The goal of `courier-ward-p01` can be written fact by fact, as in `courier-ward-p01-listed`: two `item-at` facts and three negative ones about the doors.

Criterion	`forall` goal	Listed goal
Fidelity	Follows the objects: a third medicine is covered without touching the goal. True of a type with no objects.	Says exactly what it lists. A medicine added to `:objects` and not to the goal is silently left out.
Size	46 facts after grounding.	45 facts.
Guidance	23679 states visited to prove the plan of 22 steps optimal.	23679 states.
Clarity	States the intention ("every medicine").	States the instance. Long with many objects.
Portability	Quantified conditions are not accepted by LPG-td and Metric-FF.	Negative goals aside, any planner.

Table 5.4.: A quantified goal against the same goal listed, on `courier-ward-p01`.

Verdict. They are the same problem for the planner. Use `forall` when the goal is a rule ("everything delivered", "nothing left behind") and when problems are written by hand and edited often. Use the list when problems are produced by a program, which does not forget an object, or when the target planner has no quantifiers.

5.8. Common mistakes

`and` under a `forall` where `imply` was meant

"Every door between the two rooms is open" is not

```
(forall (?d - door)
  (and (joins ?d ?from ?to) (open ?d)))
```

That asks every door of the building to join these two rooms *and* be open. With three doors it is never true, no robot ever moves, and the symptom is far from the cause (`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)
```

Nothing in the answer points at the quantifier. What does is the trivial problem of step E: if a robot cannot make a single move between two open rooms, look at `move`. The rule of thumb: under `forall`, select with `imply`; under `exists`, select with `and`.

A `forall` that is true of nothing

In `courier-ward-p04-vacuous.pddl` the medicines were declared `meds1 meds2 - item`, and the goal is `(forall (?m - medicine) (item-at ?m icu))`. There is no object of type `medicine`, so the goal holds in the initial state:

```
$ kairos courier-ward-domain.pddl courier-ward-p04-vacuous.pddl
; cost = 0
```

An empty plan with cost 0 is a valid answer, and `check` has nothing to object to. When a quantified goal gives a plan that is too short, count the objects of its type.

A quantified variable without a type

`(forall (?m) (item-at ?m icu))` ranges over *every* object: it also asks the robot, the rooms and the doors to be items lying in the ICU. Kairos accepts a variable of type `object` where an `item` is expected (the condition is simply false for objects of other types), so the check passes and the problem has no 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)
```

Under `exists` the same slip is harmless and under a negative `forall` it is invisible, which is how it survives until the day it matters. Always type the variable of a quantifier.

Requirements forgotten

Each construct of this chapter has its flag. `courier-escort-noreq-domain.pddl` declares only `:strips` and `: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 plans anyway; other planners refuse the file or, worse, ignore the part they did not expect. The warnings are also the quickest way to learn what a domain really uses, and so on which planners it can run.

`not` in the initial state

The initial state lists what is true. Writing `(not (open door-pharmacy))` there adds nothing, since the door was already not open, and Kairos says so: "negative initial facts have no effect". The danger is the belief behind it, that a fact is unknown until stated. Nothing is unknown in these models: what is not listed is false.

Complementary predicates out of step

Chapter 7 gives this mistake a name, *out-of-step twins*. A domain with `open` and `closed` depends on every action and every initial state keeping exactly one of them true. One effect forgotten (`close-door` that adds `closed` and does not delete `open`) gives a door that is both, through which robots pass and which can be "opened" again. No message reports it; the plans just become strange. If a domain has such a pair, test each action alone, and read the state after it.

5.9. Summary

A condition, in a precondition or in a goal, is no longer only a list of facts. It can require that a fact is absent, offer alternatives, and speak of some or of all the objects of a type. All of it is evaluated on the current state under the closed world assumption, and all of it can be expanded, once the objects are known, into the facts, `and`, `or` and `not` that it abbreviates.

Construct	Holds when	Requirement
`(not c)`	`c` does not hold	`:negative-preconditions`
`(or c1 c2 ...)`	some part holds	`:disjunctive-preconditions`
`(imply c1 c2)`	`c1` does not hold or `c2` holds	`:disjunctive-preconditions`
`(exists (?x - t) c)`	`c` holds for some object of type `t`	`:existential-preconditions`
`(forall (?x - t) c)`	`c` holds for every object of type `t`	`:universal-preconditions`

Table 5.5.: The constructs of this chapter. `:quantified-preconditions` declares the last two together. The same flags cover goals.

For the method: ask of each rule whether it is about absence, about any object or about every object; prefer a negative precondition to a second predicate that must be kept opposite; let an object be a parameter when an effect touches it or the plan should name it, and an `exists` otherwise; and distrust every goal that can be true because something is missing.

5.10. Further reading

Pednault's paper on ADL Pednault, 1989 is the origin of these conditions in planning. The PDDL manual McDermott et al., 1998 defines their syntax and their requirement flags. The textbook of Haslum and colleagues Haslum et al., 2019 treats quantified and disjunctive conditions more formally. How a planner removes them (quantifiers expanded, disjunctions split, negations pushed down to the facts) is described step

by step for the translator of Fast Downward Helmert, 2009. The planning textbook of Ghallab, Nau and Traverso Ghallab et al., 2004 discusses these extensions of the classical representation and what they cost a planner.

5.11. Exercises

Exercises for Section 5.1 — Conditions about absence: negative preconditions

Exercise 5.1.1. In `courier-doors-domain.pddl`, a door is described by two predicates, so four combinations exist. Name the state that each one encodes. Which combination is meaningless? Argue, action by action, that it cannot be reached from an initial state that does not contain it.

Exercise 5.1.2. A robot needs a free gripper to open or to close a door. Add that to the domain and solve `courier-doors-p02`, where the pharmacy door must end closed. Before running, predict how the robot will manage with the medicines in its gripper.

Exercise 5.1.3. Add an action `lock`: a robot with a keycard locks a door that is closed. Write a problem in which the robot takes the medicines out of the pharmacy and leaves it locked. Which negative preconditions does `lock` need, and what happens to Exercise 5.1.1 if one is forgotten?

Exercises for Section 5.2 — Negative goals

Exercise 5.2.1. Change the goal of `courier-doors-p01` so that the robot delivers the medicines to ward A and does not stay there. How many steps does the plan gain, and where does the robot end? Is that what the head nurse meant?

Exercise 5.2.2. A student writes `(not (open door-pharmacy))` in the `:init` of a problem "to make sure the door starts closed". What does `kairos check` say? Does the plan change?

Exercises for Section 5.3 — Alternatives in a condition: `or` and `imply`

Exercise 5.3.1. Solution 4.4.2 needed two `borrow` actions in the library, because a precondition could not say "if the thing needs a player, the member has one". Write the library with a single `borrow` and an `imply`, with a static predicate `(needs-player ?x)`, and solve a problem in which Ana wants an audiobook and has no player.

Exercise 5.3.2. Show with a truth table that `(imply a b)`, `(or (not a) b)` and `(not (and a (not b)))` hold in the same states. Then write the rule of the restricted rooms in the third form and check that the plans do not change.

Exercises for Section 5.4 — Some object: `exists`

Exercise 5.4.1. New rule: no robot enters a restricted room while it carries a meal, keycard or not. Add it to `courier-access` and test it with a problem that has a plan and one that must have none.

Exercise 5.4.2. In the escort domain, the escort must be in the room that the robot leaves. Rewrite the condition for a hospital where the escort waits *inside* the restricted room and lets the robot in. Does the plan of `courier-escort-p01` still work?

Exercise 5.4.3. In `chests-once-p01`, what happens if the first step is `(open pine iron)`? Why does the version with `exists` have no such trap? Say in one sentence when a variable must be a parameter.

Exercises for Section 5.5 — Every object: `forall`

Exercise 5.5.1. The floor closes for the night with an action `close-floor`, applicable when every robot is in the dock and every door is closed; after it, no robot moves. Write it over `courier-ward-domain.pddl`, with `dock` as a constant, and solve a problem with two robots and one delivery whose goal includes `(floor-closed)`.

Exercise 5.5.2. On a floor with no medicines, what is the answer of a planner to the goal `(forall (?m - medicine) (item-at ?m icu))`? And to `(exists (?m - medicine) (item-at ?m icu))`? Check the second with Kairos.

Exercise 5.5.3. In the blocks world without `clear`, write the goal "every block is on the table" and solve it for a tower of three blocks and a fourth block apart. Then write the condition "`?x` is clear" with `forall` instead of `exists`.

Exercise 5.5.4. Write the negation of `(forall (?d - door) (imply (joins ?d ?from ?to) (open ?d)))` without a `not` in front of a quantifier. Say it in words.

Exercises for Section 5.6 — What a planner does with rich conditions

Exercise 5.6.1. An action has the precondition `(and (or a b) (or c d) (or e f))`, where the six facts all change during a plan. Into how many plain actions does a planner split it? Rewrite the condition of Listing 5.2 as the list of alternatives that the planner obtains, with two robots.

Exercises for Section 5.7 — The MODEL method: absence, any, every

Exercise 5.7.1. Apply step L to the two blocks worlds of this chapter, `blocks-domain.pddl` and `blocks-clear-domain.pddl`. Run both on their first problem with `kairos -v` and fill in a table with the five criteria. Which would you choose for a course, and which for a temporal planner?

Exercise 5.7.2. Model a coffee machine from this story, with steps O and D written out. *The machine makes a drink into the cup under its nozzle, if the cup is empty and there are water and beans. Only one cup fits under the nozzle. A cappuccino also needs milk in the machine; an espresso does not. After each drink the machine must be cleaned before the next one, and it can be cleaned only when no cup is under the nozzle.* Solve a problem that asks for an espresso and a cappuccino, starting without milk, and ending with the machine clean.

6. Richer effects

The previous chapter made the conditions of an action as rich as we wanted: a robot may enter a room if the door is open, or if it has a keycard, or if no item was left in the corridor. The effects stayed as they were in STRIPS: a fixed list of facts to add and facts to delete, the same list every time the action is applied. Now the hospital buys a cart. The courier loads items on it and pushes it from room to room, and whatever is on the cart travels with it. How many facts does one push change? It depends: two if the cart is empty, four with one box on it, twelve with five. A fixed list cannot say that. Nor can it say what happens at the door of the laboratory, where an airlock disinfects everything that enters: a sealed sample comes through intact, an open one is ruined. The move is the same; what it changes is not.

This chapter gives effects the same freedom that conditions already have. A **conditional effect**, written with `when`, happens only if a condition holds in the state where the action is applied. A **universal effect**, written with `forall`, happens for every object of a type. Together they say "every item that is on the cart changes room". With them the language reaches the level known as ADL, which is all that PDDL 1.2 offers for classical planning.

We start with the cart and see why the tools of the previous chapters fall short. Then come the two constructs, first each one alone on a domain of lamps, then both together on three models: the Lights Out puzzle, the briefcase that gave these effects their first famous example, and the cart. The sterile laboratory shows the other face of a conditional effect: something that the planner must take care *not* to trigger. After the requirement `:adl` and a look at what planners do with these effects, the chapter adds two questions to step D of the MODEL method and compares three ways of encoding the cart.

6.1 Effects that depend on the situation 114
6.2 Conditional effects: `when` 115
What a conditional effect means 116
Running it 117
A condition in the effect or in the precondition 117
6.3 Universal effects: `forall` 118
6.4 Both together: for every object such that 119
Lights Out 119
The briefcase 120
The cart 121
6.5 Effects to avoid: the sterile laboratory 124
Exercising the model . . 125
One action per case . . . 126
6.6 The requirement `:adl` . 127
6.7 What planners do with conditional effects 127
6.8 The MODEL method: effects that depend on the situation 129
Alternatives compared . 129
A fourth way: do not store what can be deduced . . 131
6.9 Common mistakes 132
6.10 Summary 134
6.11 Further reading 134
6.12 Exercises 135

6.1. Effects that depend on the situation

The cart of the hospital is a trolley with shelves. A robot that carries an item can put it on the cart; a robot with its hand free can take the handle and push the cart to a connected room. Figure 6.1 shows the delivery that we want to plan: two boxes of medicines that wait in the pharmacy, next to the cart, are needed in the ICU, four rooms away.

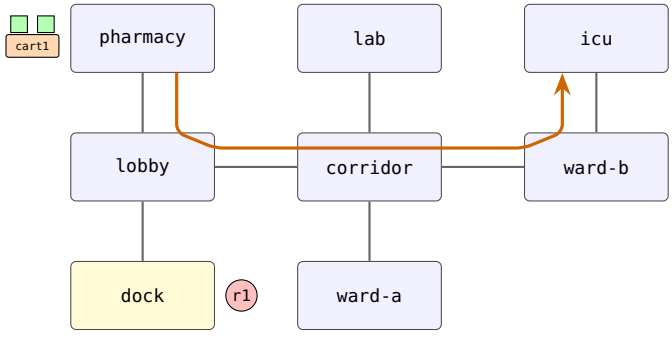


Figure 6.1.: The delivery with the cart. The two boxes (green) are on `cart1`, drawn beside the pharmacy where it stands; the orange arrow is the way of the cart to the ICU. Each of the four pushes changes the position of the robot, of the cart and of both boxes.

Following the shared vocabulary of the courier world, the model gains a type `cart` and three predicates: `(cart-at ?c ?l)`, `(on-cart ?i ?c)` and `(pushing ?r ?c)`. We decide that an item on the cart is still *somewhere*: `(item-at meds1 pharmacy)` stays true while `meds1` is on a cart that is in the pharmacy, so that goals and preconditions keep asking where an item is in the way they always did. The price is that the push must keep those facts right.

Let us try to write `push-cart` with what we know. The robot and the cart are easy: two deletes and two adds, as in `move`. The items are the trouble. An effect can only name the parameters of the action, so each item that moves needs a parameter, and an action has a fixed number of parameters. We are forced to write one action for each number of items on the cart. Listing 6.1 shows the one for two items, from a domain with four of them (`push-cart-0` to `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)))
```

Listing 6.1: One action per case: pushing a cart that holds exactly two items (from `courier-cart-cases-domain.pddl`).

The action must know that `?i` and `?j` are *all* that is on the cart: if a third item were there, it would be left behind. That is the job of the `forall` of lines 98 to 100, a universal precondition of the previous chapter. The model works. It also has three defects that grow with the problem:

- **It has a limit that nobody asked for.** With four actions the cart moves with up to three items. Put a fourth one on it and no push applies: the cart is stuck, and nothing in the hospital explains why.
- **It is large.** `push-cart-2` has two item parameters and `push-cart-3` has three. With n items and 14 directed connections in the map, the version for k items has $14\,n^k$ candidate ground actions. Every pair is also counted twice, as `(meds1 meds2)` and as `(meds2 meds1)`: two actions that do exactly the same.
- **It repeats itself.** The four actions say the same thing four times. A change to the rule (a cart that cannot enter the ICU, say) must be made in four places.

A STRIPS effect is a fixed list. When the number of facts that change, or which ones change, depends on the state, one action per case is the only way out, and the cases multiply.

What we want to write is one sentence: *the robot and the cart change room, and so does every item that is on the cart*. That sentence has two ingredients that effects do not have yet. "Every item" is a quantifier, and "that is on the cart" is a condition evaluated on the state, inside the effect. The next two sections introduce them one at a time, on a domain small enough to see every state.

6.2. Conditional effects: `when`

A corridor has lamps, each with one push button. Pressing the button changes the lamp: if it was off it goes on, and if it was on it goes off. In

STRIPS this is two actions, `switch-on` with precondition `(not (lit ?l))` and `switch-off` with precondition `(lit ?l)`. But there is only one button, and only one thing that an agent can do with it. Listing 6.2 models the button as it is: one action, `toggle`, whose effect depends on the state. (The second action of the file belongs to the next section.)

```
;; 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)))))
```

Listing 6.2: The lamps: `toggle` has two conditional effects, `all-off` a universal one (`lights-domain.pddl`).

A **conditional effect** has the form

```
(when <condition> <effect>)
```

and reads "if the condition holds, the effect happens too". The condition is a formula of the same kind as a precondition: it may use `and`, `or`, `not`, equality and quantifiers, provided the requirements of the previous chapters are declared. The effect is a literal or an `and` of literals. Conditional effects need the requirement `:conditional-effects`. The action `toggle` also declares `:negative-preconditions`, because the condition of its second `when` is a negation.

Notice what `toggle` does not have: a precondition. An action without `:precondition` is applicable in every state. The button can always be pressed; the state only decides what pressing it does.

What a conditional effect means

The meaning of an action with conditional effects is given by three rules. They are short, and most mistakes with these effects come from forgetting one of them.

1. **The conditions do not decide whether the action can be applied.** Only the precondition does. A `when` whose condition is false is skipped, and the rest of the action happens as usual.
2. **Every condition is evaluated in the state before the action.** No condition sees the changes made by another effect of the same action.
3. **All the effects that fire are applied at once**: first the deletes, then the adds. If the same fact is deleted and added by one action, it ends up true.

Figure 6.2 draws the rules for `(toggle l1)` on a lit lamp. In the state before, `(lit l1)` is true: the first `when` fires and the second does not. The

result is a lamp that is off. Rule 2 is what makes the action correct. If the two effects were executed one after the other, like two `if` statements of a program, the first would put the lamp out and the second, finding it out, would light it again.

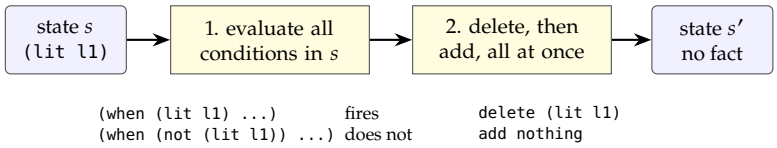


Figure 6.2.: Applying `(toggle l1)` to a lit lamp. Both conditions are read in the state before the action; the effects that fire are then applied together. No condition is evaluated half way.

A when does not block the action: it selects part of its effect. All conditions are read in the state before the action, and all the effects that fire are applied together.

Running it

The first problem has three lamps, of which only `l2` is lit, and asks for exactly `l1` lit. The goal uses negative literals, as the previous chapter taught: `(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)
```

The same action appears twice in the plan and does opposite things: it lights `l1` and puts `l2` out. A plan with conditional effects cannot be read action by action without the state in mind. To know what a step did, we must know where it was applied.

A condition in the effect or in the precondition

The same formula can sit in two places of an action and mean two different things (Table 6.1). In the precondition it says when the action is *possible*. In a `when` it says what the action *does*. The question to ask of the world is simple: when the condition is false, can the agent still do it? A lit lamp does not make the button impossible to press, so the state of the lamp belongs in the effect. A closed door does make walking through it impossible, so the state of the door belongs in the precondition.

	In the precondition	In a `when`
It decides	whether the action applies	which facts the action changes
When it is false	the action cannot be used	the action is used, without that effect
Evaluated in	the state before	the state before
The planner uses it	to prune actions	to predict the outcome

Table 6.1.: The same condition in the two places of an action where it can be written.

The file `lights-strips-domain.pddl` keeps the two actions `switch-on` and `switch-off`. It solves the same problem with the same number of steps, and for a lamp it is a perfectly good model: two cases, two actions. The difference shows when cases combine. A button that changes three lamps has 2^3 cases, one for each combination of lamps that are lit, and

would need eight STRIPS actions. With conditional effects it is still one action, as Section 6.4 will show.

6.3. Universal effects: `forall`

The corridor also has a master switch that puts out every lamp. How many lamps are there? The domain does not know: three in one problem, six in the next. The action `all-off` of Listing 6.2 says it without counting:

```
  ; The master switch: every lamp goes off, whatever it was.
  (:action all-off
    :parameters ()
    :effect (forall (?l - lamp) (not (lit ?l)))))
```

A **universal effect** has the form

```
(forall (<typed variables>) <effect>)
```

and applies the effect once for every combination of objects that the variables can take. The effect inside may mention the new variables and also the parameters of the action. A universal effect is the mirror of the universal precondition of the previous chapter: that one *asked* something of every object, this one *does* something to every object.

No new requirement is involved. In the grammar of PDDL, `forall` in an effect belongs to `:conditional-effects`, like `when`. Kairos confirms it: checking a domain whose only rich effect is a `forall` reports `conditional-effects` among the features in use.

The requirement `:universal-preconditions` is about `forall` in conditions and goals. No requirement is called "universal effects".

The set of objects is finite and known when the problem is read, so a universal effect is a shorthand. With lamps `l1` to `l6`, `all-off` is one ground action with six delete effects. Nothing is decided at planning time; the planner expands the `forall` when it grounds the action. What the modeler gains is a domain that fits every problem.

The second problem has six lamps, five of them lit, and wants only `l2` lit:

```
$ 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)
```

The first plan presses the six buttons: five lamps go off and `l2` goes on. The search goes on and finds the plan that a person would choose: the master switch, then one button. Note that `all-off` deletes `(lit l2)`, which was already false. Deleting a fact that is not there is not an error. It simply changes nothing, and this is what lets a universal effect be written without knowing the state.

A forall in an effect applies it to every object of the type. It is expanded when the action is grounded, and it needs :conditional-effects.

An unconditional `forall` is a blunt tool: *every* lamp, *every* door. It fits effects that reset something (a master switch, an alarm that opens all

the doors, a power cut). We will use one in the cart: when the robot picks up an item, the item is no longer on any cart, whichever cart it was on, and `(forall (?c - cart) (not (on-cart ?i ?c)))` says so without asking which. Most of the time, though, the effect is meant for *some* of the objects, and the `forall` carries a `when` inside.

6.4. Both together: for every object such that

The two constructs combine into the form that makes them useful:

```
(forall (?x - type)
  (when <condition on ?x>
    <effect on ?x>))
```

It reads "for every `?x` such that the condition holds, the effect happens". The `forall` brings every object in; the `when` keeps those concerned. This is the sentence that Section 6.1 could not write. We meet it three times: in a puzzle, in a classic and in the cart.

Lights Out

In the puzzle Lights Out every button is wired to several lamps and changes all of them at once. The goal is to put every lamp out. Which lamps a button changes is part of the board, so it is data of the problem: a static predicate `(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))))))
```

Listing 6.3: Lights Out: for every lamp wired to the button, lit becomes unlit and unlit becomes lit (`lightsout-domain.pddl`).

The action `press` has one parameter and changes any number of lamps. Inside the `forall`, the two `when` are the toggle of Section 6.2 with one more conjunct, `(wired ?b ?l)`. A lamp that is not wired to the button satisfies neither condition and is left alone. This is the frame rule of Chapter 2 at work inside a conditional effect: what no effect mentions does not change.

The first problem is a row of five lamps. Each button changes its own lamp and the neighbors. Lamps 1, 3 and 4 are lit (Figure 6.3).

The domain declares `:universal-preconditions` for the goal of its problems (every lamp out), not for the effect.

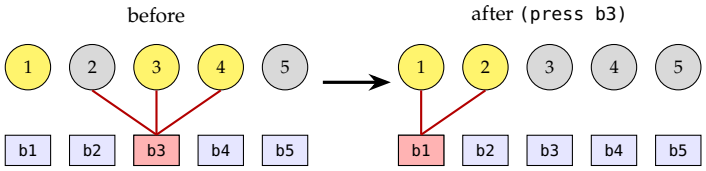


Figure 6.3.: Lights Out on a row of five lamps (yellow: lit). Left, the initial state; the red lines are the lamps wired to `b3`. Pressing it changes lamps 2, 3 and 4 and leaves lamps 1 and 2 lit, which `b1` (right) puts out.

```
$ 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)
```

The first plan presses `b2` twice, which undoes itself; the second is the one of the figure. The second problem differs in one lamp of the initial state (lamps 1, 2 and 4 lit) and has no solution:

```
$ 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)
```

Five lamps have $2^5 = 32$ states, and the planner visited 16. On this board the buttons reach only half of the states from any given one, and "all out" is in the other half. The planner does not know the algebra behind that fact. It searched every reachable state, and its answer is a proof.

The briefcase

The oldest example of these effects is a briefcase. Things can be put in it and taken out of it, and when the briefcase is carried to another place, what is inside goes with it.

```
;; 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)
```

```
24     :precondition (and (case-at ?p) (at ?t ?p)
25                        (not (in ?t)))
26     :effect (in ?t))
27
28   (:action take-out
29     :parameters (?t - thing)
30     :precondition (in ?t)
31     :effect (not (in ?t))))
```

Listing 6.4: The briefcase: carrying it moves every thing that is in it (`briefcase-domain.pddl`).

A thing in the briefcase keeps its `at` fact, and `carry` keeps it right: lines 18 to 20 move every thing that is `in`. In the problem the briefcase is at home with the keys inside. The report, also at home, must go to the office, and the keys must stay at home.

```
$ kairos briefcase-domain.pddl briefcase-p01.pddl
; cost = 3
(put-in report home)
(take-out keys)
(carry home office)
```

No goal asks to take the keys out. The planner does it because it has understood the universal effect: carrying the briefcase with the keys inside would make `(at keys home)` false. The second step is there to *avoid* an effect. We will see more of this in Section 6.5.

ADL and the briefcase

In 1989 Edwin Pednault presented ADL, the Action Description Language, at the first conference on Principles of Knowledge Representation and Reasoning (Pednault, 1989). STRIPS was simple to plan with and poor to model with; the situation calculus of logic was the opposite. ADL took a position between them: actions keep their add and delete form, but conditions may be any first-order formula and effects may be conditional and quantified. A briefcase whose contents move with it was the example. When PDDL was defined for the first planning competition in 1998 (McDermott et al., 1998), it adopted these features, each with its own requirement flag, and one flag for all of them: `:adl`.

The cart

We can now write the sentence that the cart needed. First the method: what the cart adds to the tables of the courier world.

Steps O and D: the cart

Model card, what changes: the Story gains a cart that a robot loads and pushes; Scope now includes where the cart is and what is on it, and still leaves out how much fits on it (any number of items). Assumption: a robot that pushes a cart has no hand free.

State table, new rows:

Property	About	Kind
`(cart-at ?c ?l)`	a cart: the room where it is	dynamic
`(on-cart ?i ?c)`	an item and a cart: the item lies on it	dynamic
`(pushing ?r ?c)`	a robot and a cart: the robot holds its handle	dynamic

Action table, new and changed rows. The last column is new: what the action changes *depending on the state*.

Action	Always changes	Also changes
`pick`	as before	the item leaves *every* cart
`load`	the hand is free, the item is on the cart	
`grab-cart`	the robot holds the cart, hand busy	
`release-cart`	the opposite	
`push-cart`	robot and cart change room	*every* item, *when* it is on the cart, changes room

Listing 6.5 is the domain. It starts from the typed courier domain of Chapter 4 and keeps its names.

```
;; 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)
    (free ?r - robot)
    (cart-at ?c - cart ?l - location)
    (on-cart ?i - item ?c - cart)     ; ?i lies on the cart
    (pushing ?r - robot ?c - cart))   ; ?r holds the handle

  ; A robot that holds a cart cannot walk away from it.
  (:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                       (not (exists (?c - cart) (pushing ?r ?c))))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to)))

  ; Take an item from the floor or from a cart: whatever cart
  ; it was on, it is on none afterwards.
  (: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)
                 (forall (?c - cart) (not (on-cart ?i ?c)))))

  (: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)))

  ; Put the carried item on a cart that is in the same room.
  (: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)
                 (item-at ?i ?l) (on-cart ?i ?c)))

  (:action grab-cart
    :parameters (?r - robot ?c - cart ?l - location)
    :precondition (and (robot-at ?r ?l) (cart-at ?c ?l) (free ?r))
    :effect (and (not (free ?r)) (pushing ?r ?c)))

  (:action release-cart
    :parameters (?r - robot ?c - cart)
    :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)))))))
```

Listing 6.5: The courier world with a cart (`courier-cart-domain.pddl`).

Four points of the listing deserve a comment.

- `push-cart` (lines 61 to 70) is the sentence of Section 6.1, almost word for word. Its universal conditional effect has the same shape as the one of the briefcase. With an empty cart no condition holds and only the robot and the cart move.
- The conditional effect contains a delete *and* an add. An item that moves must leave the old room as well as enter the new one, exactly as in a plain `move`.
- `pick` (line 34) takes an item from the floor or from a cart with the same action. The unconditional `forall` deletes `on-cart` for every cart, which changes nothing if the item was on none.
- `move` (line 23) forbids walking away while holding the handle, with a negated existential precondition. `grab-cart` takes away `(free ?r)`, so a robot that pushes cannot pick anything either.

The first problem is the one of Figure 6.1, except that the boxes start on the floor of the pharmacy: the robot is in the dock, and the goal is `(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)
```

The first plan ignores the cart and carries the boxes one by one: 18 steps, of which 12 are walking. The last one, which Kairos proves optimal, loads both boxes and pushes four times. The goal becomes true at the fourth push, as an effect of an action whose parameters do not mention any box. The plan ends there, with the boxes on the cart in the ICU: the goal asked where the boxes are, not that they lie on the floor.

The second problem has three items and adds a goal: the cart must be back in the pharmacy. The best plan found, of 25 steps, ends like this:

```
$ 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)
```

Before the cart goes back, the robot lets go of the handle, takes each item off the cart and puts it down. Nothing in the goal mentions `on-cart`. The planner unloads because what stays on the cart would return to the pharmacy with it. As with the keys of the briefcase, part of the plan exists only because of what a universal effect would otherwise do.

The forall brings every object in and the when keeps those concerned. One action then changes as many facts as the state requires, with parameters that do not name them.

6.5. Effects to avoid: the sterile laboratory

Until now a conditional effect was something the plan wanted (the boxes arrive) or had to work around (the keys stay). The laboratory brings a conditional effect that is plainly bad. The lab is a sterile room. Its door is an airlock that sprays disinfectant on everything that enters. Medicines and meals come through unharmed. A sample in a sealed bag does too. A sample that is not sealed is spoiled by the spray: it is contaminated and useless for analysis. The robot can seal the sample that it carries in a bag, at any moment.

In the model card the Scope gains two properties of samples and one of rooms, with the names of the shared vocabulary: `(sealed ?i)`, `(contaminated ?i)` and the static `(sterile ?l)`. Two assumptions are added: a seal is never opened again, and a contaminated sample stays contaminated. Items now have subtypes, since only samples can be sealed or contaminated. The new action and the changed one are in Listing 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)))
```

Listing 6.6: The move through an airlock and the seal (from `courier-sterile-domain.pddl`; `pick` and `drop` do not change).

The airlock does not stop anybody, so nothing was added to the precondition of `move`. What entering does to a sample is in the effect, and Figure 6.4 unfolds its condition. The `forall` is there for a reason of syntax: `move` has no parameter for the item, and it cannot have one, because the robot may be carrying nothing. "Every sample that the robot carries" covers the cases of none and one with the same text.

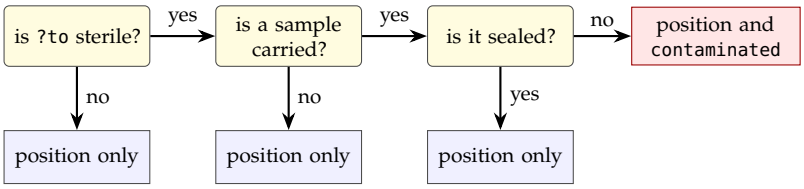


Figure 6.4.: What `(move ?r ?from ?to)` changes. The action is applicable in the four cases; three conditions, all read in the state before the move, select the one in which the sample is also contaminated.

Exercising the model

An effect that depends on the state needs a test set that visits its cases. Step E of the MODEL method asks for four problems, and here each of them has a precise job.

Step E: a test set for a conditional effect

Trivial (`p01`): a box of medicines to the lab. The condition of the effect is never true; the plan must be the one of the old courier world.

Typical (`p02`): the sample of ward A to the lab, with the goal `(not (contaminated sample1))`. The plan must seal it before the airlock.

No plan (`p03`): the lab wants the sample open *and* usable. Impossible in this world, so the planner must say so.

Cheat (`p04`): the goal only asks for the sample in the lab. We expect the planner to take the shortcut, and we want to see it.

The trivial problem gives a plan of seven steps, like the delivery of Chapter 4. The typical one:

```
$ 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)
```

The robot seals the sample in the corridor, one step before the airlock. Any moment after the pick would do; what matters is that `(sealed sample1)` is true in the state where `(move r1 corridor lab)` is applied. The third problem ends with `kairos: unsolvable (exhausted)` after 71 states, as it should. The fourth:

```
$ 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)
```

One step shorter, and the sample arrives contaminated. Nothing is wrong with the domain: it describes what the airlock does, and a robot can indeed walk an open sample into it. The problem is the goal, which forgot to say what matters. A planner does not avoid a bad effect because it is bad. It avoids it when a goal (or a later precondition) needs the opposite. The validator makes the difference visible: the plan above, saved in a file, is valid for `p04` and is rejected for `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
```

A conditional effect can be a danger. The planner avoids it only if the goal or a later precondition needs what it destroys: every harm that matters must be named in the goal.

One action per case

The move through the airlock has few cases, so the alternative of Section 6.1 is reasonable here. The file `courier-sterile-cases-domain.pddl` splits `move` in three:

```
21  ; Case 1: the room entered has no airlock.
22  (:action move
23    :parameters (?r - robot ?from ?to - location)
24    :precondition (and (robot-at ?r ?from) (connected ?from ?to)
25                       (not (sterile ?to)))
26    :effect (and (not (robot-at ?r ?from))
27                 (robot-at ?r ?to)))
28
29  ; Case 2: airlock, and no unsealed sample is carried.
30  (:action enter-clean
31    :parameters (?r - robot ?from ?to - location)
32    :precondition (and (robot-at ?r ?from) (connected ?from ?to)
33                       (sterile ?to)
34                       (forall (?i - sample)
35                         (imply (carrying ?r ?i) (sealed ?i))))
36    :effect (and (not (robot-at ?r ?from))
37                 (robot-at ?r ?to)))
38
39  ; Case 3: airlock, and the unsealed sample ?i is carried.
40  (:action enter-exposed
41    :parameters (?r - robot ?i - sample ?from ?to - location)
42    :precondition (and (robot-at ?r ?from) (connected ?from ?to)
43                       (sterile ?to)
44                       (carrying ?r ?i) (not (sealed ?i)))
45    :effect (and (not (robot-at ?r ?from))
46                 (robot-at ?r ?to)
47                 (contaminated ?i)))
```

Listing 6.7: The airlock without conditional effects: three actions for the three cases of a move (from `courier-sterile-cases-domain.pddl`).

The preconditions of the three actions exclude each other and cover every state, so exactly one of them applies wherever the old `move` applied. The plan for the typical problem is the same, with `(enter-clean r1 corridor lab)` in place of the move into the lab. The plan now says in its own text that the entry was clean, which is easier to read than a `move` that may or may not have contaminated something. The costs are the repetition (the position effect is written three times) and what happens when the

rule grows. A second independent factor, say a robot that must itself be disinfected unless it wears a cover, would double the cases. With conditional effects it would be one more `when`.

A third reading of the story is possible: an airlock that *refuses* to open for an unsealed sample. That is a precondition, and a different world: in it no plan ever contaminates anything, whatever the goal says. Which one is right is a question for step M, about the real airlock. Exercise 6.5.2 builds that model and compares the test sets.

6.6. The requirement `:adl`

With conditional effects we have met every requirement of classical PDDL. Since they are often used together, PDDL has a shorthand: `:adl` stands for the seven requirements of Table 6.2.

Requirement	What it allows	Chapter
`:strips`	add and delete effects	Chapter 2
`:typing`	types of objects and parameters	Chapter 4
`:equality`	= between objects	Chapter 4
`:negative-preconditions`	`not` in conditions and goals	Chapter 5
`:disjunctive-preconditions`	`or` and `imply`	Chapter 5
`:quantified-preconditions`	`exists` and `forall` in conditions	Chapter 5
`:conditional-effects`	`when` and `forall` in effects	this one

Table 6.2.: The seven requirements that `:adl` abbreviates.

A domain may write `(:requirements :adl)` and use any of them. This book keeps listing the requirements that each file really uses. The list is a summary of the file for the reader, and it is the first thing to look at when a planner refuses a domain: a planner that supports only some of the seven can accept a domain that declares what it uses and must refuse one that declares `:adl` to use a single negation.

Classical planning ends here

Types, equality, negation, disjunction, quantifiers and conditional effects do not change what a planning problem *is*. States are still sets of facts, actions still take one state to another, and a plan is still a sequence of actions. Every ADL model with a fixed set of objects can be rewritten as a STRIPS model that has the same plans, or plans that correspond one to one (Section 6.7 shows how for the effects). ADL is a better language for the same class of problems. The parts of the book that follow are different: numbers, time and preferences change the problem itself.

6.7. What planners do with conditional effects

Chapter 3 explained that a planner grounds the model before searching. Rich effects are treated in that phase, in up to three steps. Knowing them tells the modeler what each construct costs.

Universal effects are expanded. A `forall` becomes one effect for each object. Conditions on static predicates are decided at this point and disappear. In Lights Out the action `(press b3)` starts with two conditional effects for each of the five lamps; `(wired b3 l1)` is false forever, so the

effects for `l1` and `l5` are dropped, and the ground action keeps six: two for each of the three lamps wired to the button. Static data in the condition of an effect costs nothing at search time, just as in a precondition.

Conditional effects are kept, or compiled away. A planner whose search works with conditional effects keeps them inside the ground action. A planner that only knows STRIPS actions must first rewrite the model, and there are two ways to do it (Figure 6.5).

- **One action per combination.** An action with k conditional effects is replaced by 2^k actions, one for each choice of which conditions hold, with those choices added to the precondition. Plans keep their length. The model can grow exponentially: `(press b3)`, with three lamps, becomes eight actions.
- **Bookkeeping actions.** The action is cut into a sequence: first the unconditional part, then one small step for each conditional effect that either applies it or skips it, then a step that ends the sequence. Extra facts force the steps to be taken in order, and the conditions must be remembered so that they are still read as they were before the action. The model grows little; the plans grow longer.

There is no third way that avoids both costs. Bernhard Nebel proved in 2000 Nebel, 2000 that conditional effects cannot be compiled into plain STRIPS while keeping both the model small and the plans essentially as short. This is why conditional effects count as a real gain of expressive power, not a convenience of notation.

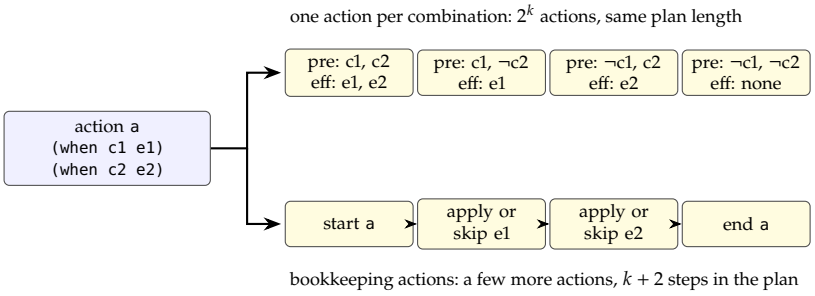


Figure 6.5.: Two ways of removing the conditional effects of an action. Above, the model grows; below, the plans do.

The heuristic looks at each effect on its own. The relaxation of Chapter 3 ignores deletes. With conditional effects, planners in the tradition of FF treat every conditional effect as a small action whose precondition is the precondition of the action plus its own condition (Hoffmann & Nebel, 2001). The estimate may count one real action several times, once for each of its effects that it needs, but it stays cheap to compute.

Kairos keeps conditional effects in its ground actions. The two models of the lamps show it:

```
$ 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)
```

Three lamps give four ground actions with conditional effects (three `toggle` and `all-off`) and six without (three `switch-on` and three `switch-off`). The toggle was not split.

Not every planner goes this far, and this is the weak point of the feature. In the tests behind the support table of the Kairos documentation, with one small domain per feature, only Kairos, ENHSP and LAMA returned a valid plan for the domain with conditional effects; OPTIC, POPF, TFD, LPG-td and Metric-FF did not. A model that must run on several planners, in particular on temporal ones, pays for each `when`.

Universal effects are expanded at grounding. Conditional effects are kept by the planners that support them; compiling them away costs either an exponential number of actions or longer plans.

6.8. The MODEL method: effects that depend on the situation

Step D of the method describes the dynamics: for each action, when it applies and what it changes. With STRIPS effects the second part was a list. This chapter adds two questions to it.

Step D: two new questions

For each action of the action table:

1. **Does what the action changes depend on the situation?** Describe the action to yourself in words. If the description has an "if" or an "unless" that does not stop the agent from acting (*it can always press, enter, push; what happens depends*), the action has a conditional effect. If the "if" does stop the agent, it is a precondition.
2. **Does it change many objects at once?** If the description says "every", "all" or "whatever is in, on or near", the action has a universal effect, usually with a condition that selects the objects.

Write the answers in a third column of the action table: *always changes* and *also changes, when*.

And for step E: for each conditional effect, a test problem in which its condition holds, one in which it does not, and a cheat test whose goal can be reached more cheaply by triggering (or by not triggering) the effect.

When the answer to one of the questions is yes, a conditional effect is the direct encoding. It is not the only one, and step L asks to look at the others.

Alternatives compared

We compare three encodings of the cart. All three keep the decision that an item on the cart has an `item-at` fact, and differ in how a push keeps it right.

A. Conditional effect. The domain of Listing 6.5: `push-cart` with a `forall` and a `when`.

B. One action per case. The domain of Listing 6.1, with its four push actions, each naming the items that move.

C. A bookkeeping action. The push moves the robot and the cart only. The items follow one by one with an extra action, and a third action closes the move (Listing 6.8). It is the second compilation of Figure 6.5, done by hand.

```
75  ; Step 1: the robot and the cart change room; the items wait.
76  (:action push-cart
77    :parameters (?r - robot ?c - cart ?from ?to - location)
78    :precondition (and (not (unsettled))
79                       (pushing ?r ?c) (robot-at ?r ?from)
80                       (connected ?from ?to))
81    :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
82                 (not (cart-at ?c ?from)) (cart-at ?c ?to)
83                 (rolled ?c ?from ?to) (unsettled)))
84
85  ; Step 2, once for each item on the cart: it follows the cart.
86  (:action bring-along
87    :parameters (?i - item ?c - cart ?from ?to - location)
88    :precondition (and (rolled ?c ?from ?to) (on-cart ?i ?c)
89                       (item-at ?i ?from))
90    :effect (and (not (item-at ?i ?from)) (item-at ?i ?to)))
91
92  ; Step 3: when no item is left behind, normal life goes on.
93  (:action settle
94    :parameters (?c - cart ?from ?to - location)
95    :precondition (and (rolled ?c ?from ?to)
96                       (forall (?i - item)
97                         (imply (on-cart ?i ?c)
98                                (item-at ?i ?to))))
99    :effect (and (not (rolled ?c ?from ?to))
100                (not (unsettled)))))
```

Listing 6.8: Encoding C: the push in three kinds of steps (from `courier-cart-settle-domain.pddl`). Every other action of the domain has the precondition `(not (unsettled))`.

In encoding C, `push-cart` raises a flag, `(unsettled)`, that blocks every normal action. While it is up, only `bring-along` and `settle` apply; `settle` requires that no item on the cart is still in the old room, and lowers the flag. A push with two boxes on the cart takes four steps of the 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)
```

Table 6.3 compares the three on the five criteria, with the numbers that `kairos -v` reports for the problem of three items and a cart that must return (`p02`).

	A. Conditional effect	B. Action per case	C. Bookkeeping action
Fidelity	exact	a cart with four items cannot move	exact only if every goal also asks for `(not (unsettled))`
Size	109 ground actions	343 ground actions; grows as n^k	189 ground actions
Guidance	best plan: 25 steps, with the cart	25 steps, with the cart	a push costs 2 to 5 steps: the best plan (28 steps) avoids the cart
Clarity	one action, one sentence	four actions that say the same	three actions and a flag in every precondition
Portability	needs `:conditional-effects`	needs quantified preconditions	needs quantified preconditions

Table 6.3.: Three encodings of the push of a cart, compared on problem `p02` (three items, the cart returns).

Two rows need a comment. On Guidance, encoding C changes what the length of a plan means. Its steps are no longer things that the robot does, and a planner that looks for short plans sees the cart as expensive: for `p02` it prefers to carry the three items by hand. Chapter 9 gives the cure, a cost of zero for bookkeeping actions. On Fidelity, the flag opens a hole. In `courier-cart-settle-p03` the goal is the loaded cart in the lobby, and the plan stops right after the push, with the cart in the lobby and its boxes, according to `item-at`, still in the pharmacy. A model with bookkeeping actions has states that mean nothing in the world, and the goal must exclude them.

On Portability neither B nor C is plain STRIPS as written here: both use a universal precondition to say "nothing else is on the cart". Removing it is possible, by counting the items on the cart with the techniques of Chapter 7, and makes both domains longer.

A fourth way: do not store what can be deduced

All three encodings work to keep a redundant fact up to date. The position of an item on the cart is already known: it is where the cart is. If `load` deletes `item-at` and the item gets a position again only when a robot takes it off the cart and drops it, the push has nothing to update:

```
  ; 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)))
```

Listing 6.9: Encoding D: an item on the cart has no position of its own (from `courier-cart-implied-domain.pddl`). The `push-cart` of this domain is the plain STRIPS action that moves the robot and the cart.

This is how the courier world treated a carried item from the start: `pick` deletes `item-at`, and nobody missed it. On `p02` the encoding has 133 ground actions and the same best plan of 25 steps. Its states are also simpler for the planner. With `-v`, Kairos reports 13 state variables for D and 37 for A, because in D it can prove that each item is in exactly one place (on the floor of a room, in a hand or on a cart) and pack those facts together.

The price is paid in Fidelity, on the side of the questions that the model can answer. In D, `(item-at meds1 icu)` is false while the box is on the cart in the ICU. The first problem of the cart, whose goal was only that fact, now needs 16 steps instead of 11: the robot must unload and drop each box. If "on the cart in the ICU" counts as delivered, the goal has to say so with a disjunction and a quantifier, and so does every precondition that asks where an item is.

Verdict.

- Use a **conditional effect** (A) when the action is one thing in the world and its outcome depends on the state, above all when it

touches a number of objects that the domain cannot bound. It is the most faithful and the clearest encoding. Its cost is portability.

- Use **one action per case** (B) when the cases are few and fixed (two or three), as in the airlock, or when the plan should say which case happened. Avoid it when the cases are combinations of objects.
- Use a **bookkeeping action** (C) only when the planner at hand does not support conditional effects and the cases cannot be listed. Give every goal the condition that the bookkeeping is finished, and remember that plan length no longer measures effort.
- Before any of the three, ask whether the fact that the effect updates must be stored at all (D). If it can be deduced from another one, storing the other one alone removes the need for the effect. The condition is that nothing else in the model needs to read it directly.

6.9. Common mistakes

Reading the effects as a program

The effects of an action are not executed in order. A toggle written as "put it out if it is lit, and light it" loses the condition of its second half:

```
    :effect (and (when (lit ?l) (not (lit ?l)))
                 (lit ?l))))
```

On a lit lamp this action deletes and adds `(lit ?l)`. Adds are applied after deletes, so the lamp stays lit, and the lamp of `lights-clash-p01.pddl` can never be put out:

```
$ 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)
```

One state searched: the action exists and leads nowhere. **Fix:** give every case its own `when`, with conditions that exclude each other, and remember that all of them are read in the state before the action.

The forgotten `when`

A universal effect without its condition applies to every object. This push forgot to ask which items are on the cart:

```
                (forall (?i - item)
                  (and (not (item-at ?i ?from))
                       (item-at ?i ?to))))))
```

The file is valid and the planner is delighted with it. For the two boxes that must reach the ICU:

```
$ 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)
```

Nobody loaded anything: every item of the hospital follows the cart. **Symptom:** a plan that is too short and never touches the objects of the goal. **Fix:** the `when`, and a cheat test with an item that is *not* on the cart.

A `when` in a precondition

`when` exists only in effects. In a condition, "if the lamp is lit it must not be broken" is written with `imply`. Kairos does not stop at this mistake; it warns that it read `when` as the name of a predicate that nobody declared:

```
$ 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" means that the action can never be applied. Other planners reject the file. **Fix:** `(imply (lit ?l) (not (broken ?l)))`, with `:disjunctive-preconditions`. Read the warnings of `check` even when the verdict is `valid`.

The requirement left out

A domain that uses `when` or a `forall` effect without declaring `:conditional-effects` gets a warning from Kairos, which plans with it anyway:

```
$ 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
```

A stricter planner refuses the file, or worse, ignores the effect. **Fix:** declare what the file uses; the list printed by `check` tells which features those are.

A `when` inside a `when`

The effect of a `when` is a literal or an `and` of literals. The grammar of PDDL has no `when` inside a `when`, and no `forall` inside one either. Kairos accepts the nested form (`lights-nested-domain.pddl` checks as valid); a planner that follows the grammar strictly does not. **Fix:** join the conditions. `(when a (when b e))` is `(when (and a b) e)`, and a `forall` goes outside the `when`.

A harm that the goal does not watch

The plan of `courier-sterile-p04` delivered a contaminated sample, and the plan of `courier-cart-settle-p03` left the boxes behind. Both models were right and both goals were incomplete. **Symptom:** a plan shorter than expected in which a step that you thought necessary

(the seal, the unloading) is missing. **Fix:** for every conditional effect that can spoil something, decide whether the goal must protect it, and keep the cheat test in the test set.

6.10. Summary

Effects can now depend on the state and reach any number of objects. Table 6.4 collects the constructs of the chapter.

Table 6.4.: The constructs of this chapter.

Construct	Meaning	Used for
`(when c e)`	`e` happens if `c` holds in the state before the action	the toggle; the airlock
`(forall (?x - t) e)`	`e` happens for every object of type `t`	the master switch; leaving every cart
`(forall (?x - t) (when c e))`	`e` happens for every object that satisfies `c`	Lights Out; the briefcase; the cart
`:conditional-effects`	the requirement of the three forms	
`:adl`	the seven classical requirements at once	

The ideas to keep:

- A precondition says whether the action can be done; the condition of a `when` says what it does. Both are read in the state before the action, and all the effects are applied together, deletes first.
- A universal effect is expanded over the objects of the problem when the action is grounded. It lets one domain serve problems of any size.
- A conditional effect can be something to avoid. The planner works around it only when a goal or a precondition needs what it would destroy.
- Step D asks two more questions: does what the action changes depend on the situation, and does it change many objects at once. Step E tests each conditional effect with its condition true, with it false, and with a goal that tempts.
- A conditional effect competes with one action per case, with bookkeeping actions and with not storing the fact at all. The first is the clearest and the least portable.

6.11. Further reading

- Pednault (1989) is the paper that defined ADL, with the semantics of conditional and quantified effects that PDDL inherited.
- McDermott et al. (1998), the manual of the first PDDL, introduces the requirement flags of Table 6.2; the grammar of effects as it stands today is in Kovacs (2011).
- Haslum et al. (2019) treat conditional and universal effects in their presentation of classical PDDL, with a discussion of how planners support them.
- Hoffmann and Nebel (2001) describe how FF handles ADL: quantifiers expanded at grounding, and conditional effects kept and seen by the heuristic one by one.

- Helmert (2009) explains the translation of a PDDL task with quantifiers and conditional effects into the state variables that Fast Downward and many later planners search with.
- Ghallab et al. (2004) discuss, in their chapter on classical representations, how the extensions of this chapter relate to plain STRIPS.

6.12. Exercises

Exercises for Section 6.1 — Effects that depend on the situation

Exercise 6.1.1. A hospital has 5 items and the map of the courier world (14 directed connections), with one robot and one cart. Using the one-action-per-case encoding, how many candidate ground actions do `push-cart-0` to `push-cart-3` have in total, before the planner prunes any? How many would `push-cart-4` add? Then run `kairos -v` on `courier-cart-cases-p02.pddl` and explain why the number of ground actions it reports is much smaller than your count for 3 items.

Exercise 6.1.2. For each sentence, say whether a STRIPS action with a fixed list of effects can model it, and if not, which ingredient is missing (a condition in the effect, a quantifier, or both): (a) "closing the valve stops the flow in its pipe"; (b) "closing the main valve stops the flow in every pipe"; (c) "the elevator goes up one floor, with whoever is inside"; (d) "pressing the call button lights it, unless the elevator is already at the floor"; (e) "the robot drops the item it carries".

Exercises for Section 6.2 — Conditional effects: `when`

Exercise 6.2.1. An action has the effect `(and (when (p) (q)) (when (q) (r)) (when (r) (not (p))))`, where `p`, `q` and `r` are facts without arguments. It is applied to the state {`(p)`}, and then again to the result, four times in all. Write the four states. Which state would the first application give if the three effects were executed in order, like a program?

Exercise 6.2.2. A lamp with a dimmer has three levels (off, low, high) and one button that goes from off to low, from low to high and from high to off. Write the domain `dimmer` with predicates `(low ?l)` and `(high ?l)` and a single action `press`, and a problem with two lamps in which one must go from low to off and the other from off to high. How many steps does the plan have?

Exercise 6.2.3. In `courier-sterile-domain.pddl`, the action `seal` has the precondition `(not (sealed ?i))`. Remove it and run the four test problems again. Does any plan change? Does the number of ground actions change? Give one argument for keeping the precondition and one for removing it.

Exercises for Section 6.3 — Universal effects: `forall`

Exercise 6.3.1. In a floor with doors (predicates `joins`, `open` and `locked` as in Chapter 5), a fire alarm button unlocks and opens every door. Write a domain `alarm` with actions `move` (through an open door), `open-door` (only if not locked) and `raise-alarm` (the robot must

be in a room with a button), and a problem in which the only way to enter a locked room is to raise the alarm.

Exercise 6.3.2. The goal `(forall (?l - lamp) (not (lit ?l)))` and the effect `(forall (?l - lamp) (not (lit ?l)))` are written with the same text. Explain what each one means, which requirement each one needs, and what a planner does with each when the problem has six lamps.

Exercises for Section 6.4 — Both together: for every object such that

Exercise 6.4.1. Edit `lightsout-p01.pddl` so that the five lamps start lit. Before running the planner, find by hand a set of buttons that puts all of them out. Does the order of the presses matter? Does pressing a button twice ever help? Check your answers with Kairos.

Exercise 6.4.2. A robot that holds the handle of a cart and walks is pushing the cart. Rewrite `courier-cart-domain.pddl` without `push-cart`: a single `move` that also moves the cart held by the robot, if any, and what is on it. Solve `courier-cart-p02` with it and compare the plan and the number of ground actions with those of the chapter.

Exercise 6.4.3. The cart domain was tested with one robot. Write a problem with two robots, `r1` and `r2`, and one cart, whose goal is `(and (cart-at cart1 icu) (cart-at cart1 lab))`. No plan should exist. Run it, read the plan that Kairos returns, find the flaw in the domain and repair it.

Exercises for Section 6.5 — Effects to avoid: the sterile laboratory

Exercise 6.5.1. Join the cart and the sterile lab in one domain (call it `courier-cart-sterile`). Decide what the airlock does to the samples that enter on a cart, and write a problem in which two samples of ward A must reach the lab, usable. Read the plan: where does the robot seal them?

Exercise 6.5.2. Write the variant of the sterile lab in which the airlock does not open for a robot that carries an unsealed sample (a precondition, no conditional effect). Run the "no plan" and the "cheat" problems of the chapter with it. Which results change? In which kind of hospital would each of the two models be the faithful one?

Exercises for Section 6.6 — The requirement `:adl`

Exercise 6.6.1. Replace the requirements of `lights-domain.pddl` with `:adl` and check the file. Which features does `kairos check` report in use, and are they all the features that `:adl` announces? Explain why a planner without disjunctive preconditions could run the original file and might refuse the new one.

Exercises for Section 6.7 — What planners do with conditional effects

Exercise 6.7.1. Take the problem `lightsout-p01.pddl` and compile by hand its ground action `(press b1)` into STRIPS actions with negative preconditions, one for each combination of its conditions. How

many are there? How many would `(press b3)` need, and how many the five buttons together?

Exercise 6.7.2. Run `kairos -v` on `courier-sterile-p02` with the domain of the chapter and with `courier-sterile-cases-domain.pddl` (problem `courier-sterile-cases-p02`). Compare the number of ground actions and of states searched. Which of the two compilations of Figure 6.5 does the second domain correspond to?

Exercises for Section 6.8 — The MODEL method: effects that depend on the situation

Exercise 6.8.1. A painter paints rooms with a spray gun. Spraying a room paints its walls and also stains every piece of furniture that is in the room and is not covered with a sheet. The painter can walk between rooms joined by a door, cover a piece of furniture, and carry a piece of furniture to the next room. Apply steps O and D of the method (state table, action table with the column of this chapter), write the domain `paint`, and test it with a problem in which the bedroom must be painted without staining the bed nor the lamp.

Exercise 6.8.2. Apply step L to the briefcase. Write the encoding in which a thing in the briefcase has no `at` fact (encoding D of this chapter), solve the same problem with it, and compare it with the domain of Listing 6.4 on the five criteria. Which requirement did the conditional effect cost, and what did it buy?

Modeling classical problems: patterns and choices

7.

The last three chapters each added a piece of the language: types, rich conditions, rich effects. With them we can write any classical model. Whether we can write a *good* one is another matter. Hand the same story to three people (a waiter robot carries dishes on a tray with two places, each course after the previous one) and three different domains come back. One has a predicate for each place of the tray, another counts the dishes, the third has no tray at all. All three pass `kairos check`. They differ in which plans they allow, in how many ground actions they produce and in how well a planner copes with them. Knowing the syntax did not decide any of that.

This chapter is about those decisions. It has no new PDDL. It takes the MODEL method, which has grown a question at a time since Chapter 2, and runs it whole on a new problem, writing out the five artifacts. Then it gives names to the solutions that modelers reach for again and again: eleven **patterns**, each with the situation that calls for it, a tested example and what it costs, and four **anti-patterns**, each with the symptom that gives it away. Later chapters and the catalog of Chapter 19 refer to these names.

The chapter starts with the waiter, from the story to a tested model. Three sections follow with the patterns: for the state, for control, and for the actions. Then come the anti-patterns and a procedure for the two bad outcomes of step E, a model with no plan and a model with a wrong plan. The method section collects the questions of Part II in one checklist, indexes the patterns, and closes the case of the waiter with step L.

7.1 A problem from a story: the waiter 138
Step M: map the situation 139
Step O: objects and state 139
Step D: dynamics 140
Step E: exercise the model 140
7.2 Patterns for the state . . 141
Static table 142
Mirrored facts 143
At-most-one invariant . . 143
Successor chain 144
Slots 145
7.3 Patterns for control . . . 145
Phase token 145
Turn token 146
Resource lock 147
7.4 Patterns for the actions . 148
Inverse pair and one-way action 148
Merged action 150
7.5 Anti-patterns 150
Forgotten delete 150
Goal in the initial state . 151
Out-of-step twins 151
Idle parameter 152
7.6 Debugging a model . . . 152
No plan 152
A wrong plan 154
7.7 The MODEL method: the classical method, consolidated 154
Alternatives compared . 155
7.8 Common mistakes 156
7.9 Summary 157
7.10 Further reading 157
7.11 Exercises 158

7.1. A problem from a story: the waiter

A small restaurant wants a robot to wait on its tables. The owner describes the job:

> *The cook leaves the finished dishes on the pass, the counter between the kitchen and the dining room. The waiter takes them to the tables on a tray. The tray has two places. Every dish is for one table, and at each table the courses come in order: nobody gets the fish before the soup. The evening has gone well when every dish that was ordered is on its table.*

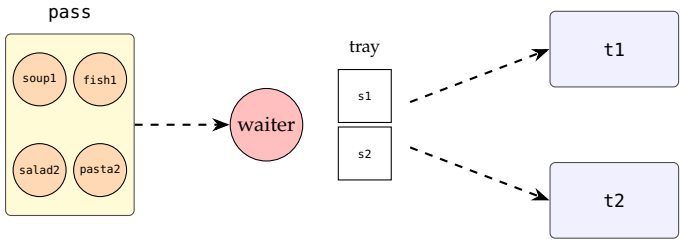


Figure 7.1.: The restaurant of the story. Four dishes wait on the pass; the tray has two places; the soup and the fish are for table 1, the salad and the pasta for table 2.

Step M: map the situation

The story is short, and it already leaves things unsaid. Step M makes them explicit before any PDDL is written. Each line of the model card is a decision that somebody could have taken differently.

Step M: the model card of the waiter

Story. A waiter robot carries dishes from the pass to the tables, two at a time at most, respecting the order of the courses at each table.
Agents. One waiter. The cook and the guests are not agents: they do nothing that the planner decides.
Success. Every ordered dish is on its table.
Scope. *In*: where the waiter is, what is on the tray, which dishes are served, which dish goes before which. *Out*: cooking (dishes are ready when the problem starts), the route between places (the room is open and the robot's navigation finds its way), time, eating, clearing the tables, payment.
Assumptions. A served dish stays served. A dish taken from the pass does not go back. The waiter never drops anything.

Two lines deserve a second look. Leaving the route out means that the model will have no map: going from any place to any other is one action. That is a choice about **abstraction**, and it is right if somebody else (the navigation software) solves what we leave out. The second assumption, that a dish does not go back to the pass, looks harmless. We will see in Section 7.4 that it creates dead ends.

Step O: objects and state

Which things exist? Places (the pass and the tables), dishes, and the places of the tray, which we call slots. The waiter is one and always the same, so it needs no object: its position is a predicate with one argument. The pass exists in every problem and the actions talk about it, so it is a constant.

Property	About	Kind	Pattern
`(waiter-at ?p)`	the place where the waiter is	dynamic	at-most-one
`(ready ?d)`	a dish waits on the pass	dynamic	
`(on-tray ?d ?s)`	a dish is in a slot of the tray	dynamic	slots
`(empty ?s)`	a slot holds nothing	dynamic	slots
`(served ?d)`	a dish is on its table	dynamic	
`(ordered ?d ?t)`	the table that asked for a dish	static	static table
`(before ?a ?b)`	dish `?a` is served before `?b`	static	static table

Table 7.1.: The state table of the waiter. The last column names the pattern behind each row; the next sections define them.

The order of the courses is the interesting row. We could have types `starter` and `main` and a rule in the domain. Instead the order is data: the problem lists which dish goes before which, and the domain only knows that such pairs exist. A dessert, or a table that skips the starter, will then need no change in the domain.

Step D: dynamics

Action	Applies when	Changes
`move ?from ?to`	the waiter is at `?from`; the places differ	the waiter is at `?to`
`load ?d ?s`	the waiter is at the pass; `?d` is ready; `?s` is empty	`?d` is in `?s`, no longer ready; `?s` is not empty
`serve ?d ?s ?t`	the waiter is at `?t`; `?d` is in `?s` and was ordered by `?t`; every dish before `?d` is served	`?d` is served; `?s` is empty again

Table 7.2.: The action table of the waiter. No effect depends on the state, so the third column of Chapter 6 is not needed.

The two tables translate into Listing 7.1 almost line by line. The features come from the three previous chapters: a constant and an inequality (Chapter 4), a universal precondition with `imply` for "every dish before this one" (Chapter 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))))
```

Listing 7.1: The waiter (`waiter-domain.pddl`).

Step E: exercise the model

The test set has the four problems that the method asks for. All use two slots.

Trivial (`p01`): one table, one dish. Expected: load, move, serve.
Typical (`p02`): the four dishes of Figure 7.1.
No plan (`p03`): the soup of table 1 never left the kitchen (it is not `ready`), and the goal asks for the fish.
Cheat (`p04`): the goal asks only for the two main courses. The short way is to serve them and skip the starters.

The trivial problem gives the three expected steps. For the typical one Kairos prints three plans, of 16, 13 and 11 steps, and proves the last one optimal:

```
$ 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)
```

The plan takes each table's two dishes in one trip and serves the starter first. It is what a good waiter does, and nobody told the model about trips: it follows from the two slots and from the cost of moving. The third problem ends with `kairos: unsolvable (exhausted)`. The cheat test returns an 11-step plan that serves the four dishes, although the goal names two. To be sure that the shortcut is impossible, and not only overlooked, we write it by hand and ask the validator:

```
$ 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
```

A cheat test has two halves: the planner does not take the shortcut, and the shortcut written by hand is rejected by the validator, for the reason we expect.

Four of the five artifacts are on the table: the model card, the state table, the action table and the test set. The fifth, the comparison of step L, needs alternatives, and the alternatives are made of patterns. We come back to the waiter in Section 7.7.

7.2. Patterns for the state

A **pattern** is a named way of encoding something that many problems have in common. It is not a rule. Each pattern is one answer to a question of step O or step D, and it usually has a rival. For each one we say when to use it, show it in a file that runs, and say what it costs.

Static table

Static table: a relation that no action changes, written as facts of the initial state and read by preconditions. The map of the courier world is one (`connected`); so are `ordered` and `before` in the waiter.

When to use it. Whenever a rule of the world could be different in another problem: which rooms are joined, which dish goes first, who is allowed to do what. The test is to ask what would change for a different instance. If the answer is "a line of an action", the knowledge is in the wrong place.

Example. A farmer must take a wolf, a goat and a cabbage across a river in a boat that carries him and one more thing. Left alone, the wolf eats the goat and the goat eats the cabbage. A direct model has an action for each thing, with the dangers written into preconditions. Listing 7.2 has none of that: who eats whom is a static table, `(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))))
```

Listing 7.2: The river puzzle with its rules as data (`river-domain.pddl`).

The preconditions say: after the crossing, no pair in the `eats` table stays together on the bank that the farmer leaves. Kairos finds the seven crossings of the classic solution:

```
$ 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)
```

The second problem adds one fact, `(eats wolf cabbage)`, and has no plan. A new puzzle cost one line of data.

Cost. Almost none at search time: static preconditions are evaluated when the model is grounded and prune ground actions (Chapter 3). The cost is in the problem files, which grow, and in mistakes there, which no type check catches: a missing fact is not an error, it is a different world.

Mirrored facts

Mirrored facts: a symmetric relation written twice in the static table, once in each direction, so that actions read it in one direction only. Every problem of the courier world does this with its map: `(connected lobby dock)` and `(connected dock lobby)`.

When to use it. For any relation that holds both ways (adjacent, connected, next to, compatible), unless the problem files are written by hand and are very large.

Example and rival. The rival writes each pair once and lets the action read both ways:

```
  (: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)))
```

With it the map has seven facts instead of fourteen (`courier-sym-p01.pddl`). The plan is the same and so is the number of ground actions, 30 in both versions.

Cost. Mirrored facts double the table and can be written inconsistently: forget one direction and the robot enters a room that it cannot leave. The `or` keeps the table short and consistent, needs `:disjunctive-preconditions`, and loses something: a one-way door can no longer be expressed. Mirrored facts are the default because they are plain STRIPS and because a symmetric relation often stops being symmetric when the model grows.

At-most-one invariant

An **invariant** is a property that holds in every reachable state. The most common one says that something has one value at a time: the robot is in one room, the waiter in one place, each thing on one bank. PDDL has no way to declare it. It is *kept* by the way the actions are written.

At-most-one invariant: every action that adds a value of the property deletes the old one in the same effect, and the initial state has exactly one. The `move` of the courier is the model: `(not (robot-at ?r ?from))` next to `(robot-at ?r ?to)`, with the old value named in the precondition so that the action knows what to delete.

When to use it. For every property that step O described with the words "the" or "its": *the* position, *its* state, *the* holder.

Example. The planner finds these invariants by itself, and it tells. With `-v`, Kairos reports how it packs facts into state variables:

```
$ 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
```

A **mutex group** is a set of facts of which at most one is true at a time. The eight `robot-at` facts form one, and Kairos stores them as a single variable with eight values. Eighteen facts became three variables.

Cost. None. The invariant is free and makes the search smaller. It is also fragile: one action that forgets its delete breaks it, as Section 7.5 shows. Chapter 16 introduces a construct, the object fluent, that makes the single value part of the language.

Successor chain

Classical PDDL has no numbers, and models still need to count: dishes on a tray, floors of a building, units of fuel. **Successor chain**: the numbers are objects, `n0`, `n1`, `n2`..., a static table `(next ?n ?m)` says that each one is followed by the next, and one fact holds the current count. Adding one moves that fact a step up the chain; removing one moves it a step down.

When to use it. For a small bounded quantity of which only the amount matters.

Example. The tray of the waiter as a counter (Listing 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))))
```

Listing 7.3: A tray that counts (from `waiter-count-domain.pddl`).

The capacity is nowhere in the domain. It is the length of the chain in the problem: `(next n0 n1) (next n1 n2)` is a tray for two. With the count at `n2`, no fact `(next n2 ?m)` exists and `load` cannot be applied: the limit is enforced by the absence of data. The same test problem gives the same 11 steps:

```
$ 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)
...
```

Cost. Two more parameters in every action that counts, and plans that are harder to read. Only "plus one" and "minus one" come for free; adding two quantities needs a table with every sum. Beyond a few tens of values the chain is the wrong tool, and Chapter 8 brings real numbers.

Slots

Slots: the capacity is a set of objects, one for each unit of room, with a fact that says whether it is free. Taking something occupies a free slot; releasing it frees the slot. The waiter of Listing 7.1 is the example: `(empty ?s)` and `(on-tray ?d ?s)`. The two grippers of the robot in the gripper domain are slots too (Figure 7.2).

When to use it. When the units of capacity are different things in the world, or when it matters *where* each item is: a rack with numbered positions, a parking lot, two hands.

Cost. Symmetry. If the slots are interchangeable, every plan exists in as many versions as there are ways of assigning items to slots, and the planner may explore them all. The numbers of the waiter show it. With slots the search for `p02` visits 220 states; with the chain, which does not say where each dish sits, 123.

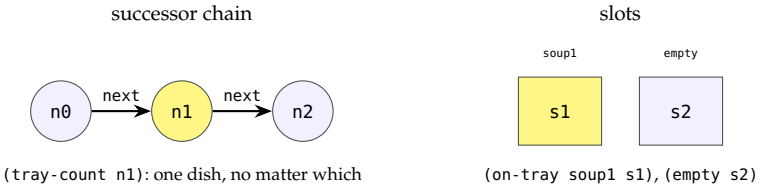


Figure 7.2.: Two encodings of a tray for two that holds one dish. The chain keeps a single fact that moves along `next`; the slots keep one fact for each place, and the same situation has a twin with `soup1` in `s2`.

Chain or slots: count when only the amount matters, use slots when the places are different or it matters which one holds what.

7.3. Patterns for control

A classical plan is a free sequence: any applicable action may come next. Many problems are not that free. Things happen in stages, agents alternate, a tool is in one hand at a time. The three patterns of this section impose such discipline with the same device, a **token**: a fact that says who or what has permission now, that actions require, and that some actions pass on.

Phase token

Phase token: one fact names the current phase of the process. Each action requires the token of its phase, and a special action moves the token to the next phase, never back.

When to use it. When the work has stages that must not mix: all the cutting before any gluing (dust ruins glue), setup before operation, loading before the oven is closed.

Example. A workshop builds a stool from three parts. Each part is cut, joined and painted, and the workshop does all the cutting first, then all the joining, then all the painting.

```
  (: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)
```

Listing 7.4: Phases with a token (from `workshop-domain.pddl`). The phases are constants; `follows` is a static table in the problem.

The preconditions on each part (`join` needs `cut`) already order the work *of one part*. The token orders the work *of the workshop*. A hand-written plan that finishes the seat before touching the legs is rejected by the validator at its second step, with `(now joining) is false`.

Cost. One extra action in the plan for each change of phase, and dead ends: advance too early and a part that was not cut never will be. The benefit is a much smaller search, since the interleavings of the three parts across phases are gone. The token must exist exactly once in the initial state (Section 7.8).

Turn token

Turn token: a fact says whose move it is. Each agent's actions require its turn and hand the turn to the next agent.

When to use it. When agents must alternate, and above all to put into a classical model a world that moves by itself in a known way: after each step of the robot, the conveyor advances, the traffic light changes, the guard walks on. The "other agent" is then the world, with exactly one action available in its turn.

Example. A robot must cross a hall while a cleaning machine goes round it on a fixed route (Figure 7.3). They move in turns, one cell each, and must never share a cell.

```
  ; 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))))
```

Listing 7.5: The robot and the machine take turns (from `patrol-domain.pddl`).

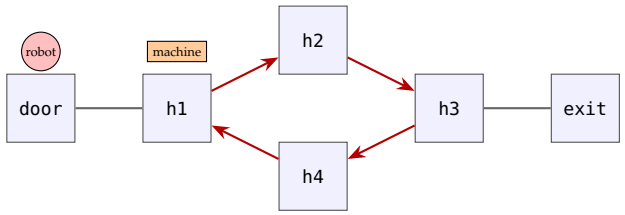


Figure 7.3.: The hall of `patrol-p01`. The red arrows are the round of the machine, which starts in `h1`; the robot starts at the door and wants to reach the exit.

```
$ 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)
```

The robot waits one turn, then follows the machine round the ring, always one cell behind. Two details make the pattern work. The machine has no choice: in its turn exactly one `machine-step` is applicable, so the planner "chooses" it only in name. And if the robot stood in the machine's next cell, no `machine-step` would be applicable, the turn would never come back, and that branch of the search would die. A world step that cannot happen is how this pattern says "forbidden".

Cost. Plans twice as long, half of them steps that nobody executes, and time counted in turns. If the robot and the machine had different speeds, or the world changed by the clock whatever the robot did, turns would be a poor fit; Parts IV and VI have the proper tools.

Resource lock

Resource lock: a thing that only one agent can use at a time is represented by a fact that says it is available. Acquiring the resource deletes that fact, and releasing it adds it back. Nobody can acquire what is not available.

When to use it. For tools, keys, chargers, narrow rooms, the handle of a cart: any resource shared by several agents, or by several activities of one.

Example. The hospital has one keycard for its restricted rooms. It hangs in the lobby.

```
  ; Acquire: the card leaves its hook, so nobody else can
  ; take it.
  (:action take-keycard
    :parameters (?r - robot ?l - location)
    :precondition (and (robot-at ?r ?l) (keycard-at ?l))
    :effect (and (not (keycard-at ?l)) (has-keycard ?r)))

  ; Release: the card is available again, here.
  (:action leave-keycard
    :parameters (?r - robot ?l - location)
    :precondition (and (robot-at ?r ?l) (has-keycard ?r))
    :effect (and (not (has-keycard ?r)) (keycard-at ?l))))
```

Listing 7.6: One keycard for two robots (from `courier-keycard-domain.pddl`). `move` requires that the room is not restricted or that the robot has the card.

The availability fact is `(keycard-at ?l)`. Since `take-keycard` deletes it, the card can never be in two hands. The cheat test asks for exactly that, `(and (has-keycard r1) (has-keycard r2))`, and is unsolvable. In the typical problem, with medicines in the restricted pharmacy and a sample in the restricted ICU, one robot takes the card and does both deliveries in 16 steps: handing the card over would cost more steps than it saves.

Cost. A lock that is taken and never released blocks everybody else for the rest of the plan, and the planner will not release it unless something needs it. When the resource must end up free, say so in the goal. In sequential plans a lock only matters for resources held *across* several actions. With concurrent actions (Chapter 10) it becomes essential.

The cart of Chapter 6 had this flaw: two robots could grab the same handle, because `grab-cart` did not delete any availability fact. Exercise 6.4.3 was a missing lock.

7.4. Patterns for the actions

Inverse pair and one-way action

For each action, step D should ask: can it be undone? The answer is a modeling decision with two named outcomes.

Inverse pair: two actions such that each one restores what the other changed. Picking and dropping an item, opening and closing a door, taking the keycard and leaving it are inverse pairs. **One-way action**: an action without inverse, whose effect is permanent: `seal`, `paint`, `serve`, the `advance` of a phase.

When to use each. Follow the world. If the world has the inverse and leaving it out only saves writing, put it in: a model without it has **dead ends**, states from which the goal cannot be reached, that the world does not have. If the world has no inverse, do not invent one: the planner would use it to escape mistakes that are final.

Example of a one-way action. In Sokoban the player pushes boxes and cannot pull them (Listing 7.7). The grid is a static table with a direction in each fact, so that "three cells in a line" can be said; boxes have no names, because one box is as good as another.

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

Listing 7.7: Sokoban: `walk` is its own inverse, `push` has none (from `sokoban-domain.pddl`).

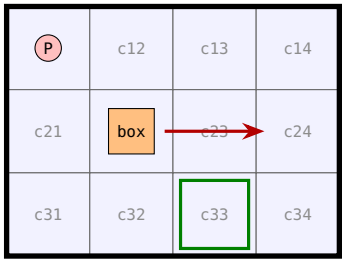


Figure 7.4.: The room of `sokoban-p01`: the player `P`, the box, and the goal cell (green). Two pushes to the east put the box against the right wall, from where no push brings it back to column 3: a dead end.

```
$ 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)
```

In `sokoban-p02` the box starts in a corner, and Kairos explains the failure without searching: `(box-at c22) can never be reached: push adds it, but is never applicable`.

Example of a missing inverse. The waiter cannot put a dish back on the pass. If the tray is full of main courses while the starters wait (`waiter-p05`), nothing can be served and nothing can be loaded: no plan. Is that faithful? A real waiter would put a dish back. The model card assumed otherwise, and here the assumption costs solutions. The repair is the inverse of `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))))
```

An inverse has the effects of the original turned around, adds for deletes and deletes for adds, and for precondition what the original left true. With it the same situation has a plan of 11 steps that begins with `(unload fish1 s1)`.

Cost. Inverse pairs remove dead ends and enlarge the state space: the planner can now go back and forth for ever, and relies on its memory of visited states not to. One-way actions keep the space small and make the order of decisions matter. Heuristics based on the delete relaxation of Chapter 3 do not see most dead ends, so a model full of them is hard for reasons that the number of ground actions does not show.

Merged action

How much of the world is one action? "Deliver the soup" can be one step or five. The **granularity** of a model is that choice. **Merged action**: several things that always happen together, in the same order, with no decision in between, become one action. Its opposite, the *split action*, cuts an action in two where a decision appears.

When to use it. Merge when the planner has nothing to decide between the steps, or when whoever executes the plan decides it better. Split when something can happen in between that matters: another agent's action, a choice of resource, a failure to recover from.

Example. Suppose the trips of the waiter are not the planner's business: the robot's controller batches dishes on the tray by itself. Then the model can merge `load`, `move` and `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))))
```

No tray, no position, four ground actions instead of 22, and a plan of four steps for the problem that took 11. The order of the courses is still enforced.

Cost. A merged action hides what happens inside it. The coarse waiter cannot count trips, so it cannot prefer plans with fewer of them, and it would accept a tray with ten dishes. What a model does not represent, the planner cannot optimize nor forbid. On the other side, every split multiplies the length of plans and the depth of the search. The level to aim for is the one at which the decisions of the story are made.

Model at the level of the decisions that the planner must take. Merge what involves no choice; split where a choice, another agent or a failure can come in between.

7.5. Anti-patterns

An **anti-pattern** is a mistake common enough to deserve a name. The four of this section have something nasty in common: the file is valid PDDL and `kairos check` is satisfied. Each one is caught only by its symptom, so the symptom is what to remember.

Forgotten delete

Forgotten delete: an action adds the new value of a property and does not delete the old one. The at-most-one invariant is broken.

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

The robot of this domain is in every room it has visited. Asked to take the medicines to ward A and return to the dock (`p03`), the correct domain needs 10 steps. This one needs 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)
```

After the pick in the pharmacy the robot leaves *from the lobby*, where it also still is, and it never returns to the dock, because it never left.

Symptoms. A plan that is too short; an action applied from a place where the agent should no longer be. And a sign that needs no plan: the `[sas]` line of `kairos -v` reports `10 variables (1 from 2 mutex groups, 9 binary)` where the correct domain had 3 variables. When a property that should have one value is not found to be a mutex group, some action is not deleting.

Goal in the initial state

Goal in the initial state: the goal is already true before any action. It comes from copying goal lines into `:init`, from a goal that is too weak, or from a negative goal about a fact that was never true.

```
$ kairos waiter-domain.pddl waiter-done-p01.pddl
; cost = 0
```

Symptom. A plan of cost 0 with no steps. Kairos reports success, and it is right: the problem as written is solved. The empty plan is easy to see. The dangerous variant is the goal that is only *partly* true at the start, where the plan looks normal and skips some of the work. Read the goal against the initial state before running anything.

Out-of-step twins

Models often keep two predicates that say the same thing in two ways: `open` and `closed`, `empty` and `on-tray`, `free` and `carrying`. Such twins are legitimate. They avoid negative preconditions and make actions readable. **Out-of-step twins**: an action updates one twin and forgets the other, and the state then says two things that cannot both be true.

The file `waiter-bug-domain.pddl` is the waiter with one fact missing in the effect of `serve`:

```
41      :effect (and (not (on-tray ?d ?s))
42                   (served ?d))))
```

The dish leaves the slot and the slot is not declared `empty`. After two dishes the tray holds nothing and has no free place.

Symptom. The trivial test passes and the typical one has no plan. Finding out why is the subject of Section 7.6.

Prevention. For each pair of twins, write down the invariant ("a slot is empty exactly when no dish is in it") and check every action that touches one of them against it. Or drop a twin and use a negation, at the price of

a requirement. Chapter 13 offers a third way: define one twin as a derived predicate of the other, so that it cannot go out of step.

Idle parameter

Idle parameter: a parameter that no precondition constrains and no effect uses, or whose value does not change what the action does.

```
(:action load
  :parameters (?d - dish ?s - slot ?t - place)
  :precondition (and (waiter-at pass) (ready ?d) (empty ?s))
```

Somebody thought that loading a dish should mention its table. The parameter `?t` is free, so each load exists once for every place: `(load fish1 s2 pass)` and `(load fish1 s2 t1)` are different actions that do the same.

Symptom. Plans with arguments that make no sense, and a count: `kairos -v` reports 38 ground actions for `p02` where the correct domain has 22. With three tables the damage is small. A free parameter of a type with a thousand objects multiplies the action by a thousand, and two of them by a million.

The test for every parameter: is it constrained by a precondition, and does it appear in an effect or decide which effect applies? A parameter may be needed only to look something up in a static table, like `?t` in `serve`, where `(ordered ?d ?t)` leaves one value possible. That one is not idle, and costs nothing after grounding.

7.6. Debugging a model

Step E can end badly in two ways: the planner finds no plan for a problem that has one, or it finds a plan that the world would not accept. A planner is of little help with either. It says "unsolvable" and not which line is to blame; it prints a wrong plan with the same confidence as a right one. A procedure is needed (Figure 7.5).

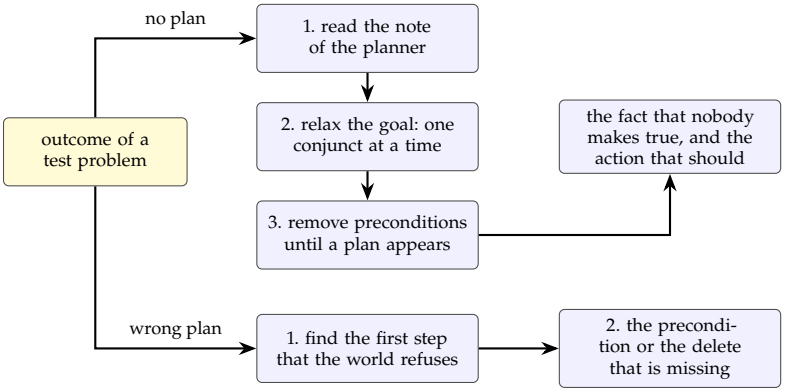


Figure 7.5.: The debugging procedure of step E. Above, for a problem that should have a plan and has none; below, for a plan that should not exist.

No plan

1. Read what the planner says. When the goal needs a fact that no sequence of actions can produce, even ignoring deletes, Kairos says which and why. The mistake of Section 7.8 with a one-way map is diagnosed this

way, without any search. When the note only says that every reachable state was searched, the model can produce each fact of the goal, and something subtler prevents producing them together. Go on.

2. Relax the goal. Replace the goal by one of its conjuncts, then by two, and so on. The aim is the smallest goal that has no plan. For the waiter with the out-of-step twins, `p02` asks for four dishes and fails. We write copies of the problem with weaker goals:

```
$ 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)
```

One dish can be served, and two. Three cannot, although the fish alone raises no difficulty once its soup is served. The model fails at *the third dish*, whichever it is. That points at the tray.

3. Remove preconditions. Take the action that should have applied, here the third `load`, and remove its preconditions one at a time in a copy of the domain, until a plan appears. The precondition whose removal brings the plan back names the fact that the model fails to produce. In `waiter-bug-relaxed-domain.pddl`, `load` has lost `(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)
...
```

The plan is absurd (four dishes in one slot) and that does not matter: it exists, so `(empty ?s)` was the obstacle. The last question is asked of the domain: *which action adds `(empty ?s)`?* None does. It is true only in the initial state. The `serve` that should restore it has lost a line.

No plan: find the smallest goal that fails, then the precondition whose removal brings a plan back, then ask which action should have made that fact true.

A wrong plan

A plan is wrong when the world would not allow it. Read it step by step against the story, not against the PDDL, and stop at the *first* step that the world would refuse. In the plan of the forgotten delete it is the fourth: the robot moves from the lobby while it is in the pharmacy. Then ask why the model allowed that step. There are two possible answers.

- A **precondition is missing**: the action lacks the condition that the story takes for granted. Add it.
- The precondition is there and was true when it should have been false: an **earlier action left a wrong fact**, most often by a forgotten delete. Print the state before the step by hand, from the initial state and the effects, and look for the fact that should not be there.

When the wrong plan comes from a cheat test, turn the repaired case into a regression test: the shortcut written by hand as a plan file, with a `validate` line in `tests.txt` that expects it to be rejected. The test sets of this book are full of them.

7.7. The MODEL method: the classical method, consolidated

Since Chapter 2 every chapter has added questions to the five steps. This section puts the classical part of the method in one place. From here on the book adds numbers, time and preferences to it; the skeleton does not change.

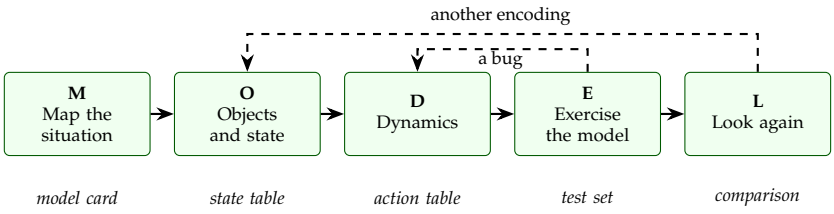


Figure 7.6.: The five steps and the artifact that each one produces. The dashed arrows are the loop: a failed test sends back to the dynamics, and a better alternative to the state.

The classical checklist

M. Map the situation. What is the story? Who acts, and who is only part of the scene? What counts as solved? What is left out, and who takes care of it? Which assumptions make the problem classical (nothing changes by itself, actions are instantaneous and never fail)?

O. Objects and state. Which kinds of things exist; is there a hierarchy; does an object belong to every problem (Chapter 4)? For each property: is it static or dynamic (Chapter 2)? Is it about absence, about any object, about every object (Chapter 5)? Does it have one value at a time? Is it a quantity, and is only the amount relevant? Can it be deduced from another one, and then need not be stored (Chapter 6)?

D. Dynamics. For each action: when does it apply, and what does it change, saying everything that changes and nothing that does not (Chapter 2)? May two parameters be the same object (Chapter 4)? Does what it changes depend on the situation; does it change many objects at once (Chapter 6)? Can it be

undone? Is it one decision or several? Is every parameter needed?

E. Exercise the model. Estimate the size before running (Chapter 3). Run the trivial, the typical, the no-plan and the cheat problems; read and validate every plan; write the cheats by hand and see them rejected. On failure, follow the procedure of Section 7.6.

L. Look again. Which other encodings exist? The pattern catalog suggests them: every pattern has a rival. Compare two or three on Fidelity, Size, Guidance, Clarity and Portability, with numbers, and decide.

The questions in the checklist that carry no chapter reference are those that this chapter adds, and each of them leads to a pattern. Table 7.3 is the index.

Table 7.3.: The pattern catalog of classical planning.

Pattern	Use it when	It costs
Static table	a rule may differ between problems	longer problem files
Mirrored facts	a relation holds both ways	a table twice as long
At-most-one invariant	a property has one value	nothing; one forgotten delete breaks it
Successor chain	a small quantity; only the amount matters	two parameters per action
Slots	units of capacity are distinct places	symmetry among slots
Phase token	stages must not mix	dead ends; an action per stage
Turn token	agents, or the agent and the world, alternate	plans twice as long
Resource lock	one user at a time	the release must be asked for
Inverse pair	the world can undo it	a larger state space
One-way action	the world cannot undo it	dead ends
Merged action	no decision between the steps	what is inside cannot be optimized

Alternatives compared

Step L of the waiter is still open. The patterns give three encodings of the same story, all written and run in this chapter: the tray as **slots** (Listing 7.1), the tray as a **successor chain** (Listing 7.3), and the **merged action** without a tray. Table 7.4 compares them on the typical problem.

Table 7.4.: Three encodings of the waiter on `p02` (four dishes, two tables). The best plan has 11 steps in the first two and 4 in the third.

	Slots	Successor chain	Merged action
Fidelity	exact; knows which dish is where on the tray	exact for the story; the places of the tray are gone	capacity and trips are not represented: any number of dishes "at once"
Size	22 ground actions	22 ground actions	4 ground actions
Guidance	220 states searched; every plan has a twin with the slots swapped	123 states searched; no symmetry	9 states searched
Clarity	plans read well: `(load soup1 s2)`	counts clutter the plan: `(load soup1 n0 n1)`	shortest domain and plans
Portability	universal precondition for the courses	the same	the same

Verdict. The decision depends on a question that belongs to step M: *who decides the trips?*

- If the planner must decide what goes on the tray and in which order the tables are visited, the merged action is out: it cannot represent

the decision. Between the other two, the **successor chain** is the better model of this story. The places of the tray are interchangeable, nothing asks where a dish sits, and the chain searches about half the states. A larger tray needs a longer chain in the problem and no change in the domain.

- **Slots** win as soon as the places differ: a tray with one deep place for soup bowls, or a robot whose controller must be told which arm holds which dish.
- If the robot's own controller batches the dishes, the **merged action** is the right model. It is the smallest, and the planner is left with the only decision that is really its own, the order of the courses.

The model that we wrote first was not the best one for the story as told. That is the normal outcome of step L, and the reason why the method is a loop.

7.8. Common mistakes

The anti-patterns of Section 7.5 are mistakes in the actions. Those of this section come from using a pattern with a piece missing, usually in the problem file.

Half of the mirrored facts

A map written with one fact per connection, for a domain whose `move` reads one direction. In `courier-p02.pddl` the robot is in the dock and the only fact about the dock is `(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)
```

The note walks back from the goal: the item needs a `drop`, which needs the robot in the ward, which needs a `move`, and no move is applicable. **Fix:** write both directions, or generate the map with a script. A milder case, a single missing direction, gives a robot that enters a room and cannot leave: the plans exist and take strange detours.

Two tokens

A token works because it exists once. In `workshop-p02.pddl` the initial state has `(now cutting)`, `(now joining)` and `(now painting)`:

```
$ kairos workshop-domain.pddl workshop-p02.pddl
; cost = 9
(cut seat)
(cut leg1)
(cut leg2)
(join seat)
...
```

Nine steps instead of eleven, and no `advance`: every phase is on, so the phases order nothing. The plan above happens to look tidy; the validator also accepts, for this problem, the plan that finishes the

seat before cutting the legs. **Symptom:** a plan without the actions that pass the token. **Fix:** exactly one token in `:init`; check that every action that adds a token deletes one.

A chain with a broken link

In a successor chain the limit is the *absence* of a `next` fact, so a fact left out by mistake is a limit that nobody intended. A tray declared with `n0 n1 n2` and only `(next n0 n1)` holds one dish. Nothing fails: the waiter makes more trips, and the plan for `p02` grows from 11 steps to 15. **Symptom:** plans longer than expected, with a count that never passes some value. **Fix:** check the chain as carefully as the map, and include in the test set a problem that needs the full capacity.

A lock that nobody returns

With a resource lock, the planner releases the resource only if somebody needs it later. In the plan of `courier-keycard-p01` the robot `r1` ends with the keycard on board, wherever its last delivery left it. If the next shift expects the card on its hook, the model is right and the problem is incomplete. **Fix:** put the final state of every shared resource in the goal, here `(keycard-at lobby)`.

7.9. Summary

This chapter added no syntax. It added vocabulary: names for the choices that every classical model makes.

- A model starts with a **model card**. What it leaves out and what it assumes decide more about the final PDDL than any later step.
- Eleven **patterns** (Table 7.3) answer the recurring questions of steps O and D. Each has a rival and a cost; none is always right.
- Four **anti-patterns** produce valid files and wrong models. Their symptoms: a plan that is too short (forgotten delete), an empty plan (goal in the initial state), a small test that passes while a larger one has no plan (out-of-step twins), too many ground actions (idle parameter).
- **Debugging** is a procedure. No plan: read the note, relax the goal, remove preconditions, ask who should add the missing fact. Wrong plan: find the first step that the world refuses, then the missing precondition or delete.
- The `-v` output is a diagnostic tool as well as a progress report: the number of ground actions and the mutex groups show what the planner understood of the model.
- Step L is not a formality. For the waiter it replaced the first model by a better one, and showed that the choice depends on a question of step M.

Part II ends here. The models of the next part keep everything of this one and add what classical planning cannot count: quantities.

7.10. Further reading

- Haslum et al. (2019) discuss modeling choices throughout their presentation of PDDL, with many of the benchmark domains of

the planning competitions.

- McDermott (2000) reports on the first planning competition, held in 1998, whose domains (gripper and logistics among them) became the common test bed of the field and the source of several patterns of this chapter.
- Helmert (2009) explains how a planner finds mutex groups and turns them into multi-valued state variables: the machinery behind the `[sas]` line that we used to check the at-most-one invariant.
- Ghallab et al. (2004) present classical planning from the side of the algorithms; reading them after this chapter shows why dead ends and symmetries make a model hard.
- Geffner and Bonet (2013) cover the same ground in fewer pages, with the models behind the algorithms.
- Hoffmann and Nebel (2001) is the source of the delete relaxation heuristic that most of the Guidance arguments of this chapter rely on.

7.11. Exercises

Exercises for Section 7.1 — A problem from a story: the waiter

Exercise 7.1.1. Write the model card (Story, Agents, Success, Scope, Assumptions) for this story: *"A coffee machine in an office makes espresso and latte. It needs water, beans and, for latte, milk; each container holds enough for a few cups and somebody must refill it. A cup must be under the spout. People queue and each wants one drink."* List at least three things that you leave out, and say who or what takes care of each.

Exercise 7.1.2. A guest at table 1 orders soup, fish and cake, in that order. Write the problem for the waiter domain of the chapter. Did the domain need any change? How many trips from the pass does the best plan make, and why not fewer?

Exercise 7.1.3. In the waiter domain, nothing in `load` says that the dish will be served later. Is there a cheat here? Consider a goal that asks only for `(served soup1)` in the problem of four dishes, and say what the planner does with the other three dishes and whether the world would object.

Exercises for Section 7.2 — Patterns for the state

Exercise 7.2.1. A farmer has a fox, a chicken, a sack of grain and a dog. The fox eats the chicken and the chicken eats the grain; the dog eats nothing and nobody eats it. Write the problem for the river domain. How many crossings are needed? Then explain what you would have had to change in a domain with one action per animal.

Exercise 7.2.2. The hospital makes the connection between the lobby and the corridor one-way for robots (lobby to corridor only). Which of the two encodings of the map, mirrored facts or the `or` in `move`, can express it, and how? What happens to the delivery of `courier-p03`?

Exercise 7.2.3. Give the waiter a tray for three dishes, first in the slots encoding and then in the successor chain. What changes in each domain and in each problem? Run `p02` with both and compare the number of states searched (`kairos -v`). Explain the trend.

Exercise 7.2.4. For each effect, say whether it keeps the at-most-one invariant of the property it changes, assuming that the invariant

held before: (a) `(and (not (robot-at ?r ?from)) (robot-at ?r ?to))` with `(robot-at ?r ?from)` in the precondition; (b) the same effect without that precondition; (c) `(and (not (now ?a)) (now ?b))` with `(now ?a)` in the precondition and `?a` equal to `?b`; (d) `(carrying ?r ?i)` with `(free ?r)` in the precondition and no delete.

Exercises for Section 7.3 — Patterns for control

Exercise 7.3.1. In the workshop, the token forces all the cutting before any joining. Write by hand a plan for `workshop-p01` that advances to `joining` after cutting only the seat and one leg, and continue it as far as you can. Where does it get stuck, and what does that say about the cost of the pattern?

Exercise 7.3.2. Remove the action `wait` from the patrol domain and solve `patrol-p01` again. Explain the result. Then say what `wait` models in the world, and why forgetting it loses solutions.

Exercise 7.3.3. Some rooms of the hospital are narrow: only one robot fits. Extend the typed courier domain with a resource lock for narrow rooms (a static `(narrow ?l)` and a dynamic fact for the room being taken). Test it with two robots that must each fetch an item from the narrow pharmacy, and with a cheat test that wants both robots inside.

Exercises for Section 7.4 — Patterns for the actions

Exercise 7.4.1. Classify the actions of the courier-with-cart domain of Chapter 6 (`move`, `pick`, `drop`, `load`, `grab-cart`, `release-cart`, `push-cart`) into inverse pairs and one-way actions. For each action, name its inverse or explain why it has none. Does the domain have dead ends?

Exercise 7.4.2. The blocks world of Chapter 2 has four actions that go through the hand: pick up, put down, stack and unstack. Write a version with merged actions, in which a block goes from where it is to where it must be in one step and the hand is not modeled. Solve the reversal of a tower of three blocks with it. What can the four-action model express that this one cannot?

Exercises for Section 7.5 — Anti-patterns

Exercise 7.5.1. The file `gripper-buggy-domain.pddl` models a robot with two grippers that carries balls between two rooms. It passes `kairos check` and solves `gripper-buggy-p01.pddl`. It contains three anti-patterns of this chapter. Find them, first by reading and then with `kairos -v` (ground actions and the `[sas]` line), and write the repaired domain.

Exercise 7.5.2. A student writes this goal for the sterile lab of Chapter 6:

```
(and (not (contaminated sample1))
     (not (item-at sample1 corridor)))
```

The planner answers with a plan of cost 0. Explain what happened and which anti-pattern it is. Why are negative goals especially prone to it?

Exercises for Section 7.6 — Debugging a model

Exercise 7.6.1. The problem `sokoban-p03.pddl` was typed in a hurry: the box in `c22` must be pushed up to `c12`, and Kairos finds no plan. Apply the procedure of the section. What does the note of the planner say, and which facts are missing?

Exercise 7.6.2. In the debugging session of the chapter, step 2 used three relaxed goals. Suppose that instead the goal `(served soup1)` alone had no plan. Which preconditions would you remove first, and in which order? Describe the general strategy for choosing the action and the precondition to relax.

Exercises for Section 7.7 — The MODEL method: the classical method, consolidated

Exercise 7.7.1. Run the whole method on this story and write the five artifacts: *"A ferry carries cars across a river. It holds two cars at most. Cars wait on both shores, each wanting to reach the other side."* Name the patterns that you use. For step L, compare at least two encodings of the capacity of the ferry.

Exercise 7.7.2. Take any domain that you wrote for the exercises of the last three chapters. Go through Table 7.3 and name the patterns that it uses, the rival of each one, and one anti-pattern that it is close to. Change one decision and measure the effect with `kairos -v`.

Part III. Numbers: PDDL 2.1

8. Numeric fluents

The courier of the previous chapters never gets tired. It crosses the hospital floor as many times as the planner asks, because nothing in the model says that moving uses something up. A real robot runs on a battery: every meter costs charge, a long round has to be interrupted to visit the dock, and a robot that forgets it stops in the middle of a corridor. Chapter 7 showed a way to count without numbers, with a chain of level objects linked by `next`. It works for "at most three items" or "five moves left". It does not work for "the corridor is 10 meters long, the ward 12 meters further, and the robot uses a unit and a half of charge per meter": the chain would need a level for every half unit, and the arithmetic would have to be spelled out fact by fact.

This chapter brings numbers into the state. With PDDL 2.1 a model can declare *functions*: properties whose value is a number instead of true or false. Actions can compare those numbers in their preconditions and change them in their effects, with arithmetic in between. The courier world gains a battery that every move drains and the dock restores, distances between rooms, and a load capacity measured in kilograms. Around it, the water jugs, a truck with a fuel tank, a warehouse that counts units and a lab that dilutes samples show the same constructs in other settings.

The chapter starts with what a fact cannot say and how a function says it. Three sections then build the numeric courier: the battery, with comparisons and effects; distances and weights, with arithmetic over data that never changes; and the remaining ways to change a number, with the other domains. Two shorter sections look at what goes wrong when a value was never given, and at why numbers make the planner's job harder, up to the point where no algorithm can do it in general. The method section adds the questions to ask of any quantity and compares numbers with the two ways to do without them.

8.1 What a fact cannot say . 162
8.2 Functions: numbers in the state 163
8.3 Comparing and changing numbers: the battery . . 164
Numeric conditions . . . 164
Numeric effects 165
Running it 166
Exercising the model . . 169
8.4 Numbers that never change: distances, weights and capacity . . 170
A load capacity 170
Another domain: a truck and its fuel 172
8.5 Setting, scaling and asking: jugs, dilutions and stock 173
The water jugs 173
Scaling: a dilution 175
Counting: stock in a warehouse 176
8.6 Values that were never given 177
8.7 Why numbers make planning harder 178
8.8 The MODEL method: quantities 180
Alternatives compared . 181
8.9 Common mistakes 183
8.10 Summary 184
8.11 Further reading 185
8.12 Exercises 185

8.1. What a fact cannot say

Every model so far describes a state as a set of facts. A fact is true or it is not: `(robot-at r1 lab)`, `(open d1)`, `(carrying r1 meds1)`. That is the right tool for properties with a few possible values. The robot is in one of eight rooms; a door is open, closed or locked.

A battery does not fit. Its charge is a **quantity**: a property measured by a number, which takes many values, and on which the world does arithmetic. Moving 12 meters with a consumption of 1.5 units per meter takes 18 units away, whatever the charge was before. To say that with facts we would need one fact per possible charge, and for every pair of charges that differ by 18, a way to go from one to the other. The model would be a table of subtractions written by hand.

Three situations call for a quantity, and the courier world has all of them.

- **A resource that is consumed and replenished.** The battery: moves take charge out, the dock puts it back. Fuel, money, water in a tank and time left behave the same way.
- **A capacity shared by several things.** The robot can carry 5 kg. Two boxes of medicines of 2 kg fit together; a meal of 4 kg fits only alone. What matters is a sum.
- **Data of the problem that are numbers.** The lobby and the corridor are 10 meters apart. Nothing changes that value, but the actions need it to compute how much they consume.

A quantity is a property measured by a number, on which the world does arithmetic: a resource, a capacity, a measure. Facts can only imitate it with one object per value.

Not every number in a story is a quantity for the model. "Ward A has room number 12" is a name. "The robot is on floor 2" is a place among a few. A number becomes a quantity of the model when the planner must add, subtract or compare it to decide what to do. The method section of this chapter turns that remark into questions.

Numbers arrive with PDDL 2.1

The first definition of PDDL already had a requirement called `:fluents` McDermott et al., 1998, but what planners could do with it was not settled. Numbers became part of the common language with PDDL 2.1, defined by Maria Fox and Derek Long for the third International Planning Competition, held in 2002 Fox and Long, 2003. PDDL 2.1 was organized in levels: level 1 was the classical language of the previous chapters, level 2 added numeric fluents and plan metrics, and levels 3 and 4 added time, the subject of Part IV. This chapter and the next one cover level 2.

8.2. Functions: numbers in the state

A predicate, applied to objects, gives a fact that is true or false. A **function**, applied to objects, gives a number. Functions are declared in the domain, in a section of their own, with the same syntax as predicates:

```
(: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
```

The declaration says that every robot has a battery charge, a capacity and a consumption, and that every ordered pair of locations may have a distance. A function applied to objects, such as `(battery r1)` or `(distance lobby corridor)`, is a **numeric fluent**: a variable of the state that holds a number. The word *fluent* comes from the old literature on reasoning about actions, where it names anything whose value may change from one situation to the next.

The parameters of a function are typed like those of a predicate. PDDL 3.1 also lets a function declare the type of its value, `- number`, as in Listing 8.3; when it is left out, `number` is understood.

A state is now two things: the set of facts that are true, as before, and a value for each numeric fluent. The problem gives the initial values in `:init`, next to the initial facts, with an equals sign:

```
  (= (battery r1) 100)
  (= (battery-capacity r1) 100)
  (= (consumption r1) 1)
```

Read `(= (battery r1) 100)` as "the battery of `r1` starts at 100". Values can have decimals (`1.5`) and can be negative. There is no declaration of

units or of a range: that the battery is a percentage between 0 and 100 is our intention, written in a comment, and it is the job of the actions to keep it true.

A function applied to objects is a numeric fluent, a number in the state. The domain declares the functions; the problem gives each fluent its initial value.

A domain that declares functions must say so in its requirements, with `:numeric-fluents`. PDDL 2.1 called this requirement `:fluents`, and many domains that circulate still use the old name; since PDDL 3.1, `:fluents` stands for numeric fluents and object fluents (Chapter 16) together. Kairos accepts both names. If the requirement is missing, it plans all the same and warns:

```
warning: uses numeric-fluents but :requirements does not include :
    numeric-fluents [semantic.undeclared_requirement]
```

Functions, like predicates, come in two kinds that the language does not tell apart but the modeler must (Chapter 2). A **static** function is never changed by any action: the distance between two rooms, the capacity of the battery, the weight of an item. It is data of the problem. A **dynamic** function is changed by some action: the charge of the battery. Only the dynamic ones make the state bigger; the static ones are a table that the planner reads once.

8.3. Comparing and changing numbers: the battery

We start from the typed courier of Chapter 4 and give the robot a battery. The model card changes in three places.

Step M: what the battery changes in the model card

Story. Moving uses battery charge, in proportion to the meters covered. The robot can recharge at the dock, and a recharge fills the battery.
Scope. In: the charge of each robot, the length of each connection, the charger. Out: how long a recharge takes (time arrives in Chapter 10), charge used by picking and dropping, the battery wearing out.
Assumptions. Consumption is the same per meter whatever the robot carries. A robot never starts a move that it cannot finish.

The last assumption is a decision about what a valid plan is. A robot stranded between two rooms is not a state of our model, so the model must forbid the move that would strand it. That is a precondition on a number.

Numeric conditions

A **numeric condition** compares two numeric expressions with one of five operators: <, <=, =, >=, >. Like everything in PDDL the operator goes first. The move of the courier asks for enough charge to cover the stretch:

```
    (>= (battery ?r)
        (* (distance ?from ?to) (consumption ?r))))
```

The second argument is an **arithmetic expression**: a number, a numeric fluent, or an operator applied to two expressions. The operators are `+`, `-`, `*` and `/`, again in prefix form: `(* (distance ?from ?to) (consumption ?r))` is the distance times the consumption. A numeric condition can appear wherever a fact can: under `and`, `or`, `not`, inside a quantifier or a `when`, and in the goal. There is no "different from" operator; `(not (= (load ?r) 0))` does that job, and needs `:negative-preconditions`.

A minus sign with one argument, `(- (load ?r))`, changes the sign. Some planners accept sums and products of more than two arguments; written with two, an expression is accepted everywhere.

Numeric effects

A **numeric effect** changes the value of one numeric fluent. Three operators cover almost every model:

- `(increase f e)` adds the value of the expression `e` to the fluent `f`;
- `(decrease f e)` subtracts it;
- `(assign f e)` replaces the value of `f` with the value of `e`.

The move decreases the battery by what the stretch consumes; the charge assigns the capacity to it. Listing 8.1 is the whole domain. Besides the functions and those two changes it is the typed courier, with one more predicate to say where a charger is.

```
;; 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))))
```

Listing 8.1: The courier with a battery. The move checks the charge and consumes it; the charge fills the battery.

Three details of the listing deserve a second look.

The precondition and the effect of `move` use the same expression. The precondition makes sure that the subtraction will not leave a negative charge. Nothing in PDDL ties one to the other. If the precondition is forgotten, the effect still applies and the battery goes below zero; Section 8.9 shows the plan that comes out.

Expressions are evaluated in the state before the action. All the effects of an action happen at once, and every expression in them reads the values that held when the action was applied. In `move` this makes no difference. It will in the water jugs, where one effect reads a fluent that another effect of the same action sets to zero.

The charge has a numeric precondition too. `(< (battery ?r) (battery-capacity ?r))` forbids charging a full battery. The action would change nothing, and an action that changes nothing is a gift of useless states to the search.

Numeric conditions compare two expressions. Numeric effects change one fluent with increase, decrease or assign. All expressions read the state before the action.

Running it

The first problem (Listing 8.2) is the delivery of Chapter 4 with the numbers added: the distances of the map in both directions, the charger, and the three values of the robot. Figure 8.1 shows the map with them.

```
;; 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)))
```

Listing 8.2: One delivery with a full battery. The distances are part of the initial state.

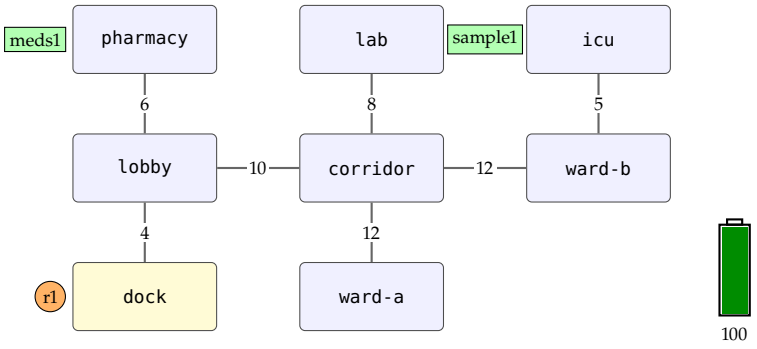


Figure 8.1.: The courier world with its distances in meters. The number on each connection is the value of `distance` in both directions. The robot starts at the dock with a full battery.

We check the files first. The report now lists the feature of this chapter:

```
$ 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)
```

The plan is the one of Chapter 4. The robot covers $4+6+6+10+12=38$ meters and has charge for 100, so the battery never matters. This is the trivial test of step E: it shows that the numbers did not break what worked.

The second problem makes the battery matter. The robot now uses 1.5 units per meter and has two deliveries: the medicines to ward A and a sample from the lab to the ICU. Besides the new object `sample1`, three lines of the problem change:

```
    (= (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)
```

Nobody told the planner to recharge. The goal speaks only of items. The visit to the dock in the middle of the plan is there because, without it, some later move would not be applicable. To read a numeric plan we have to follow the numbers, and Kairos can write them for us: the option `--dump=plan-trace` writes the state after every step to a file.

```
$ 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)
```

The trace lists only what can change; static facts and values are left out.

Figure 8.2 draws the whole trace. After the first delivery the robot is in ward A with 43 units. The sample is 20 meters away and the ICU 25 further: 67.5 units. It cannot be done, so the robot goes back. It reaches the lobby with 10 units and the dock with 4: it could not have gone anywhere else. With a full battery the second delivery takes 70.5 units and the plan ends with 29.5.

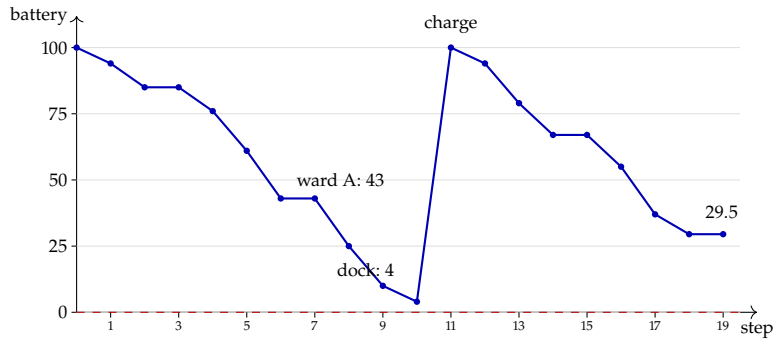


Figure 8.2.: The battery of `r1` along the plan of the second problem. Flat stretches are picks and drops. The robot reaches the dock at step 10 with 4 units; the charge of step 11 pays for the second delivery.

The validator does the same arithmetic. If we delete the line of the charge from the plan and ask Kairos to validate what is left, it points at the first move that the battery cannot pay, with the two numbers that it compared:

```
$ 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
```

Exercising the model

Two problems that work are half of step E. The other half looks for what the model allows and the world does not.

Step E: the test set of the battery

Trivial	`p01`: one delivery, charge to spare. Plan of 7 actions, no charge.
Typical	`p02`: two deliveries. Plan of 19 actions with one visit to the dock, placed where the trace says it must be.
No plan	`p03`: the robot is in the ICU with 20 units and an item for the lab, 25 meters away. There must be no plan.
Cheat	`p04`: the goal is a battery above its capacity, `(> (battery r1) 100)`. There must be no plan.

The third test passes, and Kairos says why it is sure:

```
$ 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)
```

The fourth test is there because of a first draft of this domain. In that draft the charger added a fixed amount:

```
    :precondition (and (robot-at ?r ?l) (has-charger ?l))
    :effect (and (increase (battery ?r) 50)))
```

It looked reasonable and both delivery problems gave sensible plans. The cheat test did not:

```
$ kairos courier-battery-naive-domain.pddl \
    courier-battery-naive-p04.pddl
; cost = 1
(charge r1 dock)
```

The robot started with 60, charged once and had 110 units in a battery of 100. With that domain a planner facing a long round would charge three times in a row at the start and leave with 250. A number has no upper limit unless an action enforces one. The domain of Listing 8.1 enforces it by assigning the capacity. With it, the cheat test has no 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)
```

For every quantity, test its two ends: a problem that needs more than there is, and a goal beyond the limit. Numbers have no bounds of their own; only preconditions and assign keep them.

8.4. Numbers that never change: distances, weights and capacity

Fourteen of the seventeen numbers of the first problem are distances. They are never changed by an action: they are static, and they play for numbers the role that `connected` plays for facts. The verbose output of Kairos shows what the planner makes of them:

```
$ 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)
...
```

One numeric fluent, not seventeen. When the planner grounds the action `move` for `r1`, the lobby and the corridor, it looks up `(distance lobby corridor)` and `(consumption r1)`, finds 10 and 1.5, and keeps a ground action that needs 15 units and subtracts 15. The static numbers disappear into the actions, as static facts did in Chapter 3. Only `(battery r1)` remains as a variable of the state.

Static functions are data: the planner replaces them by their values while grounding. Modeling with distances, weights or prices as static functions costs nothing at search time.

This is why data should be data. We could have written a consumption of 15 into a special move for the pair lobby and corridor, or one `move` action per length of connection. With a static function the domain has one `move`, the map is in the problem, and a different hospital is a different `:init`.

The distances are given in both directions, with two lines per connection. The language has no way to say that a function is symmetric, as it has none for predicates (Chapter 7). A missing direction is a trap that Section 8.6 springs.

A load capacity

The second quantity of the courier is what it carries. Until now the predicate `free` let the robot hold one item. A real courier has a tray with a weight limit. We describe it with three functions and drop `free`: its role is taken by a comparison. The battery is left out of this domain to keep it short; one of the exercises puts both together.

```
;; 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)))))
```

Listing 8.3: The courier with a load capacity. `pick` checks that the item still fits; `pick` and `drop` keep the load up to date.

The precondition of `pick` compares a sum with a capacity: `(<= (+ (load ?r) (weight ?i)) (capacity ?r))`. Of its three fluents, only the load changes. The weight of each item and the capacity of the robot are static, so after grounding the condition for `meds1` reads "the load is at most 3".

The fluent `(load r1)` is redundant in a precise sense: its value is always the sum of the weights of the items that `r1` carries. PDDL cannot compute that sum when it needs it, so the model keeps it in a fluent and updates it in the two actions that change what the robot carries. This is the numeric version of a pattern of Chapter 7, the fact that summarizes others, and it carries the same duty: every action that changes what is summarized must update the summary. An `increase` in `pick` without the `decrease` in `drop` gives a robot whose tray fills up for ever.

In the problem of Listing 8.4 three items wait in the lobby for ward A. They weigh 8 kg and the robot carries 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)
```

Listing 8.4: Three items, 8 kg in total, for a robot that carries 5 kg (the lines of the map are left out).

```
(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)
```

Two trips are needed, and the planner found the only grouping that makes two enough: the two boxes of medicines together (4 kg), the meal alone. With `free` this would have been three trips. If the meal weighs 6 kg instead of 4, no trip can take it, and the problem `courier-load-p02.pddl` has no plan: Kairos searches the 1296 states that can be reached and answers `unsolvable`.

Another domain: a truck and its fuel

The same two ideas, a consumption computed from static data and a limit kept by a precondition, make the oldest numeric example of planning: a vehicle with a fuel tank.

```
;; 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)))
```

Listing 8.5: A truck with a fuel tank. Fuel is bought 10 liters at a time, and only if it fits.

The consumption is a division: kilometers over kilometers per liter. The refueling shows a second way to respect a capacity. The courier's charger used `assign`: whatever was there, now it is full. The station adds a fixed amount, so it needs a precondition that the amount fits: `(<= (+ (fuel ?t) 10) (tank ?t))`. In the problem `truck-p01.pddl` the truck leaves Madrid for Valencia through Cuenca, 170 and 200 km of road, with 5 liters in a tank of 60 and 4 km per liter:

```
$ 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)
```

The first leg burns 42.5 liters, so the truck refuels four times and arrives in Cuenca with 2.5. The second leg burns 50: five more refuelings, 52.5 liters, and 2.5 left in Valencia. A sixth refueling in Cuenca would not have fitted in the tank.

The plan repeats an action many times. A classical plan seldom does that, because doing the same thing twice to the same objects changes nothing the second time. With numbers every repetition changes the state a little, and "how many times" becomes one more thing for the planner to find.

8.5. Setting, scaling and asking: jugs, dilutions and stock

Two effects are still to be seen at work, `assign` with an expression and the pair `scale-up` and `scale-down`, and so are numbers in the goal. Three small domains show them.

The water jugs

There are two jugs without marks, of 3 and 5 liters, a tap and a sink. The task is to measure exactly 4 liters. A jug can be filled to the brim, emptied, or poured into the other until the first is empty or the second is full. Nothing else gives a known amount.

```
;; 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)))))
```

Listing 8.6: The water jugs. Pouring is two actions, because what happens depends on whether everything fits.

Pouring has two outcomes, and the domain has one action for each. `pour-all` applies when the content of the source fits in the target: the target gains it and the source ends empty. `pour-until-full` applies when it does not fit: the target ends full, and the source loses only the room that was left, `(- (volume ?to) (amount ?to))`.

Both actions depend on the rule that expressions read the state before the action. In `pour-all` the first effect adds `(amount ?from)` to the target and the second sets `(amount ?from)` to zero. If effects were applied one after the other, as the statements of a program, swapping the two lines would add zero. In PDDL the order in which effects are written means nothing: both read the old amount.

The puzzle is much older than planning, and a version of it is a scene of the film *Die Hard with a Vengeance* (1995), with jugs of 3 and 5 gallons and a scale that must read 4.

The goal of the problem is a numeric condition, `(= (amount big) 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)
```

Figure 8.3 follows the plan in the space of the two amounts. The state has no facts at all: it is a pair of numbers, and the plan is a path among the pairs that the actions can reach.

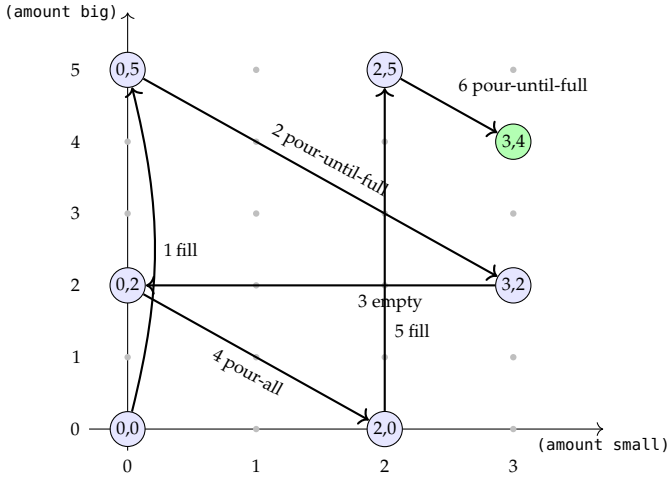


Figure 8.3.: The plan for the jugs of 3 and 5 liters as a path among states. Each state is a pair (small, big). Pouring moves along a diagonal, because the total is kept; filling and emptying move straight to a border.

With jugs of 4 and 6 liters and a goal of 3, every amount that can be reached is even. Kairos searches the 20 states of `jugs-p02.pddl`, finds none with 3 liters, and answers `unsolvable`. The amounts are numbers, but only a handful of values can ever appear, and a search that remembers the states it has seen comes to an end. Section 8.7 shows what happens when that is not so.

Scaling: a dilution

The last two numeric effects multiply and divide: `(scale-up f e)` multiplies the fluent `f` by the value of `e`, and `(scale-down f e)` divides it. They are for changes that are proportions: an interest rate, a population that doubles, a discount. In the lab of the hospital a sample is diluted by adding as much water as there is liquid, which doubles the volume and halves the concentration.

```
;; 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)))
```

Listing 8.7: Diluting a sample: the two scaling effects.

The problem `dilution-p01.pddl` starts with 2 ml at 80 mg/ml in a tube of 10 ml and asks for a concentration of 10 mg/ml or less with at least 4 ml to work with. Its goal is a pair of inequalities, `(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)
```

Two dilutions give 8 ml at 20 mg/ml. A third would need 16 ml and the tube holds 10, so the planner pours half away and dilutes again: 8 ml at 10 mg/ml.

A goal with an inequality describes many states at once: any concentration from 10 down is accepted. Numeric goals are usually of this kind (at least this much charge left, at most this much spent). An equality in a goal is a much narrower target, and with fractions it can be an impossible one (Section 8.9).

`(scale-up f 2)` could also be written `(assign f (* f 2))`, and `(increase f e)` as `(assign f (+ f e))`. The specific operators say more to the planner: an `increase` by a positive constant can only make the fluent grow, and heuristics use that.

Counting: stock in a warehouse

The third domain has no robot and no place. It only counts. A warehouse prepares an order by moving units of several products from the shelf into a box.

```
;; 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))))
```

Listing 8.8: A warehouse that counts. The units of a product are not objects: there is a number of them.

The domain has one type, `product`. The individual boxes of gloves are not objects of the model. They are interchangeable: no action cares which box is taken, only how many are left. Had we made them objects, a shelf with 40 boxes would give 40 ways to pack "a box", all the same to the world and all different to the search. A count removes that symmetry (Chapter 3) before it exists.

`(box-space)` is a function without parameters: a single number for the whole problem. The order of the problem `warehouse-p01.pddl` is its goal, `(and (= (packed gloves) 3) (>= (packed masks) 2))`, with one unit of gloves on the shelf, batches of four, and room for six units in the box:

```
$ kairos warehouse-domain.pddl warehouse-p01.pddl
; cost = 6
(pack masks)
(pack masks)
(pack gloves)
(restock gloves)
(pack gloves)
(pack gloves)
```

The planner packs the only unit of gloves, calls the supplier, which the domain allows only with an empty shelf, and packs two more. If the order asks for four units of masks besides the three of gloves, seven units do not fit in a box for six, and `warehouse-p02.pddl` is reported unsolvable after 66 states.

Indistinguishable things are better counted than named. A count is one fluent; a set of interchangeable objects is a symmetry that the search pays for.

8.6. Values that were never given

With facts, what the initial state does not mention is false: the closed world assumption of Chapter 2. Numbers have no such default. A numeric fluent that `:init` does not mention is not zero. It is **undefined**: it has no value, until some action assigns one.

PDDL 2.1 is strict about what follows Fox and Long, 2003: an action whose precondition or effects need the value of an undefined fluent cannot be applied. The only thing that can be done with an undefined fluent is to `assign` to it. So a forgotten value does not produce an error message. It produces actions that silently never apply.

Take the first battery problem and delete the line that gives `(battery r1)`. The checker finds nothing wrong, because the file is correct PDDL:

```
$ 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)
```

Two states: the initial one and the one where the robot charges, since the charger assigns without reading. No move was ever applicable, because every move compares a battery that has no value. The answer, “unsolvable”, is true of the file and useless about the hospital.

A worse case hides in the map. Delete one of the fourteen distances of the second problem, the one from ward A back to the corridor:

```
$ 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)
```

There is a plan, three actions longer than before and odd to read: the robot leaves the medicines on the floor of the corridor, does the whole delivery of the sample, and comes back for them at the end. The reason is that ward A has become a room with no way out. The fact `(connected ward-a corridor)` is still there, but the move along it needs a distance that has no value, so it never applies. The planner worked around a one-way door that we built by forgetting a line. No message warns of it, because a problem in which some distances are missing is a perfectly legal problem.

A numeric fluent without an initial value is undefined, not zero. Actions that read it never apply, and nothing warns: the symptom is a missing plan or a strange one.

Why not zero?

Defaulting to zero would be simpler and would hide the mistake better. A distance of zero makes a move free; a weight of zero makes an item fit anywhere; a price of zero is a bargain that the planner will find. An undefined value blocks the actions that depend on it, and a model that does too little is easier to notice than one that does too much. The rule also has a use: a fluent can be left undefined on purpose, when an action will measure or set it before anyone reads it.

Planners differ at the edges. Kairos follows the rule in classical and numeric models, and documents some exceptions for PDDL+ models written for other planners (Chapter 17). A model that depends on what happens with undefined values is not portable: give every fluent that is read a value.

Two habits prevent most of these mistakes. The first is to write the initial values next to the facts they belong with, as the load problem does with `(item-at meds1 lobby)` and `(= (weight meds1) 2)` on the same line, so that a missing value is seen as a gap. The second is to add a reachability test to the test set of step E whenever a static function guards an action: a problem whose goal is to visit every room and come back.

8.7. Why numbers make planning harder

A classical problem has a finite number of facts, so a finite number of states: at most two to the power of the number of facts. Chapter 3 showed that this number is huge. But it is finite, and that gives the search a floor to stand on: a planner that remembers the states it has visited will, given time and memory, either find a plan or run out of states and know that there is none.

A numeric fluent can take infinitely many values. Whether it does depends on the actions. The battery cannot: it starts at 100, loses a fixed amount with each move, and returns to 100. The jugs cannot. The truck of Listing 8.5 cannot either, because the tank has a size. Take that precondition away and the fuel can grow without limit:

```
(:action refuel
  :parameters (?t - truck ?c - city)
  :precondition (and (truck-at ?t ?c) (has-station ?c))
  :effect (increase (fuel ?t) 10)))
```

To see the difference we ask both domains for something impossible. Two cities are joined by a road that burns 40 liters, the truck starts with 5, and the goal is to have exactly 12 liters. The tank only ever holds 5 plus tens minus forties, so the goal cannot be reached. With the tank of 60 liters the planner proves it at once:

```
$ 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)
```

Without the limit, there is always one more state to look at:

```
$ 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)
```

A million and a half states in five seconds, each with a little more fuel than another, and no conclusion. The planner cannot answer "unsolvable", because it has not seen every state and never will. It cannot even tell us that more time will not help. Figure 8.4 shows the two state spaces.

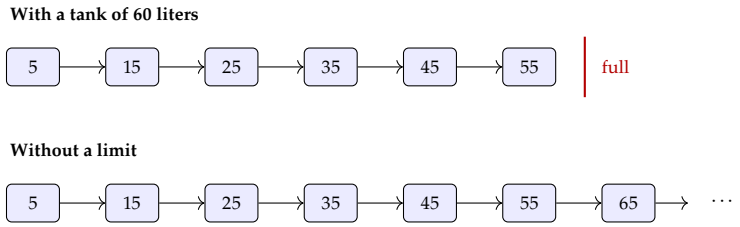


Figure 8.4.: The fuel values that refueling in the first city can reach (the trips to the second city add more states of the same kind). With a tank the chain ends and the search with it. Without a limit there is always a new state.

This is not a weakness of one planner.

Numeric planning is undecidable

For classical planning the question "does this problem have a plan?" can always be answered, at a cost that grows very fast (Chapter 3). For planning with numeric fluents, Malte Helmert proved in 2002 that no algorithm can answer it in every case Helmert, 2002. The reason is that numeric fluents can count without bound, and with two unbounded counters, an action that adds one, an action that subtracts one and a test for zero, a planning domain can imitate any computer program. Asking whether a plan exists is then asking whether a program stops, the classic question that no algorithm settles.

The same paper draws the border: it shows which combinations of comparisons and effects keep the question decidable and which do not, and the undecidable side starts with very little. What this means for a modeler is practical. When the numbers of a model can

only take finitely many values, as with a battery that has a capacity, the planner is on classical ground and "unsolvable" can be proved. When they can grow for ever, a planner may find a plan, but its silence proves nothing.

Unbounded values are one difficulty. The other two appear even in bounded models.

More states. A fact doubles the number of possible states. A numeric fluent multiplies it by the number of values that it can take. The second battery problem has the same rooms and items as a classical one, and every situation of robot and items can occur with many different charges. To prove that its plan of 19 actions is the shortest, Kairos visits several thousand states.

Weaker guidance. Chapter 3 explained the idea behind most heuristics: solve an easier problem in which facts, once true, stay true. Numbers need their own version of that relaxation, and the usual one lets a fluent that can be increased grow as far as needed and forgets that it was ever decreased Hoffmann, 2003. In the relaxed problem the courier never runs out of battery, so the heuristic does not see that a detour to the dock is needed. It finds out late, when the search runs into moves that are not applicable. Resources that run out are exactly what the relaxation ignores.

Numbers can make the state space infinite, and then no planner can prove that there is no plan. Bounded quantities avoid that, but still multiply the states and weaken the heuristics.

For the modeler the lessons are these. Bound every quantity that the world bounds: the tank has a size, the battery a capacity, the tray a limit. Prefer few values to many: a battery that moves in steps of 5 gives the planner fewer states to tell apart than one with two decimals. And keep the option of doing without numbers, which the next section examines.

8.8. The MODEL method: quantities

Numbers enter the MODEL method at step O, when we list what the planner must know about each thing. Until now the answer for each property was a predicate, static or dynamic. Now a property may be a quantity, and three questions decide how to encode it.

Step O: three questions for every property with a number in it

Is it a quantity? Does the model add to it, subtract from it or compare it with another amount? A room number or a priority among three is a name or a level, and a predicate or an object will do. A charge that moves consume and a weight that adds up are quantities.
Does only a threshold matter? If all that the actions ever ask is whether there is enough for one more move, or whether the tray is full, the model may need a fact (`charged`, `full`) and not the number behind it. Keep the number when different actions consume different amounts, or when a goal or a metric speaks of the amount itself.
Is it bounded? What are the smallest and the largest value that the world allows, and which precondition or `assign` keeps the fluent between them? A bound that the model does not enforce is a plan that the world does not allow; an unbounded fluent is a state space without end.

The state table gains a column for the kind of value. For the courier of this chapter:

Property	About	Kind	Encoding and bounds
Charge	a robot	dynamic	`(battery ?r)`, 0 to the capacity; kept by the precondition of `move` and the `assign` of `charge`
Capacity of the battery	a robot	static	`(battery-capacity ?r)`
Consumption	a robot	static	`(consumption ?r)`, per meter
Length	two rooms	static	`(distance ?from ?to)`, both directions
Load	a robot	dynamic	`(load ?r)`, 0 to `(capacity ?r)`; kept by `pick`
Weight	an item	static	`(weight ?i)`

Table 8.1.: The numeric rows of the state table of the courier world. Every dynamic quantity names its bounds and the actions that keep them.

Steps D and E inherit one question each. In step D, for every action and every quantity: does the action need some of it, and does it change it; and if the quantity summarizes something, as the load does, is it updated by every action that changes what it summarizes. In step E, the test set gains the two tests of Section 8.3 for each quantity: a problem that needs more than there is, and a goal on the far side of a bound.

Alternatives compared

Is a number the right encoding for the battery? To compare like with like we simplify it: the battery is counted in moves. Every move uses one unit whatever its length, a full battery has four, and the dock refills it. The delivery from the pharmacy to ward A takes five moves, so the robot must recharge on the way. We wrote this three times.

A, a number. The battery is a numeric fluent. The domain, in the file `moves-number-domain.pddl`, is the one of Listing 8.1 with a simpler move:

```
   :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, a chain of levels. This is the *successor chain* pattern of Chapter 7. The levels are objects `n0` to `n4`, static facts `(next n0 n1)` put them in order, and a predicate says at which level the battery of each robot is. Moving takes the battery from a level to the one below, so the action needs both levels as parameters (`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, one predicate per level. The levels are written into the domain as five predicates, `battery-4` down to `battery-0`, and conditional effects (Chapter 6) step from each one to the next (`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)))
```

Figure 8.5 shows the same state, three units left, in the three encodings.

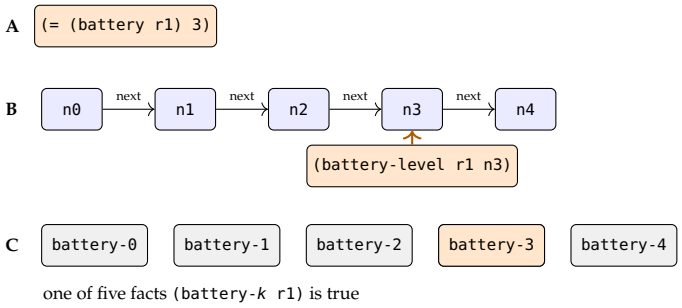


Figure 8.5.: "Three moves left" in the three encodings. A is one number. B is a fact that points at one of five ordered objects. C is one true fact among five predicates.

The three give the same plan of 10 actions for the same problem (the plan of B carries the levels as extra arguments, `(move r1 dock lobby n4 n3)`). They differ in everything else.

	A: number	B: chain	C: flags
Fidelity	Any amount: meters times consumption, fractions, different robots	Whole steps only; a move that uses 15 units needs 15 levels crossed, or one action per amount	As B, and the number of levels is fixed in the domain
Size	31 ground actions whatever the capacity; 1 numeric fluent	76 ground actions with 5 levels, 1516 with 101: every move once per level	31 ground actions, each with one conditional effect per level
Guidance	Finite only if bounded; numeric relaxation is blind to running out	Finite; the levels of a robot form one state variable that classical heuristics handle well	As B
Clarity	Reads like the story	Two extra parameters per action and the `next` facts in every problem	Long; a new capacity means editing the domain
Portability	Needs `:numeric-fluents`: not in classical planners	STRIPS and types (we also used a negated equality in `charge`)	Needs conditional effects and negative preconditions

Table 8.2.: A battery counted in moves, encoded three ways. The sizes are the ground actions that `kairos -v` reports for the files `moves-*-p01.pddl` and, for 101 levels, `moves-chain-p02.pddl`.

Verdict. Use a number when the amounts are data: consumption that depends on distance and on the robot, weights that add up, anything with fractions or with a wide range. That is the courier of this chapter, and it is the only one of the three encodings that survives the step from "one unit per move" back to meters. Use a chain when the quantity is a small count that changes by one (items on a tray, floors, a handful of charges), above all when the planner at hand is classical: the model stays finite and the strongest heuristics apply. Keep one predicate per level for two or three levels with names of their own (empty, low, full), where it is the threshold of step O and no longer a count.

On portability the table needs a longer note. Classical planners such as Fast Downward and LAMA do not read numeric fluents at all Helmert, 2006; Richter and Westphal, 2010. Numeric and temporal planners do, among them Metric-FF Hoffmann, 2003, ENHSP Scala et al., 2016, POPF Coles et al., 2010 and OPTIC Benton et al., 2012. Small differences remain. POPF and OPTIC, for example, reject a numeric condition or effect that

is not wrapped in `(and ...)`: for them the effect of `refuel` must be written

```
    :effect (and (increase (fuel ?t) 10))
```

8.9. Common mistakes

The consumption without the precondition

The mistake. The action subtracts, and nothing checks that there is enough:

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

The symptom. Plans that are too good. The problem `p03`, which had no plan because 20 units do not pay for 25 meters, now has one:

```
$ 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)
```

The trace shows the battery at 15, then 3, then −5. A fluent is a number, and numbers go below zero without complaint.
The fix. Every effect that consumes needs a precondition that there is enough, with the same expression. The "no plan" test of step E is what catches the omission.

A forgotten initial value

The mistake. A fluent that some action reads has no line in `:init`.
The symptom. No error. The actions that read it never apply (Section 8.6): either there is no plan and Kairos reports a handful of states searched, or the plan avoids something for no visible reason. When the forgotten value makes the goal unreachable, the note may point at the action. For a load problem without the weight of the meal:

```
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]
```

The fix. Give a value to every fluent that is read. For static functions over pairs, count the lines: seven connections in two directions are fourteen distances.

The syntax of facts used for numbers

The mistake. Writing an initial value as if it were one more argument of a fact, or writing the assignment of an effect with the equals sign.
The symptom. Both are syntax errors, and the checker points at them:

```
$ 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
```

The fix. The equals sign has two jobs and effects are not one of them. In `:init`, `(= (battery r1) 100)` gives a value. In a condition, `(= (amount big) 4)` is a test. An effect that sets a value is `(assign (battery ?r) 100)`.

Equality on fractions

The mistake. A goal or a precondition that asks a computed fraction to be exactly equal to a decimal. A drip adds 0.1 ml at a time and the goal is `(= (dose) 0.3)`.
The symptom. No plan, for a problem that a child solves:

```
$ 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)
```

Planners compute with the floating-point numbers of the machine, in which 0.1 has no exact representation: three additions of 0.1 give 0.30000000000000004, and that is not 0.3.
The fix. Ask for a range, as `drip-p02.pddl` does with `(and (>= (dose) 0.29) (<= (dose) 0.31))`, which is solved with three drips. Better still, choose units in which the numbers are whole: tenths of a milliliter, cents, percent.

8.10. Summary

PDDL 2.1 adds numbers to the state. A function, declared in `:functions`, gives a numeric fluent for each tuple of objects; the problem gives its initial value. Conditions compare arithmetic expressions and effects change one fluent at a time, always reading the state before the action. Static functions are data that the planner folds into the ground actions; dynamic ones are new variables of the state. The power has a price: a model whose numbers can grow without bound has an infinite state space, and for such models no planner can always tell that there is no plan.

Construct	Meaning
`:numeric-fluents`	the requirement (`:fluents` in PDDL 2.1)
`(:functions (f ?x - t))`	declares a function; `- number` after it is optional
`(= (f a) 5)` in `:init`	initial value; without it the fluent is undefined
`(< e1 e2)`, `<=`, `=`, `>=`, `>`	numeric conditions, in preconditions and goals
`(+ e1 e2)`, `-`, `*`, `/`	arithmetic expressions
`(increase f e)`, `(decrease f e)`	add to, subtract from a fluent
`(assign f e)`	set a fluent
`(scale-up f e)`, `(scale-down f e)`	multiply, divide a fluent

Table 8.3.: The numeric constructs of PDDL 2.1 used in this chapter.

For the method, the chapter adds three questions to step O (is it a quantity, does only a threshold matter, is it bounded), the duty to update every summary fluent in step D, and two tests per quantity in step E. The courier now has a battery, distances and a load. It still treats all plans that reach the goal as equally good: a plan that crosses the hospital twice is as valid as one that does not. Chapter 9 uses the numbers of this chapter to tell a good plan from a bad one.

8.11. Further reading

The definition of numeric fluents, with the precise semantics of conditions, effects and undefined values, is in the article that introduced PDDL 2.1 Fox and Long, 2003; its sections on level 2 are short and readable. The complete syntax, as extended up to PDDL 3.1, is in Kovacs's grammar Kovacs, 2011. The textbook of Haslum and colleagues Haslum et al., 2019 devotes a chapter to numeric planning with more examples of modeling.

The undecidability result, and the map of which numeric features keep planning decidable, is Helmert, 2002. How a heuristic can "ignore the decreases" is explained in the article on Metric-FF Hoffmann, 2003, the planner that extended the relaxation of FF Hoffmann and Nebel, 2001 to numbers; a later and more general relaxation, based on intervals of values, is the one of ENHSP Scala et al., 2016.

8.12. Exercises

Exercises for Section 8.1 — What a fact cannot say

Exercise 8.1.1. For each property, say whether a model needs a quantity or whether facts are enough, and why: (a) the floor on which an elevator is, in a building of six floors; (b) the money in a wallet in a shopping domain; (c) the number of a hospital bed; (d) the temperature of an oven that recipes need "at 180 degrees or more"; (e) whether a printer has paper; (f) the number of sheets left in the printer, when print jobs have different lengths.

Exercises for Section 8.2 — Functions: numbers in the state

Exercise 8.2.1. A delivery drone has an altitude in meters, a payload limit in grams, and flies between sites that are some meters apart. Parcels have a weight. Write the `:functions` section of its domain with typed parameters, and say for each function whether it is static or dynamic.

Exercise 8.2.2. Find the three mistakes in this fragment of a problem for the battery domain, and say what a planner would do with each one:

```
(:init
  (robot-at r1 dock) (free r1)
  (battery r1 80)
  (= (battery-capacity r1) 100)
  (= (distance lobby dock) 4)
  (= (consumption) 1))
```

Exercises for Section 8.3 — Comparing and changing numbers: the battery

Exercise 8.3.1. In the second battery problem the robot starts with a full battery. Before running anything, predict the plan if it starts with 50 units instead: how many times will it charge, and when? Then write the problem and check your prediction.

Exercise 8.3.2. The engineers of the robot ask that the battery never drops below a reserve of 10 units, which is kept for emergencies. Change the domain so that no move leaves the robot under its reserve, with the reserve as a function of the robot. Run the second problem with it. Why does the plan of Section 8.3 stop being valid, and what does the planner do instead?

Exercise 8.3.3. Write a problem like the second one in which the robot must also end with at least half of its charge. Do you need to change the domain? How does the plan change?

Exercises for Section 8.4 — Numbers that never change

Exercise 8.4.1. Put the battery and the load in one domain, and make the consumption depend on the load: a robot uses 2 percent more charge per meter for every kilogram that it carries. Test it with a robot that takes a meal of 5 kg and medicines of 2 kg from the lobby to the ICU, and compute by hand the charge with which it arrives.

Exercise 8.4.2. The first battery problem gives 17 numeric values in its `:init` and `kairos -v` reports "1 numeric fluents". Explain the difference. How many would it report for the load problem of Listing 8.4, and why?

Exercise 8.4.3. Write a problem for the truck domain in which the direct road to the destination is too long for the fuel on board and the only station is in a third city, off the route. Predict the plan and its length.

Exercises for Section 8.5 — Setting, scaling and asking

Exercise 8.5.1. A classic variant of the jugs has no tap and no sink: a jug of 8 liters is full of wine, two jugs of 5 and 3 liters are empty, and the wine must be split in two equal parts. Write a domain without `fill` and `empty` and the problem. How long is the shortest plan?

Exercise 8.5.2. Replace the two pouring actions of the jugs by a single action `pour` with conditional effects (Chapter 6). Which of the two versions do you prefer, and for which of the five criteria?

Exercise 8.5.3. In the dilution domain, `discard-half` requires at least 2 ml in the tube. What could go wrong, for the planner and not for the chemist, if that precondition were removed?

Exercise 8.5.4. Model an elevator that carries people between floors and holds at most a given number of them. Decide what is a fact and what is a number. Test it with three people who go from the ground floor to the second in an elevator for two.

Exercises for Section 8.6 — Values that were never given

Exercise 8.6.1. Delete the weight of the meal from the problem of Listing 8.4. Predict what `kairos check` and `kairos` will say, then try it. What would have happened if undefined values counted as zero?

Exercise 8.6.2. A colleague proposes to avoid forgotten distances by declaring only one direction, `(distance lobby corridor)`, and writing two `move` actions, one that reads `(distance ?from ?to)` and one that reads `(distance ?to ?from)`. Judge the proposal with the five criteria.

Exercises for Section 8.7 — Why numbers make planning harder

Exercise 8.7.1. Which of these models have a finite number of reachable states, whatever the problem? (a) The battery domain of Listing 8.1. (b) The warehouse of Listing 8.8. (c) The warehouse without the precondition of `restock`. (d) The dilution domain. (e) The dilution domain without the precondition of `discard-half`.

Exercise 8.7.2. Run `truck-endless-p03.pddl` with time limits of 1, 2 and 4 seconds and write down the states searched. What do the three answers have in common? Explain to a colleague why "the planner found nothing in an hour" does not mean that the model has a bug.

Exercises for Section 8.8 — The MODEL method: quantities

Exercise 8.8.1. Encode the capacity of the courier without numbers, with the slots pattern of Chapter 7: the robot has two slots and each item takes one. Run the problem of the three items for ward A with it and apply step L against the domain of Listing 8.3: fill a table with the five criteria and give a verdict.

Exercise 8.8.2. In encoding B of Section 8.8 every move costs one level. How would you make the two long connections (12 meters) cost two levels and the rest one, still without numbers? What does it do to the size?

Exercise 8.8.3. *From a story to a model.* A watering robot in a greenhouse carries a tank of 6 liters, which it fills at a tap. Each bed of plants needs a number of liters, and the robot gives water in doses of 2 liters, never more than a bed needs. Apply steps M, O and D (write the changes to the model card, the state table and the action table), write the domain and a problem with two beds that need 6 and 4 liters, and run the four tests of step E.

9. Plan quality: metrics and action costs

A meal waits in the lobby for the desk in the corridor. One robot stands in the pharmacy, 6 meters from the meal; another one is already in the corridor, 10 meters away. Either can do the job with four actions: go to the lobby, pick the meal, go to the corridor, drop it. For every model that we have written so far the two plans are the same plan. Both reach the goal, and a planner asked for "a plan" may return either. For the hospital they are not the same: one robot travels 16 meters and the other 20. Chapter 8 gave the model numbers, and the plan kept the battery within its limits. But nothing in a problem has yet said that less is better.

This chapter adds that sentence. A problem can end with a *metric*: an expression that gives every plan a number, with the instruction to make it as small, or as large, as possible. The simplest metrics add up a cost that each action declares, and PDDL has a small, widely supported feature for exactly that, action costs. More general metrics read any numeric fluent of Chapter 8 and can weigh several of them in one sum. The courier will minimize the meters it covers, then the energy it uses, then a mix of both; a kitchen will cook with the cheapest ingredients; a car will choose between toll highways and free roads.

A metric changes what we ask of the planner, so the chapter also looks at how a planner answers. We watch Kairos print a plan, then a better one, then a better one, and learn what it means when it says that the last one is optimal, and how to tell it that good is enough. The chapter ends with the question that the method adds at its very first step, what makes one plan better than another, and with two comparisons: three ways to say "better", and a limit against a metric.

9.1 When any plan is not enough 188
9.2 Action costs 189
The requirement and its rules 191
9.3 Costs from data: a kitchen with prices 191
Actions that cost nothing 193
9.4 The planner at work: anytime search 193
What `proven optimal` means 195
Good enough 195
9.5 Metrics over any fluent . 196
Maximizing 198
9.6 Several wishes in one number 199
A road trip: tolls and fuel 200
9.7 A limit or a metric 202
9.8 The MODEL method: what makes a plan better 203
Alternatives compared . 203
9.9 Common mistakes 205
9.10 Summary 206
9.11 Further reading 207
9.12 Exercises 207

9.1. When any plan is not enough

The planners of Chapter 3 were described as *satisficing*: they stop at the first plan they find. Which plan that is depends on the order in which the search happens to look at things. Here is the situation of the opening paragraph as a problem for the typed courier, with no numbers at all in what the planner is asked:

```
$ 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)
```

The planner chose `r2`, the robot that walks 20 meters. It is not a mistake. Look at the first line: `; cost = 4`. Since Chapter 2 that comment has headed every plan, and it has always been the number of actions. When a problem says nothing about quality, Kairos takes the **plan length**, the number of actions of a sequential plan, as its cost. By that measure the plan above is as good as a plan can be, and so is the one with `r1` (Figure 9.1).

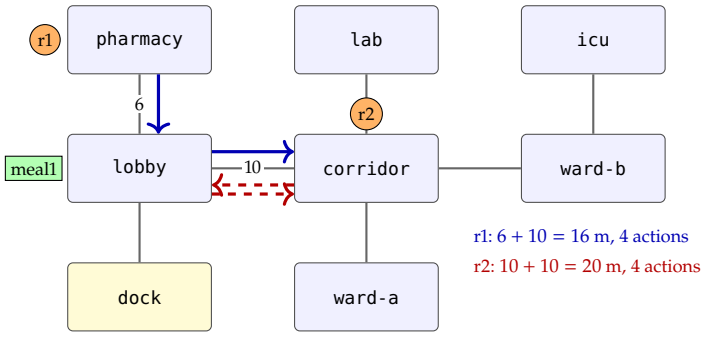


Figure 9.1.: Two plans for the meal. They have the same length, four actions, and differ in the meters covered. Only a model that counts meters can prefer one.

Plan length is a reasonable default. Short plans are easy to read and to check, and for the puzzles of the first chapters, where every move is as good as another, fewer moves is what "better" means. It stops being reasonable as soon as actions differ in what they cost the world: a move of 4 meters and one of 12, an ingredient of one euro and one of five, a toll road and a free one. Then the number of actions measures the length of the text of the plan, and nothing that anybody cares about.

Without a metric, the cost of a sequential plan is its number of actions. That is the right measure only when all actions are worth the same.

Two things are needed to do better. The model has to say what a plan costs, and the planner has to look for cheap plans and not only for plans. The first is the subject of the next sections; the second, of Section 9.4.

9.2. Action costs

The commonest idea of quality is a sum: every action has a cost, and the cost of a plan is the sum of the costs of its actions. Shortest paths in a graph are the oldest example, and the courier that minimizes meters is one more.

PDDL writes this with what we already know. The domain declares a function without parameters, `(total-cost)`; every action that costs something increases it; the problem starts it at zero and asks to minimize it. Listing 9.1 is the typed courier of Chapter 4 with those additions.

```
;; 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

  (: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))))

  ; picking and dropping cover no distance: they cost nothing
```

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

Listing 9.1: The courier with action costs. A move costs its meters; picking and dropping cost nothing.

The cost of `move` is not a constant: it is read from the static function `distance` of Chapter 8, so each ground move has its own cost. The problem adds three things to its `:init` and its end: the distances, the starting value of the cost, and a new section after the goal.

```
    (= (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)))
```

The **metric** is the section `(:metric minimize` *`expression`*`)`. It belongs to the problem, not to the domain: the domain says what each action adds to the cost, and each problem says whether it cares. A plan is evaluated by computing the expression in the state reached at the end of the plan; a plan with a smaller value is a better plan. With the metric, the answer changes:

```
$ 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)
```

Two plans. The first is the one we saw, and its cost is now 20, the value of the metric, not 4. Kairos did not stop there: it kept searching, found the plan with `r1` and printed it after the comment `; better plan 2`. The last plan printed is the best one. Section 9.4 explains this way of working.

An action cost is an increase of total-cost in the effect of an action. The problem starts it at zero and ends with (:metric minimize (total-cost)). The cost printed with each plan is then the value of the metric.

The validator computes the metric too. For the first plan, saved in a file:

```
$ kairos validate courier-cost-domain.pddl courier-cost-p01.pddl \
    courier-cost-p01-r2.plan
plan valid, makespan 4, metric 20
```

(For a sequential plan, what the validator calls *makespan* is the number of steps; the word gets its real meaning in Chapter 10.) A valid plan with a poor metric is still a valid plan. Validation says that a plan works, never that it is good.

The requirement and its rules

The listing declares `:action-costs` and not `:numeric-fluents`. Action costs are a deliberately small piece of numeric planning, defined for the planning competition of 2008 as part of PDDL 3.1 Kovacs, 2011. A domain that declares only `:action-costs` accepts these rules:

- the only numeric fluent that changes is `(total-cost)`, a function without parameters;
- it changes only through `(increase (total-cost) e)`, where `e` is a non-negative number or a static function applied to the parameters of the action, such as `(distance ?from ?to)`;
- no precondition and no goal reads a number;
- the metric is `(:metric minimize (total-cost))`.

Under these rules the cost never influences which actions are applicable. The state space is the classical one, finite, with the same states and the same plans as without costs. The costs only put a weight on each transition. That is why the feature is so widely supported: a classical planner needs no numeric machinery for it, only a search that adds weights. It is also the reason to prefer it when it is enough, as the Portability row of Section 9.8 will say.

Kairos 1.0.6 treats action costs as a case of numeric fluents, and its checker prints a warning for a domain that declares `:action-costs` alone: *uses numeric-fluents but :requirements does not include :numeric-fluents*. The domain is correct PDDL 3.1 and is planned normally. Adding `:numeric-fluents` silences the warning.

Costs enter the competition

Metrics came with PDDL 2.1 in 2002 Fox and Long, 2003, as expressions over numeric fluents, and the competition of 2006 put plan quality at the center of its evaluation A. E. Gerevini et al., 2009. But classical planners, which do not handle numbers, were left out of all that. The competition of 2008 introduced `:action-costs` for them, and from then on its classical tracks scored planners by the cost of their plans and not by their length. The winner of the satisficing track that year was LAMA, by Silvia Richter and Matthias Westphal, which found a first plan quickly and then used the remaining time to look for cheaper ones Richter and Westphal, 2010: the anytime behavior that this chapter shows with Kairos.

9.3. Costs from data: a kitchen with prices

In the courier the choice that the metric decides is who goes. In the next domain it is what to use. A kitchen prepares dishes. A recipe asks for one ingredient of each of some kinds (an egg, a fat, a sweetener) and any ingredient of the kind will do: oil or butter for the fat, sugar or honey for the sweetener. Every ingredient has a price.

```
;; 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))))
```

Listing 9.2: A kitchen with prices. Types say which ingredient can play which role; each `add` action costs the price of what it uses, and cooking costs a fixed amount.

The domain shows the two forms of cost side by side. The actions that use an ingredient cost its price, a static function of the ingredient: `(increase (total-cost) (price ?f))`. The two cooking actions cost a constant, the gas: 1 for frying and 2 for baking. The types of Chapter 4 do the work of the recipes: `add-fat` takes any object of type `fat`, and the planner chooses which.

The pantry of the problem has cheap and dear versions of everything (Listing 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)
```

Listing 9.3: A pantry: each ingredient is an object with its price. The goal is an omelette and a cake.

Both plans have seven actions, and every plan for this problem has seven: two dishes need five ingredients and two cookings. Plan length cannot tell them apart. The first plan takes what comes first, the farm egg, the butter and the honey, and costs 18. The second uses the two plain eggs (2), the two bottles of oil (4) and the sugar (1), plus 3 of gas: 10. No cheaper meal exists with this pantry.

Actions that cost nothing

In the courier, `pick` and `drop` have no `increase`: they cost zero. That is what the story says, since they cover no distance, but zero costs deserve a moment of thought. For a search that only looks at costs, a zero-cost action is free, and a plan with ten useless free actions is exactly as good as the same plan without them. Planners take precautions: Kairos, like LAMA, lets its heuristics see every action as costing one more than it does while it improves a plan, and still compares plans by their real cost. As a modeler, ask whether the free actions are really free. If every handling of an item needs a nurse to come and help, a small cost on `pick` and `drop` says so, and Exercise 9.2.1 shows what it changes.

Costs can be constants or static functions of the parameters. An action without an increase costs zero: make sure that is what the world says.

9.4. The planner at work: anytime search

Finding the best plan is harder than finding a plan. To be sure that a plan is the cheapest, a planner must rule out every cheaper one, and that can mean searching much more of the state space than the first plan needed. An **optimal planner** does that work before it answers, and may not answer at all on a problem that a satisficing planner solves in milliseconds. **Anytime search** is the practical middle way: return a first plan as soon as possible, then keep searching for a strictly better one, and another, until there is none or the time runs out. Whenever it is stopped, the planner has its best plan so far to give.

Kairos works that way by default. After the first plan it restarts its search several times, each time less greedy than the last and each time accepting only plans cheaper than the best one known. A problem with more to

decide shows it well. In Listing 9.4 two robots share a morning round of four deliveries.

```
;; 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)))
```

Listing 9.4: A round of four deliveries for two robots (the map and the distances are left out).

The output has five plans of about thirty actions each. We keep only the lines with the costs:

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    | grep "cost ="
; cost = 188
; cost = 173
; cost = 168
; cost = 164
; cost = 159
```

With `-v`, the progress messages that go to the error output tell when each plan was found, and the status line gives the verdict:

```
$ 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
```

Figure 9.2 draws these numbers. The first plan arrives in 3 milliseconds. Two better ones follow within 14 milliseconds, the cheap improvements. The next two take a third of a second and a full second. Then nothing is printed for three more seconds, until the run ends at 3.9 seconds with the words `proven optimal`.

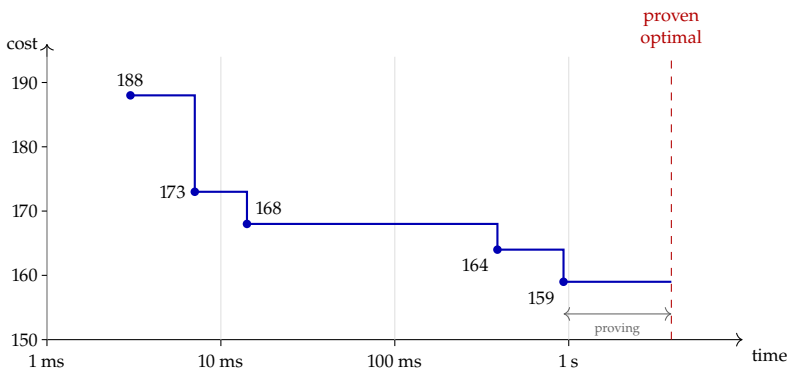


Figure 9.2.: Anytime search on the round of four deliveries: the cost of the best plan known against time, on a logarithmic axis. Most of the run is spent after the last plan was found, proving that no better one exists.

What `proven optimal` means

The search of Kairos can end in several ways, and the status line names the one that happened. `proven optimal` is the strongest: on a sequential task, the last restart exhausted every possibility of a cheaper plan. A plan of 158 meters for this round does not exist. The proof is about the model, and about the metric as written: it says nothing of plans that the domain does not allow, nor of other ideas of "better".

The best plan is worth reading with that in mind. It is saved in the file `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)
```

Every action is of `r1`. The second robot never leaves the ICU. With meters as the only measure this is right: `r2` is far from everything, and each meter that it walks towards the work is a meter added to the sum. A hospital would not be pleased with a robot that is paid for and stands still while the other does all the work. The metric asked for few meters and got them. If the wish was to finish early, the model needs time, which is the subject of Part IV; if it was to spread the wear, it needs another measure. An optimal plan is the best answer to the question that was written, and that is why step M of the method gains a question in this chapter.

Anytime search prints each better plan as it finds it. "Proven optimal" means that no cheaper plan exists in the model for the metric as written. Most of the time may go into that proof.

Good enough

Nearly three of the four seconds of the run went into proving, not into improving. On larger problems the proof may never come, and the run ends at the time limit (30 seconds unless `--timeout` says otherwise) with the best plan found until then. Three options choose how much to wait.

`--no-anytime` asks for the first plan and nothing else. It is the satisficing planner of Chapter 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` says what is good enough: the search stops as soon as it has a plan of cost `C` or less.

```
$ 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
```

The third plan, 168 meters, was the first one within the target, and the run ended there, after 14 milliseconds instead of 4 seconds. A target that cannot be met does no harm: with `--target-cost 100` the same run goes on to `proven optimal` with cost 159.

`--timeout S` gives a budget in seconds. With half a second, on our machine:

```
$ 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
```

In the three cases the exit code is 0 and the status is `solved`: a plan was found. The words in parentheses tell how far the plan can be trusted as the best. Only `proven optimal` is a guarantee. The others say why the search stopped.

When the plans go to files, with `--output round.plan`, each better plan gets its own file, `round.plan.1` to `round.plan.5` here, and the highest number is the best. That is the file to validate or to execute.

Optimal, bounded, satisficing

Planners are usually sorted by what they promise about quality. An *optimal* planner returns only plans of minimal cost, and for that it needs a heuristic that never overestimates and a search such as A* that respects it; it pays with time and memory. A *satisficing* planner promises a plan and nothing about its cost. In between, anytime planners give a sequence of improving plans, and some searches can promise a plan within a factor of the optimum. The competitions have separate tracks for optimal and satisficing planners, with different rules: in one, a plan that is not optimal is a wrong answer; in the other, plans are scored by how close their cost is to the best one known.

For a modeler the difference shows in what can be concluded from a run. From an optimal planner, or from an anytime one that ends with a proof, the plan tells something about the model: *this* is what the metric rewards. From a satisficing run, a strange plan may be the planner's doing and not the model's.

9.5. Metrics over any fluent

Action costs give a plan one number, built in one way. The metrics of PDDL 2.1 are more general: the expression after `minimize` can be any arithmetic expression over the numeric fluents of the problem Fox and

Long, 2003. A domain can keep several measures at once, each in its own fluent, and each problem can choose.

The courier has a second measure waiting since Chapter 8: energy. Robots differ in consumption, so the plan with fewest meters need not be the one that uses least energy. Listing 9.5 keeps two counters.

```
;; 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))))
```

Listing 9.5: The courier with two measures. `move` adds the meters to one fluent and the energy, which depends on the robot, to another.

This domain needs `:numeric-fluents`. It breaks the rules of action costs twice: two fluents change, and the amount added to `(energy-used)` is a product. Neither `(meters)` nor `(energy-used)` appears in a precondition; they are pure counters, there to be read by a metric.

The test problem has two deliveries, the medicines from the pharmacy to ward A and a sample from the ICU to the lab, and two robots: `r1`, light, at the dock, with a consumption of 1, and `r2`, heavy, in ward B, with a consumption of 3. The same problem is written with several metrics. With `(: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)
```

Each robot takes the delivery that is near it: 30 meters for `r2` and 38 for `r1`. With `(:metric minimize (energy-used))` the search goes through three plans:

```
$ 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
```

The first is the plan above: its 30 meters of `r2` cost 90 units of energy, and with the 38 of `r1` the total is 128. In the second, `r1` does both deliveries, one after the other: 92 meters, 92 units. The plans get longer as they get cheaper, from 13 actions to 15 and to 17. Length and cost pull in opposite directions here. The third plan is this one:

```
(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)
```

When we designed the problem we expected 92 to be the optimum. The planner saved 4 units with a trick that we had not thought of: it leaves the medicines on the floor of the corridor on its way to the ICU, and picks them up again after the sample is in the lab, so that it enters ward A only once. The plan is valid in the model. Whether medicines may wait unattended in a corridor is a question for the model card, and if the answer is no, the model must say so: with a precondition, or with the constraints of Chapter 15. An optimizing planner searches the corners of a model far more thoroughly than a satisficing one, and finds what the model forgot to forbid.

A metric can be any expression over numeric fluents. Keep each measure in its own counter fluent and let each problem choose. Optimization explores the loopholes of a model: read the best plan with suspicion.

Maximizing

The metric section accepts `maximize` in place of `minimize`. It reads naturally for wishes such as “end with as much battery as possible”.

With the battery domain of Chapter 8 and one delivery:

```
$ 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]
```

The plan ends with 62 units, and the note is honest about what happened: Kairos could not turn this metric into costs of actions, searched for a short plan, and reports the value of the metric for it. A better plan exists. If the robot goes back to the dock after the delivery and charges, it ends with a full battery, and the validator agrees:

```
$ kairos validate courier-battery-domain.pddl \
    courier-battery-max.pddl courier-battery-max.plan
plan valid, makespan 11, metric 100
```

The reason is in how planners search. Kairos minimizes costs that it derives from the metric: what each action adds to the fluents that the metric weighs. A fluent that only grows, by known amounts, under `minimize`, gives every action a cost, and the search of Section 9.4 applies. The battery goes down with moves and jumps up with an `assign`; "what an action adds" has no fixed answer. Other planners draw the line elsewhere, but all of them draw one. The advice is general: *write metrics as sums of things that only accumulate*. "As much battery as possible" becomes "as little energy used as possible", with a counter as in Listing 9.5.

A `maximize` metric has a second danger, which is in the model and not in the planner. If some action can increase the fluent without limit, there is no best plan: every plan can be improved by repeating that action once more. Section 9.9 comes back to it.

9.6. Several wishes in one number

Meters and energy gave two different plans for the same two deliveries. A hospital that cares about both must say how much it cares about each, and the usual way is a **weighted sum**: each measure is multiplied by a weight that turns it into a common unit, and the metric is the sum.

```
(:metric minimize (+ (* 3 (meters)) (energy-used))))
```

This is the metric of `courier-quality-mix1.pddl`: a meter weighs as much as three units of energy. The best plan is the one of the meters metric, with both robots, at a cost of $3 \times 68 + 128 = 332$. The file `courier-quality-mix2.pddl` weighs a meter as much as one unit, `(+ (meters) (energy-used))`:

```
$ kairos courier-quality-domain.pddl courier-quality-mix2.pddl \
    | grep cost
; cost = 196
; cost = 176
```

The plan with both robots now costs 68 + 128 = 196, and the plan in which `r1` does everything, with its stop in the corridor, costs 88+88 = 176 and wins. Figure 9.3 shows why the weights decide. Each plan is a point in the plane of the two measures. A weighted sum draws parallel lines of equal cost on that plane, and minimizing is sliding the line towards the origin until it touches the last point. The slope of the line, which is the ratio of the weights, chooses the point.

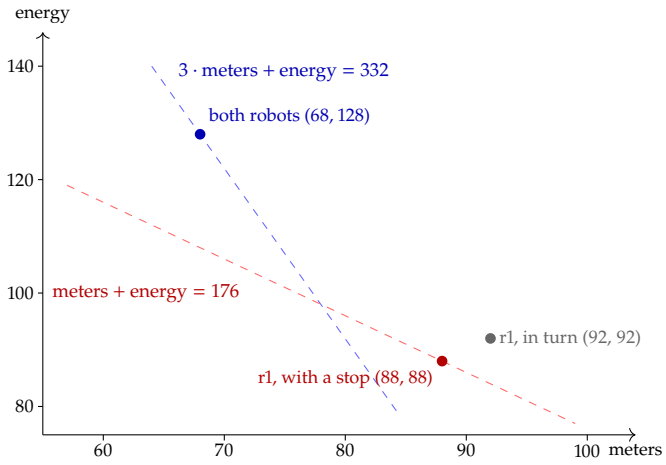


Figure 9.3.: Three plans for the two deliveries as points (meters, energy). The dashed lines are lines of equal cost for the two weighted metrics; each touches a different plan first. The gray plan is worse than the red one in both measures and no choice of weights can prefer it.

The weights are part of the model, and the hardest part to get from the people who know the problem. "Minimize distance and energy" is not a specification. "A meter of travel wears the robot as much as it costs to buy three units of energy" is one. When the measures have a common unit, money above all, the weights are prices and the sum means something.

A road trip: tolls and fuel

A domain where the common unit is obvious: a car goes from one town to another. Between some towns there are two ways, a free road and a shorter highway that charges a toll. The trip costs the tolls plus the fuel.

```
;; 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
      (not (car-at ?c ?from)) (car-at ?c ?to)
      (increase (tolls-paid) (toll ?from ?to))
      (increase (fuel-used)
                (/ (highway-km ?from ?to) (km-per-liter ?c))))))
```

Listing 9.6: A road trip. Each way of driving adds to the fuel used; the highway also adds its toll.

The domain keeps the two amounts in their own units, euros and liters. The price of fuel is not in the domain. It appears in the metric of the problem as the weight that turns liters into euros:

```
; money: the tolls plus the fuel at 2 euros per liter
(:metric minimize (+ (tolls-paid) (* 2 (fuel-used)))))
```

Figure 9.4 shows the network of the problems: four towns, a free road on each of four stretches, a highway on three of them.

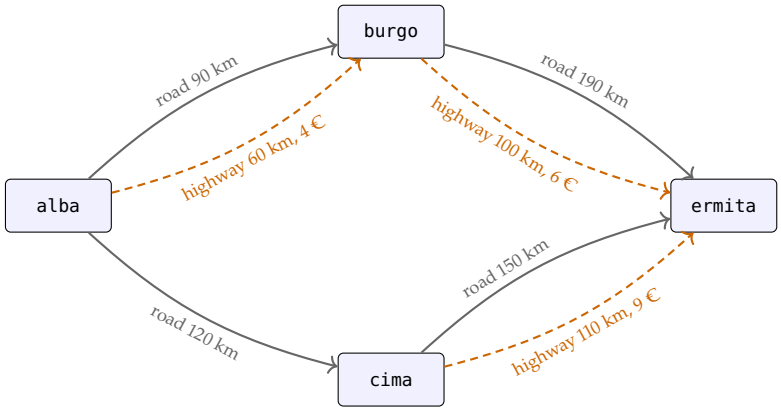


Figure 9.4.: The road network of the trip from Alba to Ermita. Solid arrows are free roads, dashed arrows are highways with their toll. The road from Burgo to Ermita climbs a pass; the highway goes through a tunnel.

A small car does 20 km per liter, so a kilometer costs it 10 cents:

```
$ 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)
```

The best trip uses one highway and not the other. The tunnel saves 90 km, 9 euros of fuel, for a toll of 6. The first highway saves 30 km, 3 euros, for a toll of 4: not worth it. The total is 190 km and 6 euros of toll, 25 euros. In the second problem the same trip is made by a loaded van that does 8 km per liter, 25 cents per kilometer:

```
$ 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)
```

For the van every highway pays for itself: 160 km are 40 euros of fuel, plus 10 of tolls. The map, the actions and the metric are the same as before; one static number changed and the best plan with it. And with `(:metric minimize (tolls-paid))`, in `road-trip-p03.pddl`, the car takes the two free roads through Cima at a cost of 0, and burns 13.5 liters that this metric does not see.

A weighted sum turns several measures into one number. The weights are modeling decisions: with a common unit such as money they are prices, and changing them changes the best plan.

9.7. A limit or a metric

"Spend as little as possible on tolls" and "do not spend more than 5 euros on tolls" sound alike and are different kinds of statement. The first is a metric: it orders the plans and excludes none. The second is a **hard limit**: it splits the plans into allowed and forbidden and says nothing about which allowed plan is better. Chapter 8 gave us the tool for a limit, a precondition on a number. In the highway action:

```
    :precondition (and
      (car-at ?c ?from) (highway ?from ?to)
      (<= (+ (tolls-paid) (toll ?from ?to)) (toll-budget)))
```

The counter `(tolls-paid)` now has two readers: the metric, as before, and this precondition, which compares it with a new static function, the budget. The file is `road-trip-budget-domain.pddl`; the rest of the domain does not change. With a budget of 5 euros and the money metric of before, the small car can no longer use the tunnel:

```
$ kairos road-trip-budget-domain.pddl road-trip-budget-p01.pddl
; cost = 27
(drive-road car1 alba cima)
(drive-road car1 cima ermita)
```

The plan of 25 euros needs a toll of 6, over the budget. Within the limit the best is the free route, 27 euros. The limit made the trip dearer: a limit can only take plans away, so the best plan under a limit is never better, by the metric, than the best one without it. The validator shows both sides of the line. A plan with the first highway alone is allowed and poor, and one with both highways is not a plan of this model:

```
$ 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
```

Limits and metrics combine well, as here: the limit says what is acceptable, the metric chooses among the acceptable. Which one a wish should be depends on what happens when it is not met. If the consequence is a failure (the battery is empty, the card is refused, the tray breaks), it is a limit. If the consequence is only a worse result, it is a metric. A wish stated as a limit for no reason other than habit ("at most 30 meters per delivery") can leave a problem without any plan, where a metric would have returned the best plan that there is.

A hard limit is a precondition: it forbids plans. A metric orders plans and forbids none. Use a limit when crossing it is a failure, a metric when it is only worse.

9.8. The MODEL method: what makes a plan better

Until this chapter the Success entry of the model card described a set of states: the items are where they should be. Any plan that reached them was a success. A metric adds a second half to that entry, and it belongs to step M because it must be asked of the people who own the problem, before a line of PDDL is written.

Step M: two questions about quality

What makes one plan better than another? Name the measure and its unit: meters, euros, units of energy, number of handlings. If the answer is "nothing, any plan will do", say so, and the model needs no metric.
Is it one number? If there are several measures, either they have a common unit and the weights are prices, or somebody must decide how much of one is worth a unit of another. Write the weights in the model card with their reason. And for each wish, ask whether it is a preference (a metric) or a condition of success (a limit).

For the courier of this chapter the card gains two lines. *Success:* every item at its destination; among the plans that achieve it, the fewer meters the better (or the less energy, or the weighted sum, in the later problems). *Assumptions:* a meter costs the same for every robot and at every hour; picking and dropping are free; a robot that stands still costs nothing. Each assumption is a way in which the optimal plan of the model can differ from what the hospital would call the best, and we met the third one in Section 9.4.

The other steps inherit small duties. In step O, every measure of the metric is a counter fluent in the state table, dynamic, starting at zero, read by no precondition unless it is also a limit. In step D, the action table gains a column: what each action adds to each counter, with the zeros written down and justified. In step E, the test set gains a problem with two plans of known cost, to check that the planner prefers the right one, and the cheat test gains a new target: an action whose cost was forgotten.

Alternatives compared

Three ways to tell the planner which plan is better have appeared, and we ran the two deliveries of Section 9.5 with each of them: no metric

at all (`courier-quality-length.pddl`), action costs with the meters (`courier-cost-p03.pddl`, for the domain of Listing 9.1) and a metric over fluents (`courier-quality-energy.pddl` and the two mixes).

	Plan length	**Action costs**	**Fluent metric**
Fidelity	Right only if all actions are worth the same. Here: 13 actions, 68 m, 128 of energy	One sum of fixed costs per action. Here: 68 m, optimal	Any expression: energy that depends on the robot, weighted sums. Here: 88 of energy, or the mixes
Size	92 ground actions, no fluent	92 ground actions, one fluent that no condition reads	92 ground actions, two counter fluents
Guidance	Classical search	Classical state space with weights; the best supported kind of optimization	Good while the counters only grow by fixed amounts; otherwise the planner may not optimize at all
Clarity	Nothing to write, and nothing said	One line per action, one in the problem	Counters to maintain; the metric documents the weights
Portability	Every planner	`:action-costs`: classical planners included	`:numeric-fluents`: numeric planners only

Table 9.1.: Three ways to say which plan is better, on the two deliveries with two robots. All three give 92 ground actions with `kairos -v`; they differ in what the best plan is.

Verdict. Say nothing when nothing matters: puzzles, feasibility questions, the first versions of a model that is still being debugged. Use action costs whenever quality is a sum of fixed costs of actions (distances, prices, risks, effort), which is most of the time; the model stays classical and almost any planner can optimize it. Keep fluent metrics for what action costs cannot say: a cost that depends on the state, as energy that grows with the load carried; several measures to be weighed differently in different problems; or a counter that is also a limit.

On portability, the support table in the documentation of Kairos is the source: among the planners tested there, action costs are accepted by LAMA, POPF and OPTIC as well as Kairos, and metrics over numeric fluents by POPF, OPTIC and ENHSP, but not by LAMA, which has no numeric fluents.

The second comparison is the one of Section 9.7, on the road trip of the small car: the tolls as part of a metric (`road-trip-p01.pddl`), as a hard limit and nothing else (`road-trip-budget-p02.pddl`, the budget without a metric), and as both (`road-trip-budget-p01.pddl`).

	Metric only	**Limit only**	**Limit and metric**
Fidelity	No plan is lost; a plan may spend more on tolls than the driver has. Best: 25 €, toll 6	Never over the budget; any plan within it is accepted, good or bad, and there may be none	Never over the budget, and the cheapest within it. Best: 27 €
Size	Counters only	One numeric precondition per limited action	Both
Guidance	Costs guide the search	The limit makes dead ends: a toll paid early can block the only way later	Both effects
Clarity	The trade-off is in the weights	The rule is where it applies, in the action	Two places to read
Portability	Fluent metric	Numeric preconditions	Both

Table 9.2.: The tolls of the road trip as a metric, as a hard limit, and as both.

Verdict. The two are not rivals: they answer different questions of step M. Use the limit for what must not happen and the metric for what should be small, and expect the limit to cost something in the metric. When in doubt, start with the metric alone and read the best plan. If it does something unacceptable, a limit was missing.

9.9. Common mistakes

An action that forgot its cost

The mistake. One action of a domain with costs has no `increase`. In this kitchen it is `add-fat`:

```
    :effect (and (not (in-stock ?f)) (with-fat ?d)))
```

The symptom. No message: an action without a cost is legal, and free. The planner uses it as much as it can. The best meal now costs 6 instead of 10, and it uses the butter, the dearest fat in the pantry, because for the planner all fats cost nothing:

```
$ 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)
```

The fix. Fill the cost column of the action table for every action, zeros included. Compute by hand the cost of one plan and compare it with the cost that the planner prints.

The metric in the wrong file

The mistake. Writing `(:metric ...)` in the domain, next to the actions that increase the cost.
The symptom. A syntax error:

```
$ 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
```

The fix. The metric goes in the problem, after the goal. The reverse mistake gives no error at all: a domain full of costs with a problem that has no metric is planned by plan length, and the only sign is a cost in the output that looks like a number of actions.

Negative costs and endless improvement

The mistake. A cost that can be negative, such as a refund or a reward, or a `maximize` metric over something that an action can increase for ever.
The symptom. With action costs, negative values break the rules

of the feature, and planners react in their own ways. Kairos stops optimizing and says so. With a pantry where the sugar has a price of −1 (`kitchen-negative.pddl`) it returns the first plan, of cost 18, with the note that we saw for `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]
```

Behind the rule there is a real problem. If a loop of actions has a negative total cost, or a maximized fluent can grow without limit, every plan can be improved by going around once more, and "the best plan" does not exist.
The fix. Keep costs non-negative: model a reward as a smaller cost, or as a cost for *not* doing the thing (the preferences of Chapter 14 do exactly that). Bound what is maximized, as Chapter 8 bounded every quantity.

Trusting the first plan, or the wrong word

The mistake. Reading the first plan of an anytime run as the answer, or reading `solved` as "optimal".
The symptom. Conclusions about the model that are really about the search: "the model sends the far robot", when a later plan does not. In the round of Section 9.4 the first plan costs 188 and the best 159.
The fix. Read the last plan, and the status line with `-v`. Only `proven optimal` says that nothing better exists; `timeout`, `target reached` and `no improvement` say why the search stopped. And remember what the anytime output is: several plans one after the other. Saved to one file and given to the validator as a single plan, it is not valid. Use `--output` and validate the last file.

The initial value of the cost

The mistake. Leaving `(= (total-cost) 0)` out of `:init`.
The symptom. By the rule of Chapter 8, a fluent without an initial value is undefined, and an `increase` of an undefined fluent cannot be applied. Kairos is lenient here and plans the kitchen without that line as if the cost started at zero. A planner that follows the rule would find no applicable action that has a cost.
The fix. Always initialize every counter that the metric reads. It is one line, and the model then means the same everywhere.

9.10. Summary

A metric gives each plan a number and asks for the smallest, or the largest. Without one, the cost of a sequential plan is its length. Action costs are the simple and portable case: a single fluent `(total-cost)` that actions increase by constants or by static data, on a state space that stays classical. Metrics over numeric fluents are the general case: several counters, weighted sums, a choice per problem. Optimizing is harder than solving; an anytime planner prints improving plans, and its last word tells whether the last plan is proved best or only the best found. And a metric is a modeling decision like any other: the optimal plan answers the question as it was written.

Construct or option	Meaning
`:action-costs`	requirement: only `(total-cost)` changes, by non-negative constants or static functions
`(increase (total-cost) e)`	the cost of an action, in its effect
`(= (total-cost) 0)`	in `:init`: every counter starts at zero
`(:metric minimize e)`	in the problem: prefer plans with a smaller final value of `e`
`(:metric maximize e)`	the same for larger values; rewrite as a minimization when possible
`(+ (* w1 f1) (* w2 f2))`	a weighted sum of counters
`kairos D P`	anytime: each better plan is printed; the last is the best
`--no-anytime`	the first plan only
`--target-cost C`	stop at the first plan of cost `C` or less
`--timeout S`	stop after `S` seconds with the best plan found
`-v`	status line: `solved (proven optimal)` and the other stop reasons

Table 9.3.: The constructs of PDDL and the options of Kairos used in this chapter.

The courier now knows how far it goes and what that costs. It still believes that every action happens in an instant and that robots act one after the other, which is why a plan that leaves one of two robots idle could be "optimal". Part IV gives actions a duration, lets them overlap, and adds the measure that this chapter could not write: the time at which the plan ends.

9.11. Further reading

Plan metrics are defined, with numeric fluents, in the article on PDDL 2.1 Fox and Long, 2003. The exact syntax of `:action-costs` and its restrictions are in the grammar of PDDL 3.1 Kovacs, 2011. The report on the competition of 2006 discusses at length how to compare planners by the quality of their plans A. E. Gerevini et al., 2009.

The article on LAMA Richter and Westphal, 2010 is the best description of anytime search with action costs: the restarts with decreasing greediness, and why treating zero-cost actions with care matters. For the search algorithms themselves (A*, weighted A*, and what an admissible heuristic guarantees), any general textbook will do Russell and Norvig, 2020, and the planning view is in Geffner and Bonet, 2013. The modeling of costs and metrics, with more examples, is covered in Haslum et al., 2019.

9.12. Exercises

Exercises for Section 9.1 — When any plan is not enough

Exercise 9.1.1. For each situation say what makes a plan better, in which unit, and whether plan length would be a fair measure of it: (a) the towers of Hanoi; (b) a courier that is paid by the delivery and pays for its own fuel; (c) a robot arm that stacks blocks, where every pick or place takes about the same time; (d) a kitchen that must serve a menu and wants to waste as little food as possible.

Exercise 9.1.2. Run the two versions of the first problem of the chapter, the files `courier-cost-p01-nometric.pddl` and `courier-cost-p01.pddl`. Both outputs start with a line `; cost =`. What does the number mean in each? Why does the first run print one plan and the second two?

Exercises for Section 9.2 — Action costs

Exercise 9.2.1. In the hospital, every pick and every drop needs a nurse to come to the robot. Give each of them a cost of 3 in the domain of Listing 9.1 and run the round of four deliveries again. What is the cost of the best plan now, and how many picks and drops does it have? Compare with the best plan of Section 9.4, which has ten.

Exercise 9.2.2. A student wants loaded moves to be dearer. In a domain whose only numeric requirement is `:action-costs`, the student makes `move` increase the total cost by `(load ?r)`, a fluent that picks and drops change. Which rule of action costs does this break? How would you write the model?

Exercise 9.2.3. Compute by hand the cost of the best plan of the round of four deliveries, saved in `courier-cost-p02.plan` (the distances are in the problem file and in Chapter 8). Check your sum with `kairos validate`.

Exercises for Section 9.3 — Costs from data: a kitchen with prices

Exercise 9.3.1. The butter of the pantry expires today, so using it costs nothing: it would be thrown away otherwise. Change the problem, not the domain, and predict the best meal and its cost before running it.

Exercise 9.3.2. The kitchen domain lets `add-egg` put a second egg into a dish that already has one. Does the planner ever do it when there is a metric? And without a metric? Should the domain forbid it?

Exercises for Section 9.4 — The planner at work: anytime search

Exercise 9.4.1. Run the round of four deliveries with `--no-anytime`, with `--target-cost 165` and with no option, adding `-v` each time. Fill a table with the cost of the plan returned, the stop reason and the total time. Which run would you use in a robot that must start moving within a tenth of a second?

Exercise 9.4.2. The best plan of the round leaves `r2` idle. Give two different changes to the model, either to the domain or to the metric, that would make an optimal plan use both robots, and say what each one assumes about the hospital.

Exercise 9.4.3. What does Kairos answer to `--target-cost 100` on the round of four deliveries, and why is that the only sensible answer?

Exercises for Section 9.5 — Metrics over any fluent

Exercise 9.5.1. Add a third counter to the domain of Listing 9.5: the number of handlings, picks plus drops. With the metric "energy, plus 5 units for each handling", which plan is the best for the two deliveries? Explain why the stop in the corridor disappears.

Exercise 9.5.2. A problem for the load domain of Chapter 8 has the metric `(:metric maximize (load r1))`. Without running anything: what kind of plan does this metric reward? Is there a best plan? What did the author probably want, and how should it be written?

Exercises for Section 9.6 — Several wishes in one number

Exercise 9.6.1. For the two deliveries, with the metric `(+ (meters) (* w (energy-used)))`, at which weight `w` does the best plan change from "both robots" to "`r1` alone"? Compute it from the points of Figure 9.3, then check it with a weight just below and one just above.

Exercise 9.6.2. At 2 euros per liter the small car uses only the tunnel. At which price of fuel does it start to take both highways? Compute it, then check with a problem at 3 euros.

Exercise 9.6.3. The weights of a weighted sum can hide a measure completely. In `(+ (* 1000 (meters)) (energy-used))`, what role does the energy play? When is such a metric useful?

Exercises for Section 9.7 — A limit or a metric

Exercise 9.7.1. Turn the fuel of the road trip into a hard limit: the car starts with a number of liters and there are no stations on the way. Keep the tolls as the metric. With 9 liters, which way does the small car go? And with 7?

Exercise 9.7.2. For each wish of the courier world, say whether you would model it as a limit or as a metric, and why: (a) the battery must not run out; (b) the robot should not carry more than 5 kg; (c) samples should spend little time outside the lab; (d) the robot should avoid the ICU when it can.

Exercises for Section 9.8 — The MODEL method: what makes a plan better

Exercise 9.8.1. Remove the metric from the kitchen problem and run it. Apply step L to the pair "plan length" and "action costs" for this domain: for which of the five criteria do they differ at all?

Exercise 9.8.2. *From a story to a model.* A drone carries a parcel from a depot to a clinic. It can fly over the main square, which is short and crowded, or around by the river, which is long and empty. Each flight uses energy and has a risk level from 0 to 5. The operator says: "I want to save battery, but not at any risk." Apply step M with its two new questions, write the domain and a problem, and find a weight for the risk at which the drone goes around and one at which it crosses the square.

Part IV. Time: PDDL 2.1 and 2.2

10. Durative actions

The plans of the previous chapters are lists: first this action, then that one. Numbers gave them a cost in Chapter 9, and with it a way to prefer one list to another, but a list is still a list. Give the hospital a second robot and ask for two deliveries, medicines to the ICU and a sample to the lab. A planner answers with thirteen actions, one per line, some of `r1` and some of `r2`. Nothing in that answer says that the two robots can work at the same time, nor that walking the twelve meters to ward A takes longer than picking up a box, nor when the medicines will be in the ICU, which is the only thing the nurse who waits for them wants to know.

This chapter gives actions a **duration** and lets them overlap. PDDL 2.1 does it with a new kind of action, the *durative action*, which has a start, an end and an interval in between, and which says at which of the three each condition must hold and each effect happens. A plan is no longer a list but a timetable: every action has a start time and a duration, and the quality we look at is the time at which everything is done.

We begin with what a sequence cannot say, and write a first durative action for the courier. Reading its plan raises two questions, which the next two sections answer: what exactly happens at each instant, and how to say that something must stay true while an action runs. Then durations are computed from distances and speeds, two robots work in parallel, and we force a robot to do one thing at a time. A kitchen and an elevator show the same ideas away from the hospital. The chapter closes, as always, with the method, the mistakes, a summary and exercises.

10.1 What a sequence cannot say 211
10.2 A first durative action . 212
The courier with fixed durations 213
Reading a temporal plan 215
10.3 What happens at each instant 215
Why 10.001 and not 10 . 216
The validator agrees . . 216
10.4 Conditions in time . . . 217
The interval is open . . 217
10.5 Durations that depend on the action 218
10.6 Working in parallel . . 220
10.7 One thing at a time . . 222
10.8 Other domains: a kitchen and an elevator 225
A kitchen: things that work alone 225
An elevator: waiting for people 226
10.9 The MODEL method: time 228
Alternatives compared . 228
10.10 Common mistakes . . . 230
10.11 Summary 232
10.12 Further reading 233
10.13 Exercises 233

10.1. What a sequence cannot say

Figure 10.1 shows the problem of the opening paragraph. Robot `r1` is in the dock and `r2` in the corridor; `meds1` waits in the pharmacy and must go to the ICU, and `sample1` waits in ward A and must go to the lab. With the model of Chapter 4 a planner returns a sequence of thirteen actions: eight for `r1`, which brings the medicines, and five for `r2`, which brings the sample.

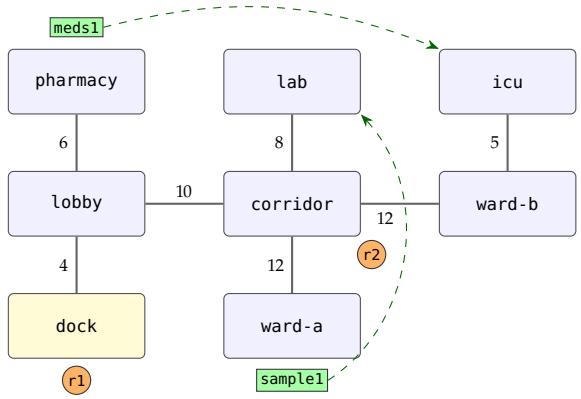


Figure 10.1.: The problem of this chapter: two robots (circles), two items (boxes) and where each item must go (dashed arrows). The numbers are the distances in meters between rooms.

Three things are missing from that sequence.

- **How long each action takes.** Crossing from the corridor to ward B is twelve meters; lifting a box takes a couple of seconds. In the sequence both are one line.
- **What can happen at the same time.** The two robots do not need each other. The sequence interleaves their actions in some order, and any other interleaving would be as good: the order between an action of `r1` and one of `r2` means nothing.
- **When the goal is reached.** If the actions are executed one after another, the work ends when the durations of all thirteen have been added up. If each robot runs its own actions without waiting for the other, it ends much earlier.

Figure 10.2 draws both readings of the same thirteen actions, with the durations that this chapter will give them. Executed as a sequence they take 67 seconds; executed in parallel, 47. This kind of drawing, with time running to the right and one row for each thing that works, is a **timeline** (or Gantt chart), and it is how we will look at every plan from now on.

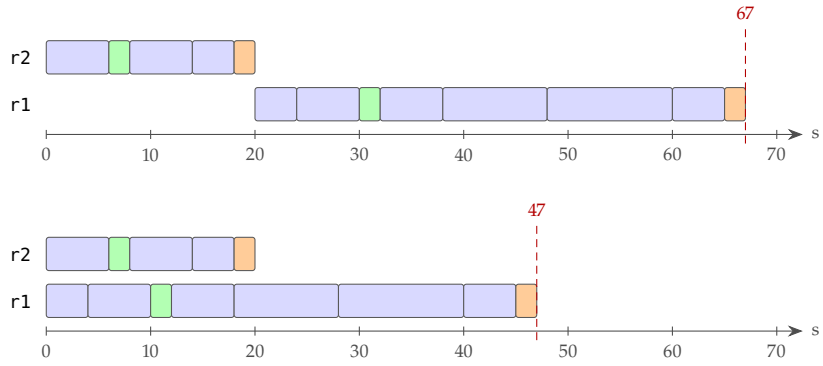


Figure 10.2.: The same thirteen actions executed one after another (top) and with each robot working on its own (bottom). Blue bars are moves, green ones picks, orange ones drops. The dashed line marks the instant at which both items are delivered.

One could try to stay with the tools we have. Make the cost of an action its duration, as in Chapter 9, and minimize the total cost: the planner then prefers short routes. But the total cost is the *sum* of the durations, the 67 of the top drawing, whatever the robots do in parallel. A cost cannot tell the two drawings of Figure 10.2 apart. We will measure exactly how far that approach gets in Section 10.9; to tell them apart, time has to enter the model.

A sequence does not say how long actions take, which of them may overlap, or when the goal is reached. A cost adds durations up; it does not see parallel work.

Time enters PDDL

Durative actions arrived with PDDL 2.1, the language that Maria Fox and Derek Long defined for the third International Planning Competition, held in 2002 (Fox & Long, 2003). PDDL 2.1 was organized in levels: numeric fluents (the subject of Chapter 8), durative actions with discrete effects (this chapter and the next), and durative actions with continuous effects (Chapter 12). The paper is also where the semantics that this chapter explains by example (what happens at an instant, which actions may share it) is defined precisely.

10.2. A first durative action

Take `move` from the typed courier of Chapter 4. As an instantaneous action it says: if the robot is in `?from` and the rooms are connected, then the robot is no longer in `?from` and is in `?to`. A move that takes ten seconds has more to say, because there are now two instants, the one at

which the robot sets off and the one at which it arrives, and the effects are not all at the same one. When the robot sets off it stops being in `?from`. Ten seconds later it is in `?to`. In between it is in neither.

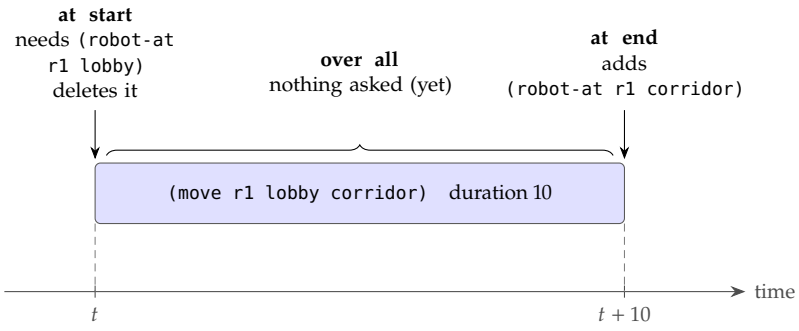
```
(:durative-action move
  :parameters (?r - robot ?from ?to - location)
  :duration (= ?duration 10)
  :condition (and (at start (robot-at ?r ?from))
                  (at start (connected ?from ?to)))
  :effect (and (at start (not (robot-at ?r ?from)))
               (at end (robot-at ?r ?to))))
```

A **durative action** is written with `:durative-action` and has four parts.

- `:parameters`, as in any action.
- `:duration`, a constraint on the special variable `?duration`. Here it fixes it: the action lasts 10. PDDL has no unit of time; we read every duration of the courier world as seconds.
- `:condition`, which takes the place of `:precondition`. The new name is deserved: these conditions are not all checked *before* the action. Each one carries a time specifier: `(at start ...)` for the instant at which the action starts, `(at end ...)` for the instant at which it ends, `(over all ...)` for the interval between the two.
- `:effect`, where each effect is `(at start ...)` or `(at end ...)`. There are no `over all` effects: a discrete change happens at an instant.[1]

Durative actions need `:durative-actions` in the `:requirements`. A domain may mix them with ordinary instantaneous actions; ours will not need to.

1: Change that goes on *during* the action, like a battery that drains second by second, is a continuous effect and belongs to Chapter 12.



A durative action has a duration, conditions at start, over all and at end, and effects at start and at end.

Figure 10.3.: The three moments of a durative action. Conditions can be attached to any of them; effects only to the two ends. During the interval the robot of this `move` is in no room at all.

The courier with fixed durations

Listing 10.1 is the whole typed courier rewritten with durative actions. Every move lasts 10 seconds and every pick and drop lasts 2. The predicates are those of Chapter 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)))))
```

Listing 10.1: The courier with durative actions of fixed duration (`courier-fixed-domain.pddl`).

For `pick` and `drop` we followed the same reasoning as for `move`: what the action takes away goes at the start, what it gives goes at the end. When the pick starts the item stops being on the shelf and the gripper stops being free; when it ends, two seconds later, the robot is carrying the item. When the drop starts the robot is no longer carrying; when it ends the item is in the room and the gripper is free. Section 10.9 shows what goes wrong with any other placement.

The problem is the delivery of Chapter 4, unchanged except for the name of its domain: nothing about time has to be said in a problem. Kairos checks the pair and tells which features it found:

```
$ kairos check courier-fixed-domain.pddl courier-fixed-p01.pddl
valid, plannable
  typing: full
  durative-actions: full (required concurrency included, in engine
     M3)
```

And it plans:

```
$ 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]
```

Reading a temporal plan

A **temporal plan** is a set of lines of the form

```
t: (action) [d]
```

where t is the time at which the action starts and d its duration; the action ends at $t + d$. The lines are sorted by start time, but the order of the lines means nothing else: two actions overlap whenever their intervals do. The clock starts at 0 with the initial state.

The **makespan** of a plan is the time at which its last action ends: here $50.006 + 2 = 52.006$ seconds. It answers the question of the nurse. With no `:metric` in the problem, Kairos takes the makespan as the cost of a temporal plan, which is why both comment lines show the same number.

Now read the plan as a student should read every plan: line by line, asking whether the world would allow it. Two things stand out.

1. The second move starts at 10.001, not at 10. All along the plan a thousandth of a second slips in between one action and the next, and by the end there are six of them.
2. The pick starts at 20.002 and lasts two seconds, but the robot leaves the pharmacy at 20.003. It grabs the box on its way out.

The first is correct and has a precise reason, the subject of Section 10.3. The second is a defect of our model: the planner found a plan that the world does not allow, and it is our fault. Section 10.4 repairs it.

A temporal plan gives each action a start time and a duration. Its makespan is the end of its last action. Read it line by line: the planner uses every freedom the model leaves.

10.3. What happens at each instant

A durative action touches the world twice: once when it starts and once when it ends. In between it changes nothing. So, however long the actions are, a temporal plan changes the world only at a finite number of instants: the starts and the ends of its actions. Each of these instants, with everything that starts or ends at it, is called a **happening**.

To follow a plan, sort its happenings by time and apply them one by one, as if each start and each end were a small instantaneous action. At a start, check the `at start` conditions and apply the `at start` effects. At an end, check the `at end` conditions and apply the `at end` effects. Between two consecutive happenings nothing changes, so it is enough to check there that every `over all` condition of the actions that are running holds. Table 10.1 does it for the first seconds of the plan above.

Time	Happening	State after it (dynamic facts of `r1`)
0	initial state	`(robot-at r1 dock)`, `(free r1)`
0.000	start of `move dock lobby`	`(free r1)`
10.000	end of `move dock lobby`	`(robot-at r1 lobby)`, `(free r1)`
10.001	start of `move lobby pharmacy`	`(free r1)`
20.001	end of `move lobby pharmacy`	`(robot-at r1 pharmacy)`, `(free r1)`
20.002	start of `pick`	`(robot-at r1 pharmacy)`
20.003	start of `move pharmacy lobby`	(none)
22.002	end of `pick`	`(carrying r1 meds1)`

Table 10.1.: The first happenings of the plan of `courier-fixed-p01`. Between 0 and 10 the robot is in no room; at 20.003 it leaves the pharmacy while the pick is still running.

Why 10.001 and not 10

The first move ends at 10.000 and adds `(robot-at r1 lobby)`. The second move needs that very fact at its start. Suppose both happen at exactly 10. Then, at that instant, one thing is making the fact true while another one is relying on it. Which comes first? Inside a single instant there is no "first". An executive that sends both commands to a real robot "at the same time" may deliver them in either order.

PDDL 2.1 settles it with a rule that Fox and Long called *no moving targets*: nothing may rely on a fact at the very instant at which something else changes it. Two happenings **interfere** when one adds or deletes a fact (or changes a number) that the other needs, adds or deletes. Happenings that interfere must be apart by some positive amount of time, written ϵ (epsilon): long enough for the change to be finished and observed before anybody depends on it.

How long ϵ is depends on who executes the plan; the language only asks for it not to be zero. Kairos uses 0.001 unless told otherwise, and that is the thousandth that separates each action of our plan from the next one. Ask for half a second and the plan stretches:

```
$ 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` is a courier that only moves; we will meet it again.) Happenings that do not interfere may share an instant. The two robots of Figure 10.2 both start at 0.000: neither touches anything the other needs.

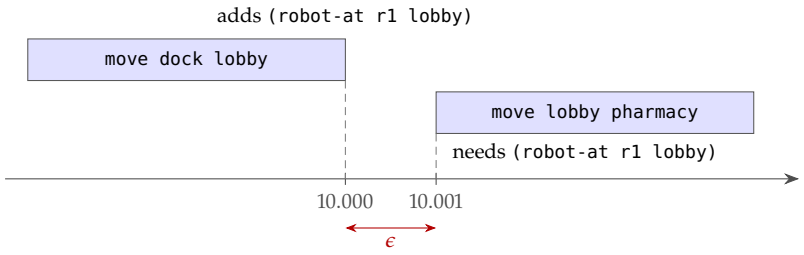


Figure 10.4.: The end of a move and the start of the next one, enlarged. The end changes the fact that the start needs, so the two happenings interfere and must be ϵ apart.

The validator agrees

The plan below is the same delivery for the domain of Section 10.5, written by hand with each action starting at the exact instant at which the previous one ends:

```
$ 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
```

At 4.000 the robot is not yet in the lobby for anybody who needs it there. In the other direction there is all the room one wants: a plan may leave pauses, and `courier-time-p01-slow.plan`, where the robot rests a second or more between actions, is `plan valid, makespan 62`. Nothing in a temporal plan forces an action to start as soon as it can.

A plan changes the world only at its happenings: the starts and ends of its actions. Happenings that interfere must be separated by epsilon (0.001 in Kairos); the others may share an instant.

10.4. Conditions in time

Back to the robot that picks a box while it walks out. Table 10.1 shows that no rule was broken. The pick of Listing 10.1 asks for the robot in the room `at start`, and at 20.002 the robot is there. One millisecond later the robot leaves, and the pick has nothing to say about it: it asked about one instant only.

What we meant is that the robot stays in the room *while* it picks. A condition that must stay true during an action is an **invariant** of the action, and it is written with `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)))
```

With that line (and the same one in `drop`) the plan that cheats is no longer a plan. We copy it to a file, with the durations of the next section, and ask the validator:

```
$ 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
```

The interval is open

Why keep the `at start` line next to the `over all` one? Because `over all` means the interval *strictly between* the start and the end, without the two ends. A condition that must hold from the first instant to the last one is written three times: `at start`, `over all` and `at end`. The three specifiers are independent, and each has its use (Table 10.2).

Specifier	When it is checked	Typical use
`at start`	at the instant t	what the action needs to begin: where the robot is, a free gripper, an item on the shelf
`over all`	in the open interval $(t, t + d)$	what must not change while the action runs: the robot stays, the door stays open, the lift waits
`at end`	at the instant $t + d$	what must be true for the action to complete: somebody is there to receive, the lift is still at the floor

Table 10.2.: The three time specifiers of a condition, for an action that starts at t and lasts d.

The open interval has a visible consequence. In the plans of the repaired model the robot leaves at the very instant at which the pick ends:

```
10.002: (pick r1 meds1 pharmacy) [2.000]
12.002: (move r1 pharmacy lobby) [6.000]
```

The pick ends at 12.002 and the move starts at 12.002, with no ϵ between them. The end of the pick only adds `(carrying r1 meds1)`; the start of the move only touches `(robot-at r1 pharmacy)`, which the pick needed until just before 12.002 and no longer at it. The two happenings do not interfere. If that is too tight for a real gripper, say so in the model: add `(at end (robot-at ?r ?l))` to the pick, and the move will have to wait ϵ. The elevator of Section 10.8 does exactly that.

Step D for a durative action: three questions for each condition

For every condition of an action that takes time, ask:

1. Is it needed to *begin*? Then `at start`.
2. Would the action fail if it stopped being true *halfway*? Then also `over all`.
3. Must it be true at the instant the action *completes*? Then also `at end`.

For `pick`: the item on the shelf is needed only to begin (the action itself takes it away); the robot in the room is needed to begin and all the way; a free gripper only to begin. Facts that never change, like `connected`, need only `at start`: asking for them `over all` is correct but adds nothing.

At start checks one instant. What must stay true while the action runs is an invariant: over all. The interval of over all is open, so the ends are asked for separately.

10.5. Durations that depend on the action

Ten seconds for every move was a placeholder. The floor has distances (Chapter 8 gave them to the courier as the static function `distance`) and a robot has a speed, so a move lasts the distance divided by the speed. The right-hand side of a duration constraint can be any numeric expression over the parameters of the action:

```
  (:durative-action move
    :parameters (?r - robot ?from ?to - location)
    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
```

The new function `(speed ?r)` is in meters per second, and `(handling-time ?r)` is the number of seconds that the robot needs to pick or to drop. Both are static: the problem sets them in `:init` and no action changes them. Because the domain now has `:functions`, it declares `:numeric-fluents` too. Listing 10.2 is the complete domain, with the invariants of the previous section.

```
;; 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)))))
```

Listing 10.2: The courier in time: computed durations and invariants (`courier-time-domain.pddl`).

The problem gives the numbers. These are the lines that the first problem of the new domain adds to the delivery of Chapter 4; every distance is given in both directions, like every connection:

```
    (= (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)))
```

The last line is new. `(total-time)` is a name that PDDL 2.1 reserves for the makespan of the plan, to be used in the `:metric` of a temporal problem. `(:metric minimize (total-time))` asks for the plan that finishes earliest. It is what Kairos minimizes anyway when a temporal problem has no metric, but other planners do not, and a reader of the problem should not have to guess: we write it in every problem from now on.

```
$ 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]
```

Each move now lasts its distance: 4 seconds from the dock to the lobby, 12 from the corridor to ward A. The plan is honest (the robot waits in the pharmacy until the pick is over) and its makespan is the sum of the seven durations, 42, plus five separations of a millisecond.

A plan must use the duration that the constraint gives. The validator computes it again from the state in which the action starts:

```
$ 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
```

"From the state in which the action starts" matters when the expression contains a fluent that changes. Here everything is static, so each ground move has one duration forever. A duration may also depend on dynamic numbers (a refill that lasts longer the emptier the tank is); the planner then evaluates the expression at the start of each action. And the duration may be left for the planner to choose between bounds, which is a subject of Chapter 11.

A duration can be any numeric expression of the parameters: distance over speed. It is evaluated when the action starts. The metric total-time is the makespan.

10.6. Working in parallel

Nothing new is needed for two robots to work at once. In a temporal plan, actions overlap unless something in the model keeps them apart: an interference between their happenings, or a condition that one of them destroys. We only have to give the planner the problem of Figure 10.1 and see what it does. First with `r1` alone (`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]
```

As in Chapter 9, Kairos keeps searching after the first plan and prints each better one. The first takes the medicines all the way to the ICU, comes back for the sample and needs 100 seconds. The second is cleverer than it looks: the robot leaves the medicines on the corridor floor on its way, deals with the sample, and picks the medicines up again when it passes by. Two extra actions, five seconds less. Whether a hospital wants

medicines left in a corridor is a matter for step M, and for the constraints of later chapters; the model as written allows it.

Now with the second robot (`courier-time-p03.pddl`), which starts in the corridor and moves at 2 meters per second. The first plan found is the one drawn at the bottom of Figure 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]
```

The lines of the two robots are interleaved by start time, and to follow one robot we have to skip the lines of the other: this is where a timeline starts to pay. Each robot makes one delivery. `r2` is done at 20.003 and then waits; the makespan, 47.006, is the time of `r1`, which has the long trip. The twelve meters to ward A take `r2` six seconds: the same action schema, another duration.

Without `-no-anytime` the search goes on and, after about a second, ends with this 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]
```

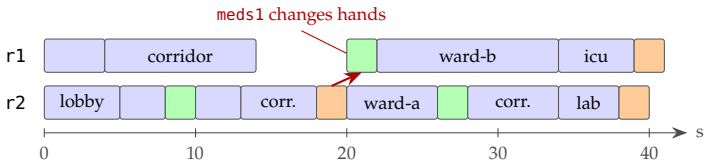


Figure 10.5.: The best plan found for two robots (makespan 41.007). Moves (blue) are labeled with their destination; green is a pick, orange a drop. The fast robot `r2` fetches the medicines and drops them in the corridor at 18; `r1`, which has been waiting there since 14, picks them at 20 and carries them to the ICU while `r2` deals with the sample.

Figure 10.5 draws it. The planner has found a relay that we had not thought of: the fast robot does the long detour to the pharmacy, hands the medicines over in the corridor (a drop and a pick, since this courier has no other way), and both finish almost together. Six seconds are gained over one delivery each.

Three remarks before going on.

The makespan is not the work. The plan of 47 seconds has 67 robot-seconds of actions in it; the plan of 41 has 75. The better plan makes the robots work *more* in order to finish *earlier*. When energy matters more than the clock, the metric must say so; Chapter 9 showed how to weigh several quantities, and `(total-time)` can be one of the terms.

No proof of optimality. After the last plan, Kairos reports `solved (exhausted)`: its searches ended. For sequential plans it can often report `proven optimal`; for the makespan of a temporal plan it does not claim it. Temporal planners in general return good timetables, not provably shortest ones.

Pauses are part of plans. In Figure 10.5 `r1` stands in the corridor from 14 to 20. No action says "wait": a robot waits when no action of its own is running.

Actions overlap unless the model keeps them apart. The planner exploits it: with two robots the makespan drops from 95 to 47, and to 41 with a relay nobody asked for.

10.7. One thing at a time

"Actions overlap unless the model keeps them apart" cuts both ways. Two robots in two rooms should overlap. One robot should not do two things at once, and so far nothing says so in general. With three actions we were lucky: a pick and a drop cannot overlap because of the gripper (`free`), and a move cannot overlap a pick because of the invariant on `robot-at`. Each pair is kept apart by a different accident.

Add a fourth action and the luck runs out. The hospital fits the robots with an ultraviolet lamp to disinfect rooms: the robot must stay in the room, lamp on, for 20 seconds.

```
;; 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))))
```

The domain `courier-uv` is `courier-time` plus this action. We ask the robot to disinfect the pharmacy and to take the medicines to the lobby:

```
$ 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]
```

The robot picks the box while it disinfects. Both actions want the robot in the pharmacy, and it is; neither destroys what the other needs. The invariant we wrote is satisfied to the letter. What is missing is a fact about the robot itself: it is a single machine and its actions take all of it.

The standard remedy treats the robot as a **resource** that an action takes and gives back. A predicate `(idle ?r)` says that the robot is doing nothing. Every action of the robot needs it at start, deletes it at start, and adds it again at end:

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

While an action of `r1` runs, `(idle r1)` is false, and no other action of `r1` can start. Listing 10.3 applies it to the four actions; the problem adds `(idle r1)` to its initial state.

```
;; 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]
```

Listing 10.3: The courier that does one thing at a time (`courier-idle-domain.pddl`).

One action after another, and thirty seconds, as the world demands. Look at the times: every action now starts ϵ after the previous one ends, even the move after the disinfection, which before started at the same instant. The end of each action adds `(idle r1)` and the start of the next one needs it, so the two happenings interfere. A token passed from action to action costs a millisecond per pass.

This is the pattern *resource lock* of Chapter 7, with one difference. For instantaneous actions a sequential plan gives mutual exclusion for free, and a token is needed only to reserve something *across* several actions. With durative actions the token is needed for the duration of a single action. The same pattern protects anything that serves one user at a time: a narrow door, a charger, a pharmacy counter, a burner. The token then belongs to the door or the burner, not to the robot (the kitchen of the next section has one).

Do the invariants of Section 10.4 still earn their place once every action holds `idle`? In this domain they are implied: a robot that is not idle cannot start a move. We keep them because they say something different. The token says that the *robot* does one thing at a time. The invariant says what the *action* relies on in the world, and it will keep protecting the

action when something other than the robot itself can break it: another robot that moves the shelf, a door that closes, a room that shuts at eight.

A robot is a resource: every action takes the token idle at its start and returns it at its end. Invariants say what an action relies on; the token says the agent is one.

10.8. Other domains: a kitchen and an elevator

A kitchen: things that work alone

In the courier world every action belongs to a robot. In a kitchen some of the work is done by nobody: once the pot is on the burner, the soup cooks by itself and the cook is free for something else. A good timetable prepares the next dish while the previous one cooks. Listing 10.4 models it with three actions. `prepare` and `serve` take the cook, through the token `available`; `cook` takes only a burner, through `burner-free`, and has no cook among its parameters. Durations come from the dish and are read as minutes.

```
;; 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)))))
```

A soup (6 minutes to prepare, 20 to cook), pasta (1 and 10) and a sauce (4 and 8), one cook and one burner:

Listing 10.4: A kitchen (`kitchen-domain.pddl`).

```
$ 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]
```

The burner never rests: the three cooking times add up to 38 minutes, and before the first one something must be prepared (at least 1 minute, the pasta) and after the last one something served (1 minute). No timetable with one burner can beat 40, and the planner reached it on its fourth try (the first took 46). The cook works in the gaps. With a second burner (`kitchen-p02.pddl`) the makespan drops to 28.003; Figure 10.6 draws both.

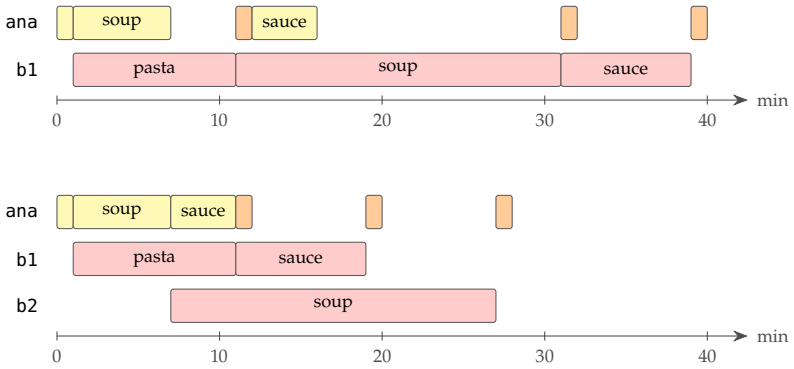


Figure 10.6.: The kitchen with one burner (top, 40 minutes) and with two (bottom, 28). Yellow is preparing, red cooking, orange serving. The rows of the burners show actions that no agent performs; with one burner that row has no gap, and it is the burner that sets the makespan.

A timeline shows at a glance which resource is the bottleneck: the row without gaps. With one burner it is the burner, and a second cook would not help (Exercise 10.8.1 checks it). With two burners nothing is saturated, and the makespan is set by the longest chain: prepare the pasta, prepare the soup, cook the soup, serve it, 1 + 6 + 20 + 1.

An elevator: waiting for people

The second domain has the opposite flavor: an invariant that *allows* sharing. A lift travels between adjacent floors; passengers board and leave, each at their own pace. While somebody boards, the lift must stay at the floor. But several passengers may board at the same time, so the lift is not a token to take: it is a fact that each boarding relies on.

```
;; 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)))))
```

Listing 10.5: Elevators (`elevators-domain.pddl`).

`board` and `leave` ask for `(lift-at ?e ?f)` three times: at start, over all and at end. The lift must be there from the first step of the passenger to the last one, both included, so that it cannot depart at the very instant somebody finishes stepping in. Three passengers go down to the ground floor; Ada and Bob wait on the second, Eve on the third, and Bob needs 4 seconds to board where the others need 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 and Bob board together at 6.002. Ada is in at 8.002, but the lift cannot start before Bob is in, at 10.002; because of the `at end` condition, the

departure comes ϵ later, at 10.003. At the ground floor the three leave at once, and the plan ends when the slowest is out. A second lift waiting on the third floor (`elevators-p02.pddl`) brings the makespan down to 19.005, and it does so alone: the plan never moves the first lift, which would arrive later than the second one passing by on its way down. More resources do not always mean more parallel work.

A token (available, burner-free) keeps users of a resource apart. An invariant (lift-at over all) lets them share it. The row without gaps in a timeline is the bottleneck.

10.9. The MODEL method: time

Durative actions leave the five steps as they were and add questions to two of them, plus one new line in the model card.

What this chapter adds to the method

Step M (model card). Under *Success*: does it matter *when* the goal is reached? If the answer is "as soon as possible", the model needs time and the metric (`total-time`). Under *Agents*: who can act at the same time as whom?
Step D (action table). For each action, four new columns:

- *Does the duration matter, and what does it depend on?* A constant, the parameters (distance over speed), the state.
- *What must hold at the start, throughout and at the end?* The three questions of Section 10.4, for each condition.
- *When does each effect happen?* What the action takes away, at the start; what it produces, at the end.
- *What may run in parallel with it?* Whatever must not needs a token (the agent, a tool, a place) taken at start and returned at end.

Step E (test set). The cheat test becomes a family: ask for an agent in two places at once; ask for a goal whose shortest honest timetable you know, and compare the makespan; read every pair of overlapping lines of a plan and ask whether the world allows that overlap.

Table 10.3 is the action table of the courier of Listing 10.3 with the new columns. Writing it before the PDDL is what would have saved us the two cheating plans of this chapter.

Action	Duration	Needs (s start, o over all)	Changes (s start, e end)
`move`	distance / speed	s: robot idle, in `?from`, rooms connected	s: not idle, not in `?from`. e: in `?to`, idle
`pick`	handling time	s: idle, item here, gripper free. s, o: robot here	s: not idle, item gone, gripper taken. e: carrying, idle
`drop`	handling time	s: idle, carrying. s, o: robot here	s: not idle, not carrying. e: item here, gripper free, idle
`disinfect`	20	s: idle. s, o: robot here	s: not idle. e: room disinfected, idle

Table 10.3.: The action table of the timed courier: step D with the columns that time adds.

Alternatives compared

Instantaneous actions with a cost, or durative actions

The alternative announced in Section 10.1 deserves a fair trial. The domain `courier-cost` keeps the instantaneous actions of Chapter 4 and makes

each one add the seconds it would take to `(total-cost)`, as Chapter 9 taught:

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

On the two robots and two deliveries of `courier-time-p03`, minimizing the cost:

```
$ 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)
```

The fourteen actions are all of `r2`. The plan is optimal, and Kairos proves it. It is also the opposite of what we wanted: the fast robot does everything and `r1` never leaves the dock, because every second of `r1` would cost as much as two of `r2`. For the least total work the answer is right. As a timetable it takes 52.5 seconds, against the 41 of Figure 10.5. Table 10.4 compares the two encodings on the five criteria.

Table 10.4.: Two encodings of "actions take time", run on the same problem (two robots, two deliveries).

	Instantaneous actions with a cost (`courier-cost`)	Durative actions (`courier-time`)
Fidelity	Plans are executable one action at a time. The cost is the total work: 52.5 s. Cannot say when the goal is reached, what overlaps, or that something must hold *while* an action runs	Plans are timetables: makespan 41.007 with both robots. Every overlap the model leaves open will be used, so invariants and tokens must be right
Size	92 ground actions, 38 facts	92 ground durative actions, 38 facts; the engine that handles overlaps splits each into a start and an end (184)
Guidance	Easy: 3336 states, optimal plan proved in 13 ms	First plan in 2 ms, best one after 0.9 s and 161871 states; no proof that it is the shortest
Clarity	One precondition list and one effect list per action	Each condition and effect needs a decision (start, over all, end); the action table grows four columns
Portability	Any planner with numeric fluents; with constant costs, any classical planner with `:action-costs`	Temporal planners only: Kairos, OPTIC, POPF, TFD, LPG-td. Not ENHSP, Metric-FF or LAMA

Verdict. Use a cost when one agent acts, or when what matters is the total effort (energy, money, wear) and not the clock: the model is simpler, the search easier and the result can be proved optimal. Use durative actions when several things can happen at once and the finishing time matters, or when a condition must hold while something goes on. A good practice is to start with the cost model, get the logic right with the fast feedback it gives, and then add time; Chapter 19 comes back to growing a model in versions.

Where an effect goes: at start, at end, or split

The second choice is made inside every durative action. `courier-walk` is the smallest courier (one robot, `move` only, 10 seconds per move) in three versions that differ in two words: both effects at the end, both at the start, or the delete at the start and the add at the end, as everywhere in this chapter. Each is run on a walk of three moves from the dock to ward A, and on a cheat test: be in the pharmacy and in the dock at once, starting from the lobby.

	Both at end	Both at start	Split
During the move, the robot is	still in `?from`	already in `?to`	nowhere
Walk of three moves	30.002	**10.002**	30.002
In two places at once	**a plan, 10.001**	no plan	no plan

Table 10.5.: Three placements of the two effects of `move`, each run on the same two problems (`courier-walk-p01` and `-p02`). In bold, the plans the world does not allow.

With both effects at the end the robot is still in the lobby while it moves, so it can start a second move from there:

```
$ 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]
```

With both at the start the robot is in the next room before it has walked, and the three moves of the walk run almost on top of each other:

```
$ 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]
```

Verdict. Delete at the start what the action uses up, add at the end what it produces. During the action the thing is then in neither state, and nobody can use it twice (the fault of "both at end") or use it before it exists (the fault of "both at start"). This rule decides most effects. It has exceptions with a reason, which Chapter 11 uses on purpose: an action that adds a fact at its start and deletes it at its end makes the fact true exactly while it runs.

A cost model answers "least total work" and proves it; a temporal model answers "done earliest" and exposes every overlap. Inside a durative action: take at start, give at end.

10.10. Common mistakes

Translating an action by renaming its parts

The quickest way to make an action durative is to add a `:duration` and change the keyword. It does not work: conditions and effects without a time specifier are not 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
```

The same happens with effects (`semantic.untimed_effect`), with `:precondition` left in place of `:condition` (`unknown key :precondition in durative action`) and with a forgotten duration (`durative action without :duration`). The error is a favor: each specifier is a modeling decision, and the checker will not take it for you.

Checking at the start what must hold all the way

Symptom: a plan in which an agent starts something and leaves, as the pick at 20.002 followed by a move at 20.003 in Section 10.2. The planner is not wrong; a condition `at start` is about one instant. **Fix:** add the condition `over all`. **How to find it:** look for lines that overlap in the plan and for a makespan shorter than the sum you expected; write the suspicious plan in a file with the repaired domain and let `kairos validate` confirm that it is now rejected.

A token forgotten in one action

The token `idle` works only if *every* action of the agent takes it and returns it. Forget it in one action and that action overlaps with all the others; forget only to return it and the agent never acts again, which usually shows up as a problem with no plan. Nothing in the language ties the predicate to the agent, so no checker will warn. Test it: for each action, a small problem whose fastest honest plan you can compute by hand (Exercise 10.7.1 is one).

A duration that cannot be computed

A duration read from a function needs a value for every ground action that should exist. In `courier-time-p05.pddl` the distance from the dock to the lobby was given in one direction only:

```
$ 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)
```

A move whose duration is undefined cannot be executed, and the robot never leaves the dock. The files are valid PDDL, so `kairos check` is silent; the hint is "numeric conditions" in the note. Give distances in both directions, like connections.

Reading the lines of a temporal plan as a sequence

The lines are sorted by start time and nothing more. An action printed after another may have started at the same instant, and may end before it. Before judging a plan with more than one agent, draw its timeline, one row per agent or resource. And do not expect round numbers: the separations of ϵ are part of the plan, and a hand-written plan without them is rejected (`at start condition false`, Section 10.3).

Declaring less than is used

A domain with durative actions and no `:durative-actions` among its requirements is accepted by Kairos with a warning:

```
$ 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
...
```

Other planners refuse the file, or read it wrongly without a word. The same goes for `:numeric-fluents` when durations come from functions.

10.11. Summary

A durative action has a start, an end and a duration. Its conditions say when they must hold (`at start`, `over all`, `at end`) and its effects when they happen (`at start`, `at end`). A temporal plan gives each action a start time; actions overlap whenever the model does not keep them apart, and the planner uses every overlap that is left open. Modeling with time is, to a large extent, deciding which overlaps the world allows.

Concept	What to remember
`:durative-action`	`:duration`, `:condition`, `:effect`; requirement `:durative-actions`
`(= ?duration e)`	`e` is a number or an expression (distance over speed), evaluated when the action starts
`at start`, `at end`	one instant each; for conditions and for effects
`over all`	the open interval between the two; conditions only; an invariant
Temporal plan	lines `t: (action) [d]`; the order of the lines is not an order of execution
Makespan	the end of the last action; `(:metric minimize (total-time))`
Happening	an instant at which actions start or end; the only moments of change
ϵ	the separation between happenings that interfere; 0.001 in Kairos, `-epsilon`
Take at start, give at end	the default placement of effects; what is in use is in neither state
Token (`idle`)	taken at start, returned at end: one thing at a time for an agent or a resource

Table 10.6.: What this chapter added.

Every plan of this chapter could have been executed by a patient robot one action at a time: overlapping made the plans shorter, never possible. Chapter 11 turns to problems in which a plan exists *only* if actions overlap, and to durations that the planner chooses.

10.12. Further reading

The reference for durative actions is the PDDL 2.1 paper of Fox and Long (2003): its sections on durative actions define the three time specifiers, happenings, interference and the "no moving targets" rule that this chapter explained by example. Haslum et al. (2019) devote a chapter to temporal planning in PDDL, with the same semantics told more formally and a survey of how planners deal with it. For temporal planning beyond PDDL (timelines, chronicles, temporal networks), see the chapters on time of Ghallab et al. (2004) and Ghallab et al. (2016). Plans are checked against the semantics by validators; VAL (Howey et al., 2004) is the classic one, and `kairos validate` follows the same rules. How Kairos chooses between its temporal engines is described in Martín Rico et al. (2026); the planners POPF (Coles et al., 2010) and OPTIC (Benton et al., 2012), which appear in the portability rows of this part of the book, are the usual references for planning with durative actions by forward search.

10.13. Exercises

Exercises for Section 10.1 — What a sequence cannot say

Exercise 10.1.1. A sequential plan has nine actions. Those of robot `a` last 5, 3, 8 and 2 seconds, in that order; those of robot `b` last 4, 4, 6, 1 and 7. The robots never need each other. How long does the plan take if the actions are executed one after another? And if each robot works on its own? Draw the timeline of the second case.

Exercise 10.1.2. For each situation, say whether a cost that adds up durations is enough or the model needs time, and why: (a) a single delivery van that must drive the fewest kilometers; (b) two cranes that must unload a ship before the tide goes out; (c) a dinner in which all the dishes must reach the table hot; (d) a robot that must use as little battery as possible.

Exercises for Section 10.2 — A first durative action

Exercise 10.2.1. In `courier-fixed-domain.pddl`, change the duration of `pick` from 2 to 5 seconds. Before running the planner on `courier-fixed-p01`, predict the makespan. Then run it and explain the result.

Exercise 10.2.2. Breakfast: a toaster takes 120 seconds to toast the bread that is in it, and a kettle takes 180 seconds to boil the water it holds. Each appliance does one thing at a time and works alone once started. Write the domain with two durative actions without parameters, and a problem that asks for toast and hot water. What makespan do you expect?

Exercise 10.2.3. A classmate turns `move` into a durative action like this, and `kairos check` rejects it. Which lines are wrong, and what does the checker say?

```
    :condition (and (at start (robot-at ?r ?from))
                    (at start (connected ?from ?to)))
    :effect (and (not (robot-at ?r ?from))
                 (robot-at ?r ?to))))
```

Exercises for Section 10.3 — What happens at each instant

Exercise 10.3.1. These are four consecutive lines of the plan of the elevator in Section 10.8. List their happenings in order of time. For each pair of consecutive happenings, say whether they interfere and through which fact.

```
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]
```

Exercise 10.3.2. The plan of `courier-time-p01` has makespan 42.005. Predict its makespan with `-epsilon 0.1`, and check. Why is the answer not $42 + 6 \times 0.1$, although the plan has seven actions?

Exercises for Section 10.4 — Conditions in time

Exercise 10.4.1. In `courier-time-domain.pddl`, remove from `pick` the condition that asks for the robot in the room `at start`, keeping the `over all` line. Can the robot now cheat? Run `courier-time-p01` and compare the times of the plan with those of Section 10.5.

Exercise 10.4.2. In `elevators-domain.pddl`, remove the `over all` and the `at end` conditions of `board` and `leave`. Predict what the plan of `elevators-p01` will do, run it, and describe the journey of Bob.

Exercises for Section 10.5 — Durations that depend on the action

Exercise 10.5.1. In `courier-time-p01`, the robot is replaced by one that moves at 2 meters per second. Compute the new makespan by hand (with $\epsilon = 0.001$), then check.

Exercise 10.5.2. A robot that carries something moves more slowly. Extend `courier-time` with a function `(loaded-speed ?r)` so that a move lasts the distance over the speed when the gripper is free, and the distance over the loaded speed when the robot carries an item. Durations cannot depend on a condition, so you will need two actions. Test it on the delivery of `courier-time-p01` with a loaded speed of 0.5.

Exercises for Section 10.6 — Working in parallel

Exercise 10.6.1. In `courier-time-p03`, set the speed of `r2` to 1. Does the planner still find a relay? What is the makespan, and which robot determines it?

Exercise 10.6.2. Take the single delivery of `courier-time-p01` (medicines to ward A) and add the fast robot `r2` in the corridor. Predict which robot will make the delivery and the makespan, and check. What does `r1` do in the plan?

Exercises for Section 10.7 — One thing at a time

Exercise 10.7.1. In a copy of `courier-idle-domain.pddl`, the token was forgotten in `drop`: the action neither needs `idle` nor changes it. A robot arrives at the pharmacy carrying the medicines; it must leave them there and disinfect the room. Compute the makespan of the

honest plan, write the problem, and find out what the planner does with the faulty domain.

Exercise 10.7.2. The counter of a room serves one robot at a time: two robots cannot pick from the same room at once. Add it to `courier-idle` with a token that belongs to the room. Test it with two robots and two boxes of medicines in the pharmacy, both to be taken to the lobby, and compare with the plan of the domain without the token.

Exercises for Section 10.8 — Other domains: a kitchen and an elevator

Exercise 10.8.1. In the kitchen with one burner (`kitchen-p01`), would a second cook shorten the makespan? And a third burner in `kitchen-p02`? Answer from Figure 10.6 first, then check both.

Exercise 10.8.2. In `elevators-p02` the plan uses only the second lift. Compute the makespan of the best plan in which `e1` carries Ada and Bob while `e2` carries Eve, and explain why the planner is right to ignore it.

Exercises for Section 10.9 — The MODEL method: time

Exercise 10.9.1. Apply step L. For each case, choose between instantaneous actions with a cost and durative actions, and justify it with the criteria of Table 10.4: (a) a single warehouse robot whose route must be as short as possible; (b) three painters and a house to be finished as soon as possible, where a wall cannot be painted while its plaster dries; (c) a fleet of trucks whose total fuel is to be minimized, with no deadline.

Exercise 10.9.2. In `courier-time-domain.pddl`, move the effect `(item-at ?i ?l)` of `drop` from the end to the start. Which plan does the world not allow now? Run `courier-time-p03` and find the two lines that show it.

11. Coordination in time

In Chapter 10 two robots worked at the same time because it was faster. Nothing forced them to: a single patient robot could have executed any of those plans one action after another, and the goal would have been reached all the same, only later. Now the lab gets a spring door. It stays open only while somebody holds it, and a robot that holds a door cannot walk through it. One robot has to hold the door *while* the other one passes. There is no slow version of this plan with one action at a time: either the two actions overlap, or the sample never enters the lab. A second situation is as common. The robot is at the dock with a battery that is almost empty, and the question is not whether to charge but for how long. With the durations of Chapter 10, fixed by the model, the planner has no say in it.

This chapter is about those two things: actions that *must* overlap, and durations that the planner *chooses*. The first is called required concurrency, and it divides temporal problems, and temporal planners, into two families. It is modeled with one recurring pattern, an action that keeps a fact true exactly while it runs. The second needs a small extension of the language, `:duration-inequalities`, which turns `?duration` from a number the model computes into a decision.

We start with the classic example, the match cellar, where required concurrency fits in twenty lines. Then the courier meets the spring door, and we look at what the distinction means for planners and for Kairos in particular. Two robots lift a crate together, which is harder to model well than it seems. The second half of the chapter gives the planner the choice of a duration (a charge, a coffee), uses the chosen value in effects, and takes a first look at the machinery with which planners keep all these times consistent. The method, the mistakes, a summary and the exercises close it.

11.1 Plans that exist only if actions overlap 236
11.2 The envelope: a door held open 238
11.3 Temporally simple, temporally expressive . 240
What Kairos does 241
11.4 Working together: two robots and a crate 242
A first model, and why it is brittle 242
A model with slack . . . 243
11.5 Durations that the planner chooses 244
`?duration` in effects . . 245
How a planner chooses a duration 246
A coffee machine 247
11.6 How planners order time 248
11.7 The MODEL method: coordination 249
Alternatives compared . 249
11.8 Common mistakes . . . 251
11.9 Summary 253
11.10 Further reading 253
11.11 Exercises 254

11.1. Plans that exist only if actions overlap

A fuse has blown and the cellar is dark. You have a box of matches. A match burns for 5 seconds; mending a fuse takes 2 seconds and cannot be done in the dark. Listing 11.1 is the whole domain.[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)
    (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))
```

1: The match cellar is a classic of temporal planning; a version of it was a benchmark of the 2011 planning competition. The file here is our own, with the names of this book.

```
    :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)))))
```

Listing 11.1: The match cellar (`cellar-domain.pddl`).

Look at the effects of `light`. In Chapter 10 we gave a rule for placing effects: delete at the start what the action uses up, add at the end what it produces. `light` follows it for the match (it stops being `unused` at the start) and breaks it, on purpose, for the light: `(lit ?m)` is added at the start and deleted at the end. The fact is true exactly while the action runs. It is not a product that remains when the action is over; it is something the action provides while it lasts.

`mend` needs that fact at its start and over all its duration. So a mend can only happen inside a `light`: after its start and before its end.

```
$ 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]
```

The mend starts ϵ after the match is lit (its start needs a fact that the start of `light` adds, so the two happenings interfere) and ends at 2.001, well inside the 5 seconds. The makespan is the life of the match: a plan ends when its last action ends, even if the goal became true before.

Now try to execute these two actions one after the other, in either order. Mend first: it is dark, the condition fails. Light first and mend when it has finished: the match is out, it is dark again. No sequence of whole actions solves the problem. A problem like this one has **required concurrency**: every plan that solves it contains actions that overlap. Figure 11.1 shows the two failed sequences and the plan.

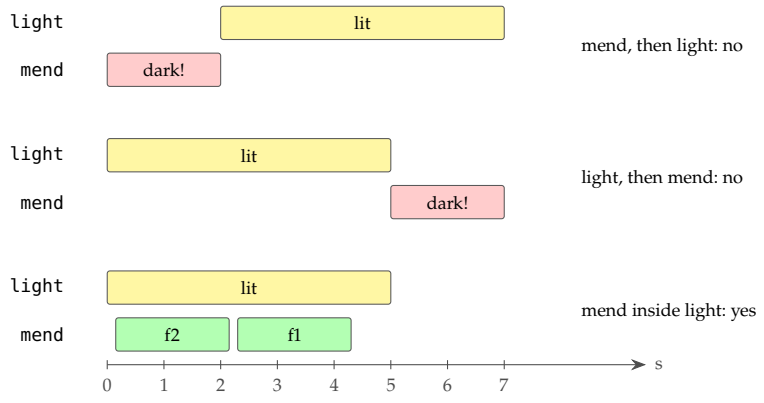


Figure 11.1.: The two ways of putting `light` and `mend` in sequence fail: the mend is in the dark. The only plans put the mend inside the light. The bottom timeline is the plan of `cellar-p02`, with two fuses mended by the light of one match.

How many fuses can one match serve? Two mends take 4 seconds, which fit in 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]
```

The two mends do not overlap each other: `(hand-free)` is a token, as `idle` was for the robot. Three mends need 6 seconds and one match is not enough:

```
$ 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
```

The answer deserves a careful reading. Kairos does not say `unsolvable` (exit code 10), as it did for the problems without a plan of earlier chapters. It says `no plan found` (exit code 14): it searched everything it knows how to search and found nothing, but the engine that handles overlapping actions does not prove that nothing exists. Here we know by arithmetic that there is no plan. In general, with required concurrency, "the planner found no plan" is weaker evidence than it was, and the test of step E that must have no plan should be one whose answer you can justify by hand. With a second match (`cellar-p04.pddl`) the three fuses are mended in 6.003 seconds, the second match lit while the first one still burns.

A problem has required concurrency when every plan needs actions that overlap. The match gives light only while it burns; the mend must happen inside it.

11.2. The envelope: a door held open

The shape of `light` is a pattern worth a name. An **envelope** is a durative action that adds a fact at its start and deletes it at its end, so that other actions, the **contents**, can run only inside it. The envelope models anything that is available for a while and then stops by itself: the light of a match, a door held open, a lamp that overheats, a window of radio contact, a grip.

The courier world gets its envelope with the spring door of the opening. Figure 11.2 shows the floor: the corridor and the lab are no longer `connected`; a door joins them, with the vocabulary of Chapter 5: `(joins ?d ?a ?b)` and `(open ?d)`. The door is closed unless a robot holds it.

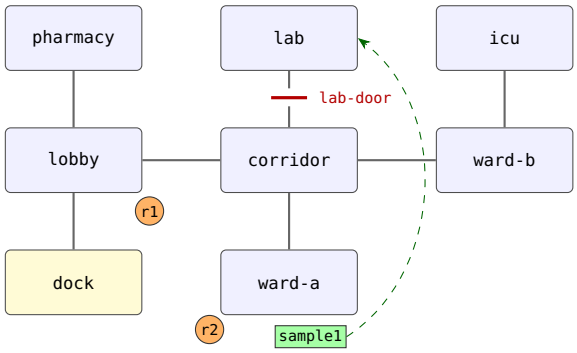


Figure 11.2.: The courier world of this chapter: a spring door (red) between the corridor and the lab. `r2` must take the sample into the lab; `r1`, in the lobby, carries nothing.

The domain `courier-door` is the courier of Chapter 10 that does one thing at a time (with `idle`), plus two actions. Listing 11.2 shows them.

```
;; 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))))
```

Listing 11.2: The envelope `hold-door` and its content `pass` (from `courier-door-domain.pddl`).

`hold-door` is the envelope: the door is open from its start to its end, 15 seconds later, and the robot that holds it is not idle and stays where it is. `pass` is a move through the door; it needs the door open at its start and all the way. Since `pass` takes the token `idle` of the robot that passes, and `hold-door` that of the robot that holds, one robot cannot do both.

Step D: an action that provides something while it runs

For every action, ask what it makes true *only while it lasts*. If there is such a fact, the action is an envelope: add the fact at start, delete it at end, and give every action that depends on it the fact as a condition at start and over all. Then ask who performs the envelope and who performs the contents, and whether the model lets them be the same agent. Here the token `idle` answers: the holder is busy for the whole envelope.

`r2` is in ward A with the sample, 12 meters from the corridor; `r1` is in the lobby, 10 meters away; passing the door takes 8 seconds.

```
$ 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]
```

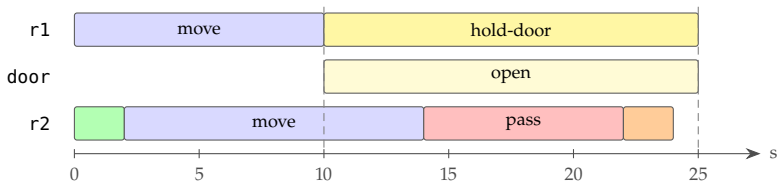


Figure 11.3.: The plan of `courier-door-p01`. The middle row is not an action but the fact `(open lab-door)`, true while `r1` holds the door. The `pass` of `r2` lies inside it; the hold starts four seconds before `r2` arrives, because nothing tells the planner to hold as late as possible.

`r1` reaches the door at 10 and holds it from 10.001 to 25.001. `r2` arrives at 14.001 and passes from 14.002 to 22.002, inside the envelope (Figure 11.3). The makespan is the end of the envelope, 25.001, although the sample is in the lab at 24.003: the door cannot be released early, because the model fixes the hold at 15 seconds. We will let the planner choose that duration in Section 11.5.

The test that must have no plan is a robot alone:

```
$ 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)
```

This time the verdict is `unsolvable`, with a proof: Kairos forgot the durations and the times, let starts and ends happen in any order, and even so a robot alone cannot be on both sides of its own token. When that simpler problem has no solution, the temporal one has none either.

An envelope adds a fact at its start and deletes it at its end; its contents need the fact at start and over all. The door is open only while a robot holds it, and the holder is busy.

11.3. Temporally simple, temporally expressive

Every problem of Chapter 10 had this property: if it had a plan at all, it had a plan in which no two actions overlap. The cellar and the door do not. Cushing et al. (2007) made the distinction precise and gave it names. A problem has required concurrency if all its solutions contain overlapping actions. A language (a set of features that a domain may use) is **temporally expressive** if problems with required concurrency can be written in it, and **temporally simple** if they cannot.

What makes PDDL temporally expressive is what the envelope uses: an effect at the start of an action that is undone at its end, and a condition of another action that needs the fact in between. Take that away (every action only consumes at start and produces at end, as in the courier of Chapter 10) and any plan can be stretched into a sequence without breaking it.

The distinction matters because it separates two ways of building a temporal planner.

- **Sequence first, schedule later.** Treat each durative action as one instantaneous action with all its conditions and effects, find a sequential plan as a classical planner would, and then compute start times, letting actions that do not depend on each other overlap. This is fast, because the search is classical. It cannot solve the cellar: as one block, `light` leaves no light behind, and no sequence works.
- **Search over starts and ends.** Split each durative action into its two happenings and let the search decide the order of all of them, so that a mend can start after a light has started and before it has ended. This solves the cellar. It searches a larger space, because every action is now two steps, and it must keep track of which actions are running and of the times that are still possible.

Cushing and his colleagues observed that most of the temporal benchmarks of the time were temporally simple, and showed that planners of the first kind cannot solve problems with required concurrency, however long they search. Planners of the second kind include POPF (Coles et al., 2010) and OPTIC (Benton et al., 2012).

What Kairos does

Kairos has both. Its engine **M2** is of the first kind: it compresses each durative action into an instantaneous one, searches, and schedules the plan. Its engine **M3** is of the second kind: it searches over start and end happenings and keeps the times in a temporal network, as POPF does. It looks at the task to choose, and with `-v` it tells what it saw:

```
$ 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" is the envelope, detected. For the courier of Chapter 10, where no action does that, the same lines said `engine M2: 92 durative actions compressed`, and M3 was used afterwards only to shorten the makespan. The option `-engine` forces one engine, which is a good way to see the difference:

```
$ 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)
```

Two states searched: the initial one, and the one after a `light` taken as a block, from which nothing useful can be done. This is what a planner of the first kind makes of the cellar. The messages also explain the exit code 14 of the previous section: neither engine is complete for these problems. M2 misses every plan that needs an overlap; M3, like POPF, keeps a

single way of reaching each state, and may discard the one that had a future.

What this means for the modeler

- **Portability.** Required concurrency is a feature, although it has no requirement flag. Of the temporal planners in the table of Appendix B, Kairos, OPTIC, POPF and TFD return valid plans for it; LPG-td does not.
- **Guidance.** A temporally simple model is searched as a classical one, and the planner can often say that a problem is unsolvable. A model with envelopes is searched over starts and ends, and "no plan found" comes without proof.
- **Choice.** Sometimes the same situation can be modeled with or without an envelope (Section 11.7 compares both). The envelope should be there because the world has one, not by accident.

PDDL is temporally expressive: an effect at start undone at end, needed by another action, forces an overlap. Planners that sequence whole actions cannot solve such problems; Kairos switches to its engine M3 for them.

11.4. Working together: two robots and a crate

A crate of oxygen bottles is too heavy for one robot. Two robots must lift it onto a cart together. This is coordination of another kind: nobody waits for anybody; two agents must be doing something during the same interval.

A first model, and why it is brittle

The direct translation of "together" gives each robot its own action, which needs the grip of its partner for as long as it lasts:

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

Listing 11.3: A lift that needs the partner's grip all the while (from `courier-crate-sync-domain.pddl`).

Each `lift` is an envelope of its own grip and a content of the other's. Think about when the two actions can start. The lift of `r1` needs the grip of `r2` from just after its own start; the lift of `r2` needs the grip of `r1` from just after its own start. If either starts first, even by a millisecond, its invariant fails at once. The only plans start both at *exactly* the same instant. The validator confirms both halves:

```
$ 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
```

So a plan exists. The planner does not find it:

```
$ 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)
```

A search over starts and ends puts happenings in an order, one after another. Whichever lift starts first, its invariant is false in the very next state, and that branch is a dead end: two starts are never taken as a single step. The warning of the note ("not a proof") was honest, and this is a case where the plan it could not exclude exists. But the fault is in the model. A model whose only plans need two things to happen at exactly the same instant is asking for something that planners do not search for and that no pair of real robots can do. Real coordination has slack.

A model with slack

Give the slack to the model. Each robot *grips* the crate for a while: an envelope of 8 seconds. The crate *rises*, which takes 4 seconds, while both grips last.

```
;; 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)))))
```

Listing 11.4: Two robots lift a crate, with slack (`courier-crate-domain.pddl`).

```
$ 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]
```

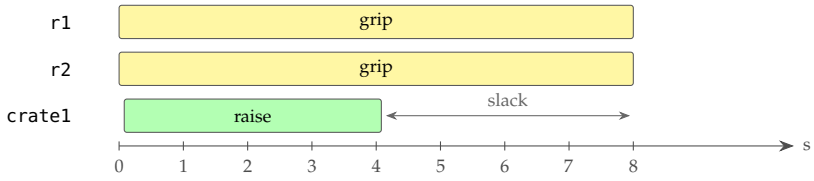


Figure 11.4.: Two envelopes and one content. The `raise` can be anywhere inside both grips; the grips do not have to start together, as long as they share 4 seconds.

Here the two grips happen to start together, because nothing prevents it, but they do not have to: any two grips that share a little more than 4 seconds will do (Exercise 11.4.1 computes how late the second may come). `raise` belongs to no robot; it is what happens to the crate, as cooking was what happened to the soup in the kitchen of Chapter 10.

Two tests complete step E. One robot alone must have no plan, and it has none (`courier-crate-p02.pddl` is `unsolvable`). And the cheat test: can one robot play both parts? The condition `(at start (not (= ?r1 ?r2)))` is there for that, with the `:equality` of Chapter 4. In a durative action, an equality between parameters also needs a time specifier; `at start` is the natural one, since it never changes. Without that line (`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]
```

"Together" modeled as mutual invariants forces an exact simultaneous start, which planners do not find and robots cannot do. Model the slack: an envelope per agent and the joint change inside both.

11.5. Durations that the planner chooses

So far a duration was a number that the model computes: a constant, or an expression like distance over speed. For many actions that is wrong, because how long they last is a decision. How long to hold a door, how long to charge, how long to keep the pump running, how long to wait. PDDL 2.1 allows the `:duration` of an action to be a set of bounds on `?duration` instead of an equality, and leaves the value to the planner. It needs the requirement `:duration-inequalities`.

The courier gets a battery, as in Chapter 8: `(battery ?r)` goes from 0 to `(battery-capacity ?r)`, a move uses `(consumption ?r)` units of battery per meter, and the dock `(has-charger dock)`. New here is `(charge-rate ?r)`, the units of battery gained per second of charging. To keep the listing short, this courier only moves and charges.

```
;; 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)))))
```

Listing 11.5: A charge of chosen length (from `courier-charge-domain.pddl`).

The `:duration` of `charge` is a conjunction of two bounds. The lower one, 1 second, says that a charge shorter than that is not worth plugging in. The upper one is computed from the state at the start of the action: the battery that is missing, divided by the rate, is the time that fills it. A robot with 20 of 100 that gains 2 per second can charge for at most 40 seconds. Either bound can be left out: `(<= ?duration 60)` alone means "any duration up to a minute".[2]

The move takes all the battery it will use at its start, and needs to have it then. This is the cautious choice: if the consumption were subtracted at the end, a second action could spend, during the move, the charge that the move was counting on.

2: PDDL does not fix a smallest duration, but a plan with an action of length zero makes little sense, and the separation ϵ applies between its two ends. Give a lower bound whenever the world has one.

`?duration` in effects

The effect of `charge` is where the choice counts:

```
(at end (increase (battery ?r)
          (* ?duration (charge-rate ?r))))
```

`?duration` can be used in the numeric effects of the action, and stands for the value that was chosen. A longer charge gives more battery. Without this the choice would be idle: nothing would depend on it, and the planner would always take the shortest. The whole gain arrives at the end of the action, in one step; a battery that fills little by little while the action runs is a continuous effect, for Chapter 12.

Kairos lists what it found in the files:

```
$ 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)
```

In `courier-charge-p01` the robot is in the dock with 20 of battery and must reach the ICU: 31 meters, at 1 unit per meter. It needs 11 more, that is, at least 5.5 seconds of charge.

```
$ 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]
```

Two plans, and both tell something about how the planner chooses. In the first the charge lasts 40 seconds, its upper bound: a full battery. In the second there are six charges of 1 second, the lower bound, one after another: 12 units, one more than needed, and a makespan of 37.009.

Neither is the plan a person would write, a single charge of five seconds and a half. That plan is valid, and better:

```
$ kairos validate courier-charge-domain.pddl courier-charge-p01.pddl
      \
    courier-charge-p01-hand.plan
plan valid, makespan 36.504, metric 36.504
```

How a planner chooses a duration

Choosing a real number is a different kind of decision from choosing an action. There are infinitely many values between 1 and 40, and a search cannot try them all. Each planner has its own way out. What Kairos does is this:

- Its engines M2 and M3 try each duration at its two **bounds**, the shortest and the longest. That is what we just saw: the full charge, and then the minimum charge repeated.
- If no plan is found that way, the task is also planned in **steps of time**: an action then lasts until a step of 0.1 seconds passes, or until the instant at which ending it makes a condition true, and `?duration` in its end effects is the time it ran. This finds durations *in between* the bounds when a plan needs one.
- Whatever the search does, the validator accepts any duration within the bounds, as it accepted our 5.5, and rejects any other: `courier-charge-p01-long.plan`, with a charge of 41 seconds, fails with `41 <= 40 is false`.

The plans are valid; they are not always the tightest. If the exact length matters, say so in a way the planner can see, or accept a plan and adjust the duration yourself, with the validator as judge. The next example needs a duration in between, and gets it.

A coffee machine

The pump of a coffee machine runs for as long as it is told, and the cup receives 25 milliliters per second. An espresso is between 40 and 50 milliliters.

```
;; 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)))))
```

Listing 11.6: A coffee machine (`coffee-domain.pddl`).

A cup is filled in one go: `(empty ?c)` is consumed at the start of the pour. The upper bound of the duration is the time that fills the cup, here 250/25 = 10 seconds. Neither bound gives an espresso: 1 second is 25 ml, too little, and 10 seconds is 250 ml, far too much. The goal is a range, as Chapter 8 taught for numeric goals: `(>= (level cup1) 40)` and `(<= (level cup1) 50)`.

```
$ kairos coffee-domain.pddl coffee-p01.pddl
; cost = 1.6
; makespan = 1.600
0.000: (pour cup1) [1.600]
```

The pour lasts 1.6 seconds: the instant at which ending it makes the goal true (40 ml). With `-v`, the line `[orchestrator] also in steps of 0.1 s` shows where the plan came from. Any duration from 1.6 to 2 would be valid; the planner returns the first that works.

With :duration-inequalities the duration is bounded, not fixed, and the planner chooses it. Used in an effect, ?duration makes the choice matter. Kairos tries the bounds first and durations in between when the bounds fail.

11.6. How planners order time

A planner that searches over starts and ends decides an *order* of happenings: this start, then that start, then this end. It still has to find *times* for them, such that every duration is respected and every separation kept, or to discover that no such times exist (three mends inside one match). The tool for that is the same in most temporal planners, and it is worth knowing because it explains what is easy for them and what is not.

Simple temporal networks

A **simple temporal network** (STN) has a variable for each time point (here, each start and each end) and constraints of one form only: the difference between two time points lies between two bounds, $l \le t_j - t_i \le u$ (Dechter et al., 1991). Everything a temporal plan asks of its times has that form:

- an action of duration 5: $t_{end} - t_{start} = 5$ (both bounds are 5);
- a duration to choose between 1 and 40: $1 \le t_{end} - t_{start} \le 40$;
- a happening that must come after another one: $t_j - t_i \ge \epsilon$.

An STN is drawn as a graph: a node for each time point, and an arrow from i to j labeled $[l, u]$. Whether times exist that satisfy all the constraints, and the earliest such times, can be computed with a shortest-path algorithm: the network is consistent exactly when no cycle of constraints adds up to a contradiction. No search is needed, which is why planners can afford to check the network every time the plan grows.

Figure 11.5 is the network of the plan of `cellar-p01`. The search chose the order: start of `light`, start of `mend`, end of `mend`, end of `light`. The network turns it into times.

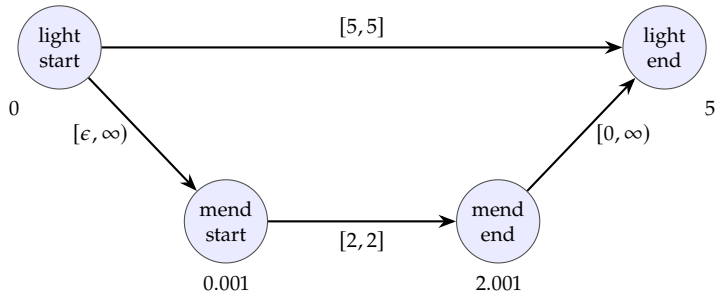


Figure 11.5.: The simple temporal network of the plan of `cellar-p01`. Each arrow bounds the time from its tail to its head. Under each node, the earliest time that satisfies all of them: the times of the plan. If the mend lasted 5 seconds, the path through the mend would need $\epsilon + 5$ where the direct arrow allows exactly 5: a contradiction, and no plan.

The four arrows are the four things the plan must respect: the two durations; the mend starts after the light is on, at least ϵ later because its start needs `(lit m1)`; and the mend ends no later than the light does (the

invariant covers the open interval, so the two ends may coincide). The earliest times that satisfy them are 0, 0.001, 2.001 and 5, exactly the plan printed in Section 11.1.

This picture explains several things we have seen. Plans start actions as early as the network allows, which is why `r1` began to hold the door four seconds before it was needed. A chosen duration is an arrow with two different bounds, which the network handles with no effort; what is hard is that the *effects* depend on the value chosen (the battery gained), and effects are outside the network. And an exact simultaneous start, as in the brittle lift, is not a problem of times (the network can say $t_j - t_i = 0$) but of the search, which has to think of it.

The search orders starts and ends; a simple temporal network turns the order into times, or shows that none exist. Durations, chosen or fixed, and separations are all bounds on differences.

11.7. The MODEL method: coordination

What this chapter adds to the method

Step D (action table). Two questions for each action, after those of Chapter 10:

- *Must it overlap with another action?* Because it provides something only while it runs (an envelope), because it needs such a thing (a content), or because two agents must act during the same interval. Write down which action contains which, and how much slack there is.
- *Is its duration a decision?* If somebody could reasonably ask "for how long?", give bounds instead of a value, and make sure that some effect or some condition depends on `?duration`; otherwise the choice means nothing.

Step E (test set). For a model with envelopes: a content that does not fit (three fuses, one match) must have no plan; the agent of the envelope must not be able to be its own content (one robot at the door); two roles must not be played by one object (the crate). And remember that "no plan found" is not a proof here: prefer tests whose answer you can compute. For a chosen duration: write by hand the plan with the duration you would choose, and validate it.

Alternatives compared

An envelope action, or a lock predicate

The envelope is not the only way to say "the door is open while a robot holds it". The pattern of Chapter 7 for a resource that is taken and released was two actions and a predicate: a **lock**. In time it looks like this.

```
;; 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))))
```

Listing 11.7: The spring door with a lock predicate (from `courier-door-lock-domain.pddl`).

`open-door` takes the token of the robot and does not return it: the robot stays busy, holding. `release-door` closes the door and frees the robot. Between the two, the door is open for as long as the plan wants. On the delivery of `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]
```

The plan is a second shorter than with the envelope, and it has a hole: nobody releases the door. The goal did not ask for it, so the plan ends with `r1` holding the door of the lab forever. An envelope cannot be left open; a lock can, unless the goal forbids it. `courier-door-p04` is the same delivery with `(idle r1)` and `(idle r2)` added to the goal, and then the plan has its `release-door`, at 22.002, the instant at which `r2` is through. Table 11.1 compares the two encodings on that problem.

Table 11.1.: Two encodings of a door that is open while a robot holds it, run on `courier-door-p04`.

	Envelope (`hold-door`)	Lock (`open-door`, `release-door`)
Fidelity	The door always closes, after exactly 15 s. Too rigid if the hold could be shorter (makespan 25.001)	The door is open as long as needed (24.003), with no upper limit, and stays open for good unless the goal says otherwise
Size	64 ground actions, 31 facts	66 ground actions, 33 facts
Guidance	Required concurrency: searched over starts and ends (1919 states); "no plan found" may come without proof	Temporally simple: searched as a sequence and scheduled (first plan by engine M2)
Clarity	One action says the whole thing: who holds, for how long	Two actions and a predicate that must stay in step; the goal has to mention the door or the robots
Portability	Planners with required concurrency only	Any temporal planner

Verdict. Use an envelope when the thing ends by itself or has a maximum length that the agent does not control: a match burns out, a lamp overheats, a pass of a satellite ends. There the lock is wrong, because it lets the plan keep the match burning (Exercise 11.7.2). Use a lock when the agent decides when to stop and there is no limit: a door that is simply opened and closed, a tool that is picked up and put down. Then add to the goal, or to every problem, that things are left as they should. And when the agent decides *and* there is a limit, the best of both is an envelope with a chosen duration, `(<= ?duration 15)`.

A fixed duration, or a chosen one

For the charge there are three encodings, and they are three files with the same domain name, so that the same problem runs on all of them.

Full: the charge always fills the battery, with a duration computed from the state (the missing battery divided by the rate, written with = where Listing 11.5 has <=) and an effect that assigns the capacity. *Blocks*: every charge lasts 5 seconds and adds 5 times the rate, and the planner chooses how many. *Chosen*: the inequality of Listing 11.5. Figure 11.6 draws the plans for the trip to the ICU.

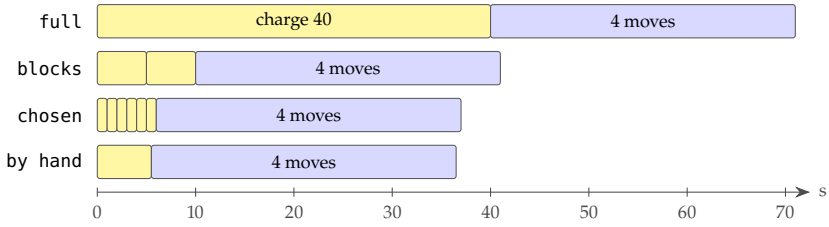


Figure 11.6.: The trip of `courier-charge-p01` with the three encodings of the charge, as planned by Kairos, and the plan written by hand for the third. Yellow is charging. The four moves take 31 seconds in all of them; only the charge differs.

Table 11.2.: Three encodings of a charge, run on `courier-charge-p01`.

	Full charge	Blocks of 5 s	Chosen duration
Fidelity	Loses solutions: the robot cannot leave early. Makespan 71.004	Close: charges of 5, 10, 15... s. Makespan 41.005	Every charge from 1 s to full is a plan. Kairos returns 37.009; the best is 36.504
Size	15 ground actions	15 ground actions	15 ground actions
Guidance	One charge decides everything	More steps; the block size trades precision for plan length	The planner must choose a number; engines differ in how, and plans may be far from the best
Clarity	Simple	Simple, with an arbitrary constant (why 5?)	Says what is meant
Portability	Any temporal planner that computes durations from the state	Any temporal planner with numbers	Kairos, OPTIC, POPF; not TFD or LPG-td

Verdict. If the action really has one natural length (fill the tank, bake the bread), fix it. If the length is a decision and the planner you will use supports it, bound it: the model says what is meant, and every reasonable plan is a plan of the model. Blocks are the portable compromise, and a perfectly good one when the precision of a block is enough; choose the block with the same care as any other number of the model.

An envelope cannot be left open and has a length; a lock is flexible and portable but must be closed by the goal. A chosen duration says what is meant; blocks are its portable approximation.

11.8. Common mistakes

An envelope by accident

Symptom: a model that was solved at once stops being solved, or Kairos with `-v` reports `engine M3: (...) sets (...) only while it runs, and other actions can use that` for an action you did not think of as an envelope. **Cause:** a fact added at start and deleted at end that another action reads; often a "busy" flag written the wrong way round (`(working ?r)` added at start and deleted at end, and used as a condition somewhere). **Fix:** if nothing should happen inside the action because of that fact, use a token that is deleted at start and added at end, as `idle`. Keep envelopes for what the world provides only for a while.

Taking “no plan found” for “unsolvable”

With required concurrency the two are different answers, with different exit codes (14 and 10). The first comes with a note that says so:

```
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]
```

The brittle lift of Section 11.4 had a plan and got this answer. Before concluding that a model has no plan, write the plan you expected in a file and validate it. If it is valid, the model is right and the search could not find it: look for what makes it hard (usually a need for exact timing) and give the model slack.

The agent of the envelope is also its content

If `hold-door` did not take `idle`, one robot would hold the door and pass through it; if `raise` did not ask for two different robots, one would lift the crate alone (the plan with `(raise crate1 r1 r1 dock)` of Section 11.4). Parameters of the same type can always be bound to the same object. Whenever a model says “another”, there must be a token or an inequality behind the word.

A chosen duration that nothing depends on

An action with `(<= ?duration 60)` whose effects do not mention `?duration` gives the planner a choice without consequences, except on the clock. Used as an envelope it makes sense (hold the door for as long as it takes). Used for a charge, it lets the planner fill the battery in the shortest time allowed. Check that the quantity the action is for grows with `?duration`, and that the upper bound stops it where the world would: the validator enforces the bound,

```
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]
```

but only if the model states it. Without the upper bound, the battery would go past its capacity.

Expecting the best duration

The first plan for the trip to the ICU charged for 40 seconds when 5.5 were enough, and the best one returned used six charges of a second. Both are correct answers to the model. A planner that tries the ends of an interval will give plans at the ends. When a duration in between matters, make the model need it (a goal or a condition that the bounds do not satisfy, as the range of the espresso), or use blocks, or adjust the plan and validate it.

11.9. Summary

Some problems can be solved only by overlapping actions, and some durations are decisions. The first comes from a fact that an action provides only while it runs; the second, from a `:duration` written as bounds.

Table 11.3.: What this chapter added.

Concept	What to remember
Required concurrency	every plan has overlapping actions; no sequence of whole actions works
Envelope	adds a fact at start, deletes it at end; its contents need the fact at start and over all
Temporally expressive	a language that can state required concurrency; PDDL 2.1 is. Planners that sequence whole actions are incomplete for it
Kairos engines	M2 compresses and schedules (temporally simple tasks); M3 searches starts and ends (envelopes). `-v` shows which, `-engine` forces one
No plan found (exit 14)	the search ended without a plan and without a proof; validate the plan you expected
Joint work	an envelope per agent and the joint change inside all of them; never an exact simultaneous start
`:duration-inequalities`	`:duration (and (>= ?duration a) (<= ?duration b))`; the planner chooses
`?duration` in effects	`(increase (f) (* ?duration rate))` at end: the choice has a consequence
STN	time points and bounds on their differences; turns an order of happenings into times
Lock predicate	two actions and a fact: flexible and temporally simple; the goal must close it

In all of this the world changed only at the two ends of an action: the battery jumped by 80 units at the last instant of a charge of 40 seconds. Chapter 12 lets quantities change *while* the action runs, so that a robot can be stopped halfway through a charge with half the gain, and a condition can watch a value as it moves.

11.10. Further reading

Required concurrency, and the classification of temporal languages into simple and expressive, come from Cushing et al. (2007); the paper is short and its examples are close to the ones of this chapter. Duration inequalities and the use of `?duration` in effects are defined with the rest of PDDL 2.1 by Fox and Long (2003). Simple temporal networks were introduced by Dechter et al. (1991), who also study the harder networks in which a constraint may be a choice among several intervals. How a forward search over starts and ends uses such a network is the subject of the POPF paper (Coles et al., 2010), and OPTIC (Benton et al., 2012) builds on it. Haslum et al. (2019) explain required concurrency and the compilation of durative actions into start and end actions in their chapter on temporal planning, and Ghallab et al. (2016) treat temporal networks and their controllability, the question of what happens when some durations are decided by the world and not by the agent. The engines of Kairos and how it chooses among them are described by Martín Rico et al. (2026).

11.11. Exercises

Exercises for Section 11.1 — Plans that exist only if actions overlap

Exercise 11.1.1. In the cellar, a match burns for 5 seconds and a mend takes 2. Suppose the mend took 5 seconds, exactly as long as the match. Is there a plan for one fuse? Argue from the conditions of `mend`, then change the duration in a copy of the domain and run `cellar-p01`. Which exit code do you get, and why not 10?

Exercise 11.1.2. For `cellar-p01`, the match is lit at 0. Predict whether each of these plans is valid, then check with `kairos validate`: (a) the mend starts at 0.000; (b) the mend starts at 3.000 and so ends at the instant the match goes out; (c) the mend starts at 3.500.

Exercise 11.1.3. A camera in a greenhouse photographs plants. A photo takes 4 seconds and needs a flash lamp on all that time. A lamp stays on for 10 seconds and then must cool down for the rest of the session; the camera takes one photo at a time. Write the domain, and a problem with three plants and two lamps. How many lamps does the plan light, and when?

Exercises for Section 11.2 — The envelope: a door held open

Exercise 11.2.1. In `courier-door-p03` the sample is inside the lab and must be brought to ward A. Going in, picking and coming out take 18 seconds, more than one hold. Run it and draw the timeline of the plan. Where is `r1` at the end? Write a shorter plan by hand in which `r1` never enters the lab, and validate it.

Exercise 11.2.2. In `courier-door-p01`, robot `r2` becomes slow: its speed is 0.5, so passing the door takes it 16 seconds, one more than a hold. Is the problem unsolvable? Predict, then run it.

Exercises for Section 11.3 — Temporally simple, temporally expressive

Exercise 11.3.1. Say whether each domain can state problems with required concurrency, and which action is the envelope if it can: `courier-time` and `kitchen` of Chapter 10, `cellar`, `courier-door`, `courier-door-lock`, `courier-crate`. Check your answers with the `[orchestrator]` lines of `kairos -v`.

Exercise 11.3.2. Run `courier-door-p01` and `courier-door-p04` with `-engine m2`, first with the envelope domain and then with the lock domain. Explain each of the results.

Exercises for Section 11.4 — Working together: two robots and a crate

Exercise 11.4.1. In `courier-crate`, a grip lasts 8 seconds and the raise 4. If `r1` grips at 0, what is the latest instant at which `r2` can start its grip so that a plan still exists (with $\epsilon = 0.001$)? Write the plan with that grip and validate it, and one a millisecond later.

Exercise 11.4.2. A heavier crate needs three robots. Change `raise` so that three different robots must be gripping. How many inequalities does it need? Test it with three robots, and say how many ground

instances of `raise` there are for three robots, one crate and one place.

Exercises for Section 11.5 — Durations that the planner chooses

Exercise 11.5.1. In `courier-charge-p01` the goal changes: the robot must go to ward A and be back in the dock at the end. With pen and paper: how much battery does the round trip need, what is the shortest single charge before leaving that allows it, and what is the makespan of that plan (without the separations)? Would it help to charge again on the way?

Exercise 11.5.2. Extend the coffee machine with a second cup and an action that swaps the cup under the spout, which takes 3 seconds and must not happen while the pump runs. Ask for an espresso (40 to 50 ml) and a lungo (100 to 120 ml). Which durations does the planner choose?

Exercise 11.5.3. Every second of charge at the dock costs 3 cents. Add a function `(price ?l)` and make `charge` add its cost to `(total-cost)`; minimize it in `courier-charge-p01`. Read what Kairos prints with the plan. Then validate the plan by hand with a charge of 5.5 seconds and compare the metric.

Exercises for Section 11.6 — How planners order time

Exercise 11.6.1. Take the plan of `courier-crate-p01` and draw its simple temporal network: six time points and the constraints among them. Which constraints come from durations, which from conditions at start, which from invariants? Compute the earliest times by hand and compare them with the plan.

Exercise 11.6.2. In the network of Figure 11.5, replace the duration of `light` by a chosen one between 1 and 5 seconds. Which arrow changes? What is the shortest match that still allows the mend, and what is then the makespan?

Exercises for Section 11.7 — The MODEL method: coordination

Exercise 11.7.1. Apply step L. For each case choose between an envelope and a lock, and between a fixed and a chosen duration, with one sentence of justification: (a) a drawbridge that a boat must pass under while it is raised, and that an operator raises and lowers; (b) a satellite that can send data only while it is over the ground station, 8 minutes per orbit; (c) a robot arm that holds a part while another arm welds it; (d) an oven that must bake a loaf for exactly 35 minutes.

Exercise 11.7.2. Rewrite the match cellar with a lock: `strike` lights a match and `blow-out` puts it out. Run the problem with three fuses and one match (`cellar-p03`). What does the planner answer now, and what has the model lost?

12. Continuous change

The durative actions of Chapter 10 and Chapter 11 take time, but their numbers do not: a number changes at the start of an action or at its end, in one jump, and between the two it stands still. For the battery of the courier this is a simplification that we accepted without looking at it closely. Now we look. Robot `r1` is at the dock. Its battery is old: it holds 30 units and has 10 left. Before it enters the ICU it must run its ultraviolet lamp for 20 seconds, and the lamp takes 2 units per second, 40 in all. Anyone who has used a phone while it charges knows what to do: plug the robot in and switch the lamp on. The charger gives 5 units per second, the lamp takes 2, and the battery goes up while the lamp runs. The model of Chapter 11 cannot find this plan. In it, a charge adds its energy when it ends, so the lamp would have to wait for a battery of 40 that this robot can never have.

This chapter gives numbers the ability to change *while* an action runs. PDDL 2.1 calls this a continuous effect: an effect that states the rate at which a number grows or shrinks, per second, for as long as the action lasts. With it, the battery is a line that goes up and down with a slope, several actions can push the same number at the same time, and a condition can ask about the value at any instant in between. Continuous change in this chapter always happens inside a durative action that the agent decides to run; a world that changes by itself, whether the agent acts or not, is the subject of Chapter 17.

We start with the lamp and see exactly where a model of jumps goes wrong. Then we meet the construct on the simplest example there is, a tank and a tap, and return to the lamp to solve it. Three small domains follow, each with one lesson: a greenhouse where water enters and leaves the tank at once, the generator, where the question is *when* to start an action, and an oven, where the question is when to stop one. After a look at what all this asks of a planner, and at how Kairos answers, the courier world gets a battery that drains along the way and charges for as long as the planner decides. The chapter closes with the questions that continuous change adds to the MODEL method, and with a comparison that says when a jump is still good enough.

12.1 A staircase where the world has a ramp 256
12.2 Continuous effects . . . 258
12.3 Reading a number while it changes 260
Water that comes and goes: the greenhouse . . 262
12.4 The generator: when to start 265
12.5 The oven: when to stop 267
12.6 What the planner must find 270
12.7 The courier world: a battery that drains and charges 271
Exercising the model . . 272
Where to charge, and for how long 274
12.8 The MODEL method: continuous change . . . 275
Alternatives compared . 276
12.9 Common mistakes . . . 277
12.10 Summary 278
12.11 Further reading 279
12.12 Exercises 279

12.1. A staircase where the world has a ramp

Let us write the scene at the dock with the tools of Chapter 11. There are two actions. `charge` lasts as long as the planner wants, and adds to the battery, when it ends, the charge rate times `?duration`. `sanitize` runs the lamp for 20 seconds. The lamp must not find the battery empty half way, and in a model where numbers only jump there is one safe way to guarantee it: ask for the whole energy before starting, and take it away at the start. Listing 12.1 shows both actions.

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

Listing 12.1: The scene at the dock with discrete changes (the two actions of `lamp-discrete-domain.pddl`).

The problem is the one of the story: 10 units of battery, a capacity of 30, a charger of 5 per second and a lamp of 2 per second.

```
$ 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 plan, and for this model rightly so: `sanitize` asks for 40 units at its start and the battery holds 30. (Why the answer is `no plan found` and not `unsolvable` is a matter for Section 12.6.) The model is careful, and its care has lost a solution that exists in the world.

We could be less careful. Suppose the lamp is paid at its end, like the charge, and the action only checks at its start that the battery is not empty. That is the file `lamp-careless-domain.pddl`, which differs from the listing in two lines of `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]
```

Now there is a plan, and it is worse than no plan: the robot switches the lamp on without plugging in, and the model sees nothing wrong until the last instant, when the battery jumps from 10 to −30. Nothing in the model looks at the battery then. In the world, the battery is empty after five seconds and the lamp goes out (Figure 12.1).

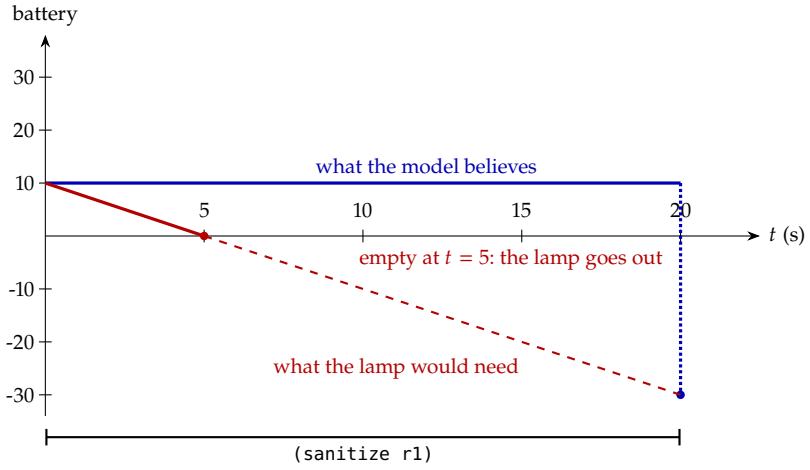


Figure 12.1.: The plan of the careless model. In the model the battery is a staircase with one step, at the end of the action. In the world it is a ramp that reaches zero after 5 seconds. Look at the 15 seconds in which model and world disagree: no condition of the model can see them.

Both models have the same defect, seen from two sides. In them the battery has a value at the start of an action and a value at its end, and nothing in between: a **discrete change**, one jump at one instant. The careful model places the jumps where they can do no harm (energy leaves early, arrives late) and so refuses plans in which energy is used as it arrives. The careless model places a jump where it hides the trouble and so accepts plans that fail. Neither can say what is true: that the battery goes down two units every second while the lamp is on, and up five every second while the charger is connected.

With discrete changes a number has no value in the middle of an action. Placing the jump early loses plans; placing it late accepts plans that fail.

Discrete and continuous change

A **discrete change** happens at an instant: before it the number has one value, after it another. All the numeric effects of Chapter 8, and those of durative actions under `at start` and `at end`, are discrete. A **continuous change** happens during an interval: the number passes through every value between the first and the last. What a model states about a continuous change is its **rate**: how much the number changes per unit of time. If the rate does not vary while the action runs the change is **linear**, and its graph is a straight segment. All the changes of this chapter are linear.

12.2. Continuous effects

A tank and a tap are enough to see the construct alone. The tap lets in a fixed number of liters per second. The action `fill` opens it, lets the water run for a time that the planner chooses, and closes it (Listing 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)

  (:types tank)

  (:predicates
    (tap-closed ?t - tank))

  (: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
```

```
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))))))
```

Listing 12.2: A tank filled by a tap: the level rises while the action runs.

Everything in the file is known from Chapter 11 except line 28:

```
      (increase (level ?t) (* #t (flow ?t)))
```

This is a **continuous effect**. Three things tell it apart from the effects we know.

- It has no `at start` and no `at end` around it. It does not happen at an instant: it is at work during the whole action, from its start to its end.
- Its amount is a product with the symbol `#t`. The symbol stands for the time that has passed since the action started, so `(* #t (flow ?t))` reads "the flow, for every second elapsed". After s seconds of an action that started with a level l_0, and if nothing else touches the tank, the level is $l_0 + s \cdot \mathit{flow}$. The factor that multiplies `#t` is the rate.
- It can only be an `increase` or a `decrease`. A number cannot be assigned "continuously", and the scaling effects have no continuous form.

The requirement is `:continuous-effects`. A domain that declares it also needs `:durative-actions` and `:numeric-fluents`, as any domain with durative actions and numbers does. The product can be written in either order, `(* #t rate)` or `(* rate #t)`, and the rate can be any numeric expression: a constant, a function such as `(flow ?t)`, or arithmetic over several.

A continuous effect has no time label. It states a rate: how much the number grows or shrinks per second while the action runs.

The duration of `fill` is an inequality, as in Chapter 11: at least one second, and at most the time that fills the tank, computed from the level at the start. The problem `tank-p01.pddl` has an empty tank of 100 liters, a tap of 2 liters per second, and the goal `(>= (level t1) 100)`. First the check, which reports how much of each feature Kairos plans with, then the 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]
```

One action of 50 seconds: 2 liters per second for 50 seconds are the 100 liters. The plan looks the same as a plan of Chapter 11 would, and for this problem a discrete effect at the end, `(at end (increase (level ?t) (* ?duration (flow ?t))))`, would have given it too. The difference is in what the model knows about the 50 seconds. With the discrete effect the tank is empty until the last instant. With the continuous one the level at second 10 is 20 liters and at second 30 it is 60 (Figure 12.2), and the next section puts that knowledge to use. The two long lines of the check, about durations and continuous effects, say in short how Kairos plans them; Section 12.6 explains them.

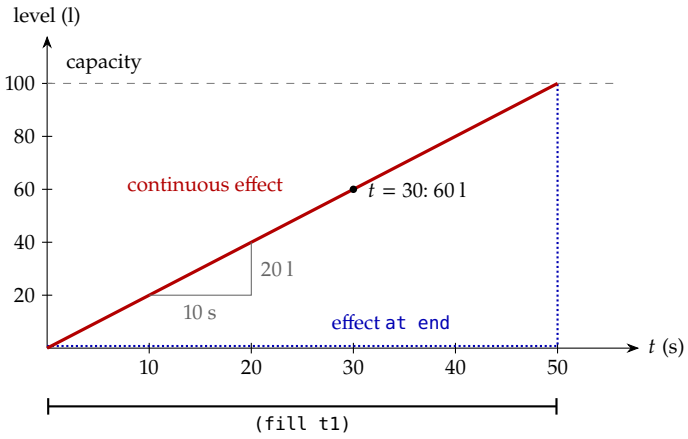


Figure 12.2.: The level of the tank during the plan. The slope of the solid line is the rate, 2 liters per second. The dotted line is what a discrete effect at the end would describe: the same first and last values, and nothing true in between.

The levels of PDDL 2.1

PDDL 2.1, by Maria Fox and Derek Long Fox and Long, 2003, was defined in levels. Level 1 is the classical language of Part II, level 2 adds numbers, level 3 durative actions whose effects happen at their start and end, and level 4 durative actions with continuous effects, the subject of this chapter. A fifth level, with changes that the world starts by itself, was left out of the language and became PDDL+ Fox and Long, 2006, which we study in Chapter 17. Planners followed the same order: many more plan with level 3 than with level 4.

12.3. Reading a number while it changes

A continuous effect earns its place when somebody looks at the number in the middle of the action. That is the situation of the lamp, and now we can write it (Listing 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)))))
```

Listing 12.3: The scene at the dock with continuous effects (the two actions of `lamp-continuous-domain.pddl`).

Compare it with Listing 12.1. In `sanitize`, the lamp no longer asks for 40 units in advance. It states two facts of the world: the battery goes down at the lamp's rate while it is on (line 46), and it must never be negative while the lamp is on (line 43). In `charge`, the energy arrives all along the action (line 34), and the limit of the battery is not a bound on the duration any more but a condition `over all` (line 31). Both conditions are comparisons on a number that is moving, and `over all` means for them what it meant for facts in Chapter 10: at every instant between the start and the end of the action.

```
$ 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]
```

The plan that the careful model could not find, in 20 seconds, the duration of the lamp alone. Let us read it with a pencil. From 0 to 5 only the lamp runs: the battery goes from 10 to 0. At 5 the charger joins. Now two continuous effects act on the same number, and their rates add: $+5-2=+3$ per second. In the 10 seconds of the charge the battery climbs from 0 to 30. At 15 the charge ends, the lamp is alone again, and in its last 5 seconds the battery goes down to 20 (Figure 12.3).

Continuous effects of several actions on the same number add up: at each instant the number moves at the sum of the rates of the actions that are running.

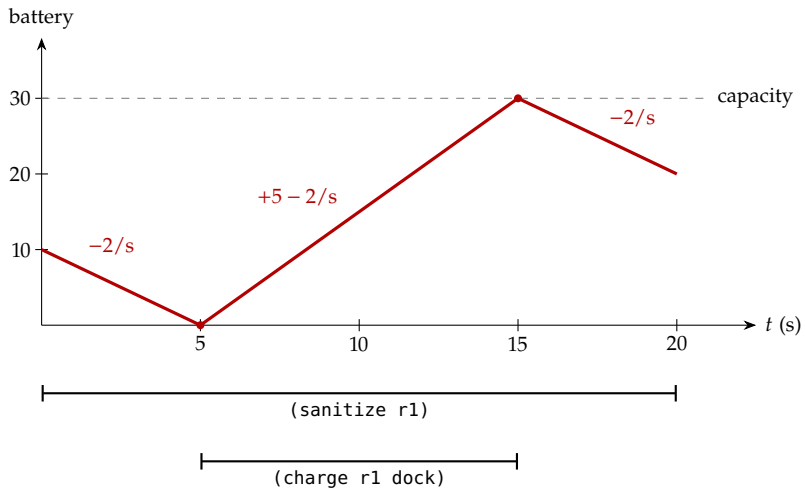


Figure 12.3.: The battery during the plan of the continuous model. The slope changes only where an action starts or ends. Look at the two dots: the plan touches the empty battery at $t = 5$ and the full one at $t = 15$, and both conditions, written with >= and <=, allow it.

The plan is correct and uncomfortable: it lets the battery reach exactly zero before plugging in, and fills it exactly to the top. The conditions say `(>= (battery ?r) 0)` and `(<= (battery ?r) (battery-capacity ?r))`, and a planner takes a model at its word. A modeler who wants a reserve writes the reserve: `(>= (battery ?r) 5)`. The plan of the opening of the chapter, charger and lamp switched on together, is also a plan of this model. It is in the file `lamp-together.plan`, and `kairos validate` accepts it. The plan of the careless model, the lamp alone, is in `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
```

The message names the condition, the action that owns it, and the value that broke it: at the end of the interval that the validator was checking, the battery would be −30.

How a moving number is checked

Between two consecutive happenings of a plan (starts and ends of actions) the set of running actions does not change, so every number moves along a straight segment. A comparison such as `(>= (battery ?r) 0)` can then only become false inside the segment if it is false at one of its ends: a straight line that is above zero at both ends is above zero all along. A validator, or a planner, has to look only at the ends of each segment. This is what makes linear change tractable, and it is the reason why the messages of the validator speak of "the happening at" some time.

Water that comes and goes: the greenhouse

The same pattern appears wherever something is stored, used and refilled. In a greenhouse, a tank fed by a tap supplies the drip lines of the beds. Watering a bed takes 40 seconds, and during them the bed drinks from the tank at a constant rate (Listing 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
```

```
20
21  ; The tap may run while the beds drink: how long it can stay
22  ; open is not known at its start, so the limit is a condition
23  ; on the level all along.
24  (:durative-action fill
25    :parameters (?t - tank)
26    :duration (and (>= ?duration 1) (<= ?duration 120))
27    :condition (and
28      (at start (tap-closed ?t))
29      (over all (<= (level ?t) (capacity ?t))))
30    :effect (and
31      (at start (not (tap-closed ?t)))
32      (increase (level ?t) (* #t (flow ?t)))
33      (at end (tap-closed ?t))))
34
35  ; Watering a bed takes 40 s, and the tank must not run dry.
36  (:durative-action irrigate
37    :parameters (?b - bed ?t - tank)
38    :duration (= ?duration 40)
39    :condition (and
40      (at start (dry ?b))
41      (over all (feeds ?t ?b))
42      (over all (>= (level ?t) 0)))
43    :effect (and
44      (at start (not (dry ?b)))
45      (decrease (level ?t) (* #t (demand ?b)))
46      (at end (watered ?b)))))
```

Listing 12.4: The greenhouse: a tap fills the tank while the beds drink from it.

The action `fill` of the lonely tank had a duration bound that computed, at the start, how long the tap could stay open: `(/ (- (capacity ?t) (level ?t)) (flow ?t))`. Here that computation would be wrong. If a bed is drinking, the tank takes longer to fill than the level at the start suggests, and how much longer depends on what else happens during the action. So the bound is a generous constant, and the real limit is stated as what it is, a condition on the level `over all` (line 29). This is a general rule: *a bound computed at the start of an action is only right if nothing else changes the number meanwhile.*

In `greenhouse-p01.pddl` the tank holds 30 liters and has 5, and one bed drinks 1 liter per second: 40 liters in all, more than the tank can hold. Filling first and watering later cannot work. The plan must do both at once:

```
$ 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]
```

The shape is the one of the lamp. The bed drinks alone for 5 seconds and takes the level from 5 to 0; the tap opens and for 30 seconds the level rises at $2 - 1 = 1$ liter per second, up to 30; the tap closes at 35 and the last 5 seconds of watering leave 25 liters (Figure 12.4). The note under the plan appears whenever Kairos plans in steps of time, as it did here (Section 12.6): it found a plan with few actions, and reports the makespan without having searched for the shortest one. We leave the note out of the next listings.

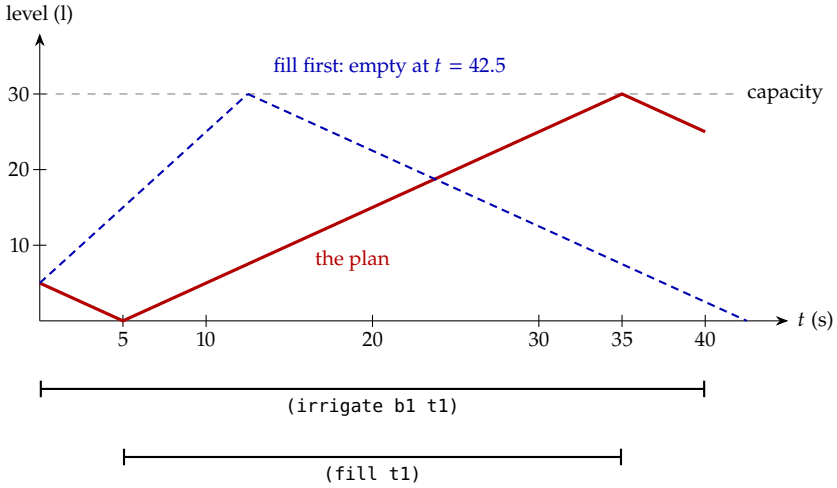


Figure 12.4.: The level of the tank in the plan of `greenhouse-p01` (solid). The dashed line is the plan that fills the tank to the top and waters afterwards: 30 liters do not last 40 seconds.

The plan "fill first" is in the file `greenhouse-first.plan`: a fill of 12.5 seconds and then the watering. The validator finds the place where the tank runs dry:

```
$ 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
```

With two beds on the same tank (`greenhouse-p02.pddl`, each bed drinking half a liter per second) three actions move the level at the same time:

```
$ 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]
```

In the second plan the level starts at 5 and rises at 2 liters per second with the tap alone, at 1.5 when the first bed joins, and at 1 with both beds, until the tap closes at 24 with 29.95 liters. Then it falls at 1 liter per second while both beds drink. The start times 0.9 and 1.0 are not round by chance: they come from the steps of a tenth of a second in which Kairos advanced time to find this plan.

Steps O and D on the greenhouse

Objects and state. The level of the tank is a quantity (Chapter 8). The new question is how it changes: not in jumps but along the actions, at rates that are static numbers (`flow`, `demand`). The state table gets a column for it.

Property	How it changes	Encoding
`(level ?t)`	while `fill` runs, +flow per second; while `irrigate` runs, −demand per second	numeric fluent
`(flow ?t)`, `(demand ?b)`	never	static functions

Dynamics. For each action that changes the level we ask who may be looking at the level meanwhile. `irrigate` looks itself (the tank must not run dry), and so does `fill` (it must not overflow), and each one runs while the other changes the level. That settles it: the changes are continuous effects and the limits are `over all` conditions.

12.4. The generator: when to start

In the examples so far the planner had a duration to choose. In the next one every duration is fixed and there is still something to find. The generator is a small domain that has accompanied continuous change in PDDL for a long time; the paper that presents the plan validator VAL uses it to show what plans with continuous effects look like Howey et al., 2004. A generator must run for a fixed time. It burns fuel at a constant rate, and its tank is too small for the whole run. Jerrycans can be poured into the tank while it runs. Listing 12.5 is our version, with a run of 100 seconds.

```
;; 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)))))
```

Listing 12.5: The generator: fuel burns during the run and jerrycans are poured in meanwhile.

The two actions pull the fuel in opposite directions and each watches its own limit: `generate` must never find the tank empty, `refuel` must never overflow it. A jerrycan is poured at 2 liters per second for 10 seconds, 20 liters. In `generator-p01.pddl` the tank holds 100 liters and has 95, and there is one jerrycan. The run needs 100 liters, so the jerrycan is necessary, and 115 liters do not fit in the tank, so it cannot be poured before starting. It must be poured during the run. But not at any moment:

- Poured too early, it overflows. While both actions run the fuel rises at $2 - 1 = 1$ liter per second, so a pouring that starts at time s ends with $95 - s + 10$ liters. That fits in 100 only if $s \geq 5$.
- Poured too late, the generator has stopped. Alone, the generator empties the tank at $t = 95$.

Any start between 5 and 95 gives a plan. The planner has no duration to choose here, only an instant, and the right instants form a window that no single action states: it follows from two rates, two limits and the initial level.

```
$ 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 picks the last instant of the window. It lets time pass until a condition is about to break (the tank is empty at 95) and pours then; half of the jerrycan goes into a generator that has already finished, which the goal does not forbid. Figure 12.5 draws this plan and a more sensible one, with the pouring at $t = 50$, from the file `generator-middle.plan`.

With continuous change, the instant at which an action starts can decide whether a plan is valid, even when every duration is fixed.

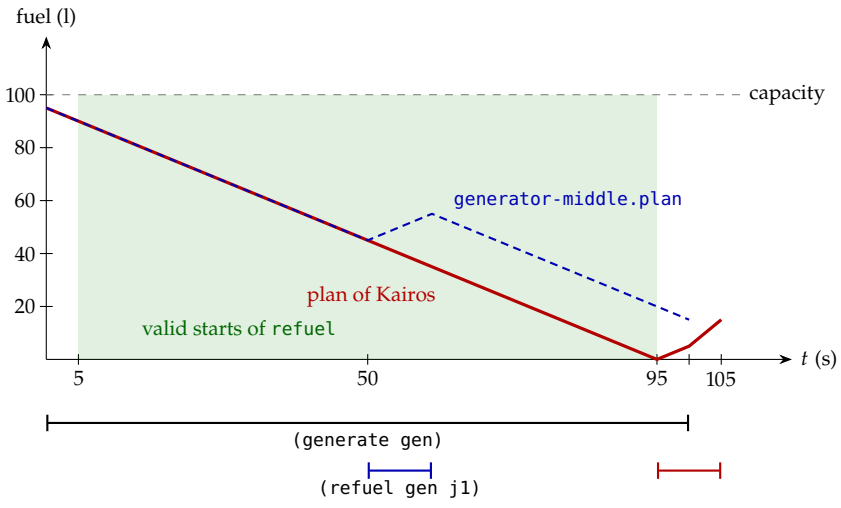


Figure 12.5.: Fuel in the tank for two valid plans of `generator-p01`. The shaded band is the window in which the pouring may start. Before $t = 5$ the line would cross the capacity; after $t = 95$ it has already crossed zero.

Three plans written by hand show the edges of the window. Pouring at the start of the run overflows, pouring before the run overflows more, and pouring in the middle is fine:

```
$ 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
```

With 65 liters and two jerrycans (`generator-p02.pddl`) one jerrycan fits before the run and the other does not:

```
$ 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]
```

The first pouring takes the tank to 85 liters, which last until $t = 95.001$, and there the second jerrycan arrives, again at the last instant.

The third problem has no plan: 60 liters and one jerrycan are 80 liters for a run that burns 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
```

The answer is `no plan found`, with exit code 14, and not the `unsolvable` with code 10 that we know from classical problems. The notes say why, and they are honest about a real limit: the search tried every instant that its steps of time can reach and found nothing, but time is continuous and the steps are not, so this is evidence and not a proof. Here we can supply the proof ourselves, with one subtraction.

12.5. The oven: when to stop

The generator asked for the instant to start an action of fixed length. An oven asks for the opposite: the instant to stop an action of free length. The heater raises the temperature 2 degrees per second for as long as it is on,

and each dish bakes for 10 seconds inside its own range of temperatures (Listing 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)))))
```

Listing 12.6: The oven: the heater runs for a chosen time, and a dish needs the temperature inside its range during the whole baking.

The duration of `heat` may be anything from 1 to 300 seconds. In `oven-p01.pddl` the oven is at 20 degrees and bread bakes between 200 and 220. Heating must therefore last between 90 and 100 seconds: neither bound of the action, but a value in between that no line of the domain mentions. It follows from the two conditions of `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]
```

The heater stops at 90 seconds, the first instant at which bread can go in. With a second dish, pizza, that bakes between 210 and 240 degrees (`oven-p02.pddl`), there are two plans:

```
$ 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]
```

The first plan heats twice. The second heats once, up to 210 degrees, the only temperature at which the ranges of both dishes can be served one after the other without touching the heater again (any value from 210 to 220 would do). Notice its second line: the bread goes in at 94.9, a tenth of a second *before* the heater stops, with the oven at 209.8 degrees and still rising. The condition of `bake` is checked on a temperature that is moving, and it holds (Figure 12.6).

When the duration of an action is free, the right moment to end it can be fixed by the conditions of other actions that read the number it changes.

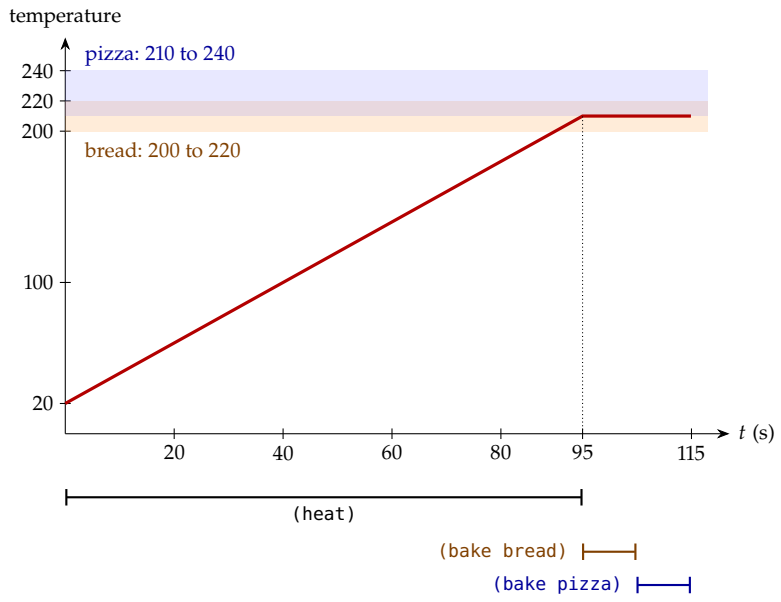


Figure 12.6.: The temperature in the second plan of `oven-p02`. The heater stops at 210 degrees, inside both ranges (the darker band where they overlap). A heating of 90 seconds, the first value that suits the bread, would leave the pizza out.

A cook in a hurry stops the heater at 80 seconds (`oven-short.plan`), and the validator shows the bread and the oven disagreeing by 20 degrees:

```
$ 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
```

Our oven does not lose heat when the heater is off, and nothing in it cools. Both are decisions of step M: the scope of this model is "heating up to bake", and a dish that needs a cooler oven than the present one has no plan. Exercise 12.5.2 adds a way to cool.

12.6. What the planner must find

A classical planner chooses which actions and in which order. A temporal planner, since Chapter 10, also schedules them, and since Chapter 11 it may choose durations. But until now every choice of time could be made late and cheaply: once the actions and their order were fixed, any schedule that respected the order and the durations was good. Continuous change breaks this. In the generator, the same two actions in the same order are a plan or not depending on a number, the start of the pouring, and the valid numbers are those of an interval. In the oven it is a duration. The planner has to find real numbers, and the conditions that restrict them are spread over several actions.

For linear change there is a way out, which the concept box of Section 12.3 announced: since numbers move along straight segments between happenings, only the happenings matter, and the conditions on them are linear inequalities over the times of the plan. A planner can collect them and ask a linear programming solver for times that satisfy them all. This is the approach of planners in the family of POPF Coles et al., 2010. Other planners, and Kairos is one, combine simpler devices.

How Kairos plans continuous change

Kairos plans a task with continuous effects in two ways, and validates whatever either of them finds.
As discrete changes. When a number that changes continuously is only read as "large enough" or only as "small enough", each continuous effect is replaced by one jump of its total amount, placed at the safest instant for the conditions that read the number: what drains leaves at the start of the action, what fills arrives at its end. Each duration given by an inequality is tried at its two bounds. This is our careful model of Section 12.1, built by the planner, and the ordinary temporal engines solve it.
In steps of time. The task is also searched by letting time pass: a step of 0.1 seconds, or a jump to the instant at which some numeric condition is about to change. An action of free duration then ends at a step, or at the instant where ending it makes a condition of some action true. This finds plans in which actions push a number against each other, and durations between the bounds.
Validation. Every plan is checked with the real continuous semantics, segment by segment, before it is printed.

With the option `-v` the planner says which way it took. The courier of the next section is planned as discrete changes, with the steps of time beside them; the generator, where `refuel` needs the fuel small enough and `generate` needs it large enough, only in steps:

```
$ 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+" names the machinery, which Chapter 17 describes; the model is still the one we wrote.) Three consequences of this way of planning are visible in the outputs of the chapter, and it is better to know them than to be surprised by them.

- **Durations come at their bounds, or where a condition fixes them.** The oven was heated for exactly 90 seconds because at that instant ending `heat` made `bake` possible. A duration that nothing fixes comes out as the shortest or the longest allowed. We will see the courier charge for 5 seconds, the minimum, or until full.
- **Plans wait until the last moment.** The jump "to where a condition is about to change" is why the jerrycan is poured at $t = 95$ and the tap of the greenhouse opens when the tank is empty.
- **Without a plan, the answer may be `no plan found`.** Steps of time cannot visit every real number, so running out of states proves nothing. Kairos then says `no plan found` (exit code 14) and keeps `unsolvable` (exit code 10) for what it has proved.

The third point deserves a habit. When a model with continuous effects has no plan, and you believe it should, write the plan by hand and give it to `kairos validate`. The validator works in continuous time and its verdict is exact for linear change: if it accepts your plan, the plan is one, whatever the search said.

Portability. Continuous effects are among the least supported features of PDDL. In the check that the documentation of Kairos keeps, one small problem with ten seconds per planner, only POPF returned a valid plan besides Kairos; OPTIC, TFD, LPG-td, ENHSP, Metric-FF and LAMA did not.

Planning with continuous change means finding instants and durations, not only actions. Kairos tries the bounds and the instants where conditions change, and validates every plan in continuous time.

12.7. The courier world: a battery that drains and charges

The courier of Chapter 11 paid for a move when it started and was paid for a charge when it ended. We now give it the battery of this chapter: it goes down all along a move and up all along a charge, and the planner chooses for how long to charge.

Step M: what changes in the model card

Scope, in: the charge of the battery at every instant, not only between actions. **Scope, out:** consumption while the robot stands still, picks or drops; a discharge that depends on the load or on the charge left. **Assumptions:** the battery drains at a constant rate while the robot moves and fills at a constant rate while it is plugged in; a robot is plugged in only at a room with a charger; a charge of less than 5 seconds is not worth docking for. The state table gains two static functions from the shared vocabulary of the courier world: `(drain-rate ?r)` and `(charge-rate ?r)`, both in battery units per second.

The domain is the typed courier world with the durative actions of Chapter 10: `move` lasts the distance over the speed, `pick` and `drop` last 2 seconds, and `(idle ?r)` lets the robot do one thing at a time. Listing 12.7 shows the functions and `move`, and Listing 12.8 shows `charge`.

```
18  (:functions
19    (distance ?from ?to - location)  ; meters
20    (speed ?r - robot)               ; meters per second
```

```
21    (battery ?r - robot)             ; charge, 0 to 100
22    (battery-capacity ?r - robot)
23    (drain-rate ?r - robot)          ; per second, moving
24    (charge-rate ?r - robot))        ; per second, charging
25
26  ; The battery goes down all along the way, and must never
27  ; reach a negative value on it.
28  (:durative-action move
29    :parameters (?r - robot ?from ?to - location)
30    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
31    :condition (and
32      (at start (robot-at ?r ?from))
33      (at start (idle ?r))
34      (over all (connected ?from ?to))
35      (over all (>= (battery ?r) 0)))
36    :effect (and
37      (at start (not (robot-at ?r ?from)))
38      (at start (not (idle ?r)))
39      (decrease (battery ?r) (* #t (drain-rate ?r)))
40      (at end (robot-at ?r ?to))
41      (at end (idle ?r))))
```

Listing 12.7: The functions and the action `move` of `courier-battery-domain.pddl`.

```
72  ; Charge for as long as the planner wants: at least 5 s, at
73  ; most until the battery is full.
74  (:durative-action charge
75    :parameters (?r - robot ?l - location)
76    :duration (and
77      (>= ?duration 5)
78      (<= ?duration (/ (- (battery-capacity ?r) (battery ?r))
79                       (charge-rate ?r))))
80    :condition (and
81      (at start (idle ?r))
82      (at start (has-charger ?l))
83      (over all (robot-at ?r ?l)))
84    :effect (and
85      (at start (not (idle ?r)))
86      (increase (battery ?r) (* #t (charge-rate ?r)))
87      (at end (idle ?r)))))
```

Listing 12.8: The action `charge` of `courier-battery-domain.pddl`.

`move` states the drain as a continuous effect (line 39) and the limit as a condition on the whole way (line 35). It does not compute how much battery the move needs: that follows from the rate and the duration, and the validator and the planner work it out. `charge` is the lonely tap of Listing 12.2: its upper bound is the time to a full battery, computed at the start. By the rule of Section 12.3 that is right only if nothing else changes the battery during a charge, and here nothing can: the charge takes `(idle ?r)`, so the robot cannot move meanwhile, and only moves drain.

Exercising the model

Step E asks for four kinds of problems. All use the map and the distances of the courier world, one robot with a speed of 1 meter per second, a drain rate of 1 and a charge rate of 5. At that speed a meter costs one unit of battery.

A trivial one. In `courier-battery-p01.pddl` the battery is full. The plan is the one of Chapter 10, 42.006 seconds without a charge, and the 38 meters leave 62 units.

A typical one. In `courier-battery-p02.pddl` the robot starts at the dock with 20 units and the same delivery, which needs 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]
```

A charge of 5 seconds, the shortest allowed, takes the battery from 20 to 45, and the trip brings it down to 7 (Figure 12.7). The exact need was 18 units, 3.6 seconds of charge, but the action does not allow less than 5.

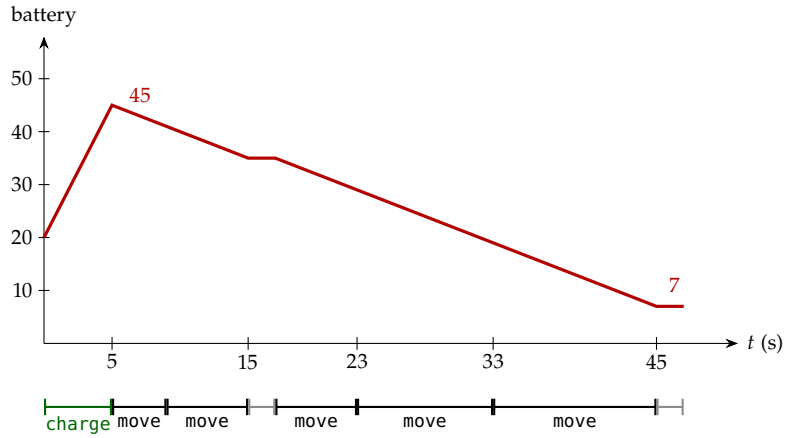


Figure 12.7.: The battery of `r1` during the plan of `courier-battery-p02`. It rises 5 units per second during the charge and falls 1 per second during each move. Look at the two flat pieces, the pick and the drop (gray): the model says that they cost nothing.

One that must have no plan. In `courier-battery-p04.pddl` the robot is in the lobby with 3 units. The dock is 4 meters away.

```
$ 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)
```

Thirty-five states were enough to run out of things to try: with 3 units the robot cannot finish any move.

One that tempts the planner to cheat. In `courier-battery-p05.pddl` the robot is in the lobby with 30 units and the delivery from there is 34 meters long. A model that looked at the battery only at the start of each move would let the last move begin with 8 units for 12 meters. That plan is in `courier-battery-cheat.plan`, and our model rejects it at the right place:

```
$ 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]
```

The honest plan takes the medicines, comes back through the lobby, turns to the dock with 14 units left, charges 5 seconds and delivers with 13 to spare (Figure 12.8).

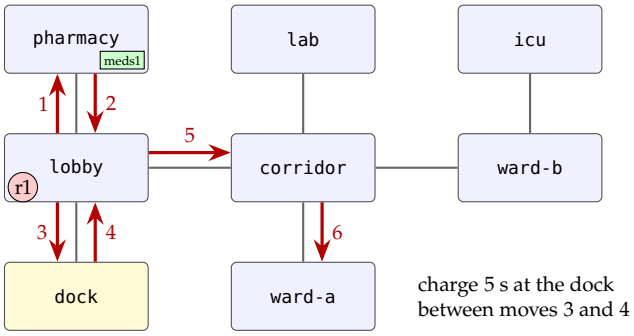


Figure 12.8.: The plan of the cheat test `courier-battery-p05`. The straight way (moves 1, 2, 5 and 6) is 34 meters long and the battery lasts 30. The detour to the dock adds 8 meters and a charge.

Where to charge, and for how long

The last problem is the one for which the chapter was written. In `courier-battery-p03.pddl` the robot starts at the dock with 60 units and has a round of 90 meters: medicines from the pharmacy to the ICU, a sample from the ICU to the lab, and back to the dock. It must charge, and it passes by the dock more than once. Kairos prints four plans in a few hundredths of a second and then keeps searching for a better one until it has tried everything, which here takes several seconds; we stop it at three with `-timeout`. The listing keeps the first lines of each plan and its charges.

```
$ 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]
```

The four plans are four answers to "where and for how long". The first three fetch the medicines, come back to charge, deliver, and come back to charge again, with charges of every kind: the minimum of 5 seconds, two minimums in a row, and durations that are no bound of anything, 19.4 and 17.8 seconds, found in steps of time. The last plan sees that one charge at the beginning is enough: 8 seconds, from 60 up to a full battery, the upper bound of the action. The round uses 90 units and the robot is back at the dock with 10.

Is it the best plan? Six seconds of charge would have been enough, and the plan would be two seconds shorter. Kairos does not find that duration: nothing in the model makes "90 units" a condition that ending the charge satisfies; it is the sum of the needs of fifteen later moves. The plan is valid, and close to the best. What we can say about the model is more useful than what we can say about the planner: the lower bound of 5 seconds, which looked like a detail of step M, is what shapes the first three plans. A bound on a free duration is a modeling decision with consequences, and Exercise 12.6.2 looks at them.

The courier charges for a time that the planner chooses. The plans are valid for the continuous battery; the durations come at their bounds or from steps of time, so the last plan may charge more than strictly needed.

12.8. The MODEL method: continuous change

Continuous effects add questions to two steps of the MODEL method, and the answers decide whether the feature is needed at all.

New questions for steps O and D

Objects and state. For every quantity of the state table:

- Does it change *while* an action runs, or only when actions start and end?
- If it changes meanwhile: at what rate, and is the rate constant?

Dynamics. For every action that changes such a quantity:

- Does anybody read the quantity while the action runs: a condition of another action that may run at the same time, or a limit that the action itself must respect all along?
- Can another action change the same quantity at the same time?
- Is the duration a decision? Then what bounds it: a constant, a value computed at the start (right only if nobody else changes the quantity), or a condition `over all`?

If nobody reads the quantity meanwhile and nobody else changes it, a discrete change is enough: at the start if the action consumes, at the end if it produces.

The last sentence of the box is the important one. Continuous effects are expensive: few planners accept them, and those that do search harder. They should be the answer to a need of the problem, and the need is always the same: somebody looks at the number in the middle.

Alternatives compared

The scene at the dock gave us three encodings of the same battery, all run in this chapter: the careless discrete model (the lamp is paid at its end), the careful discrete model (paid at its start, the charge credited at its end), and the continuous model. Table 12.1 judges them with the five criteria.

Table 12.1.: Three encodings of a battery that is used while it charges.

	Careless discrete	Careful discrete	Continuous
Fidelity	Impossible plans: the lamp alone, battery at −30	No impossible plans; loses every plan that uses energy as it arrives (no plan for the old battery)	Exact for constant rates
Size	2 ground actions	2 ground actions	2 ground actions
Guidance	Easy, and wrong	Ordinary temporal search	The planner must also find instants and durations; without a plan, often no proof
Clarity	Hides the rule about the battery	The modeler computes totals by hand (`(* 20 (lamp-rate ?r))`)	States rates and limits as they are in the world
Portability	Any temporal planner with numbers	Needs `?duration` in effects (Kairos, OPTIC, POPF)	Kairos and POPF

Verdict. The careless model is never the right choice: a wrong plan costs more than no plan. Between the other two, the question of the method

box decides.

- *Choose the careful discrete model* when each quantity is changed by one action at a time and read by nobody meanwhile. The courier of this chapter is such a case: a robot that does one thing at a time, with a battery that only its own moves and charges touch. Its continuous model and a careful discrete one have the same plans, and Kairos itself turns the first into the second to plan it. The continuous version is still the clearer text, since it states a rate where the discrete one states a product.
- *Choose the continuous model* when producers and consumers of the same quantity may overlap (the lamp and the charger, the tap and the beds, the generator and the jerrycan), or when a limit must hold in the middle of a long action and the action cannot know at its start what else will happen.
- *Go beyond both*, to the processes of Chapter 17, when the quantity changes even if the agent does nothing: a battery that drains whenever the robot is switched on.

12.9. Common mistakes

`#t` under a time label

A continuous effect is not `at start` nor `at end`. Writing the label out of habit, as in `tank-wrong-domain.pddl`,

```
      (at end (increase (level ?t) (* #t (flow ?t)))))))
```

is an error that the check reports:

```
$ 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
```

If the amount at the end is what you want, the discrete effect of Chapter 11 says it with `?duration`: `(at end (increase (level ?t) (* ?duration (flow ?t))))`. The same message appears for `#t` in an `assign` or in an instantaneous action.

A limit checked only at the start

A continuous effect without a condition that reads the number all along changes nothing: the number moves and nobody looks. The condition `(at start (>= (battery ?r) 0))` is true of almost any battery and says nothing about the way. The symptom is a plan that is accepted and should not be, as with the careless model of Section 12.1:

```
$ kairos validate lamp-careless-domain.pddl lamp-careless-p01.
     pddl lamp-alone.plan
plan valid, makespan 20
```

A limit that must hold during an action is an `over all` condition. The cheat test of step E is the tool that finds its absence: a problem whose only short plan breaks the limit in the middle.

Reading `no plan found` as "there is no plan"

With continuous effects Kairos often cannot prove that a problem has no plan, and it says so:

```
$ kairos generator-domain.pddl generator-p03.pddl
...
kairos: no plan found (exhausted)
```

`unsolvable` (exit code 10) is a proof. `no plan found` (exit code 14) means that the search ended without a plan and without a proof; the notes before it tell which part of the search was incomplete. Scripts that test a model should expect the code that matches, and a modeler who doubts should write the plan by hand and validate it.

A requirement left out

A domain that uses `#t` without declaring `:continuous-effects` is planned by Kairos, with a warning that `kairos check` shows:

```
warning: uses continuous-effects but :requirements does not
     include :continuous-effects [semantic.
     undeclared_requirement]
```

Other planners read the requirements to decide whether they can plan the domain at all, and some stop at the first construct that was not declared. Declare what the file uses.

12.10. Summary

A number that only jumps at the start or at the end of an action has no value in between. Continuous effects give it one. They state a rate, they add up when several actions run, and conditions `over all` can watch the number while it moves. The price is paid by the planner, which must find instants and durations as well as actions, and by the modeler, who has fewer planners to choose from and weaker answers when there is no plan.

Table 12.2.: What this chapter added.

Construct or idea	What to remember
`:continuous-effects`	The requirement; with `:durative-actions` and `:numeric-fluents`
`(increase (f) (* #t r))`	While the action runs, `f` grows `r` per second; no time label; only `increase` and `decrease`
Several effects on one number	The rates add
`(over all (>= (f) 0))`	Holds at every instant of the action; for linear change it is enough to look where actions start and end
Bound of a free duration	A constant, a value computed at the start (only if nobody else changes the number), or an `over all` condition
What the planner finds	Which actions, when each starts, how long each lasts
Kairos	Bounds of durations and safest jumps, plus steps of time; every plan validated in continuous time; `no plan found` is not a proof
When to use it	Somebody reads the number, or changes it too, while the action runs

12.11. Further reading

Continuous effects are defined in the paper of PDDL 2.1 Fox and Long, 2003: its sections on level 4 give the syntax of `#t` and the semantics of plans in which numbers change during the actions, and discuss why the language separates discrete from continuous effects. The validation of such plans, with the generator as the example, is described with the tool VAL Howey et al., 2004. For planning, the forward search of POPF Coles et al., 2010 is the reference for linear continuous change in a planner that also handles the required concurrency of Chapter 11; OPTIC Benton et al., 2012 extends that line of planners towards plan quality. The book by Haslum and colleagues Haslum et al., 2019 places continuous effects in the whole family of PDDL versions. The way Kairos plans them, which Section 12.6 summarized, is documented with the planner Martín Rico et al., 2026. Change that the world starts by itself, with processes and events, is the subject of PDDL+ Fox and Long, 2006 and of Chapter 17.

12.12. Exercises

Exercises for Section 12.1 — A staircase where the world has a ramp

Exercise 12.1.1. A robot with 50 units of battery makes a move of 30 seconds that drains 1 unit per second, and during that move, from second 10 to second 20, a camera on the robot takes pictures, which needs the battery at 35 or more. Say whether the camera has enough battery (a) in the world, (b) in a model where the move takes its 30 units at the start, (c) in a model where it takes them at the end. Which of the two models is wrong in the dangerous direction?

Exercise 12.1.2. In `lamp-discrete-p01.pddl`, change the capacity of the battery to 100 and run the careful model. Read the plan and compare its makespan with the 20 seconds of the plan of Section 12.3. What did the careful model lose this time?

Exercises for Section 12.2 — Continuous effects

Exercise 12.2.1. Add to the tank of Listing 12.2 a drain valve: an action `drain` that lowers the level at `(outflow ?t)` liters per second for a time that the planner chooses, at most until the tank is empty. Write a problem in which a tank with 80 liters must be emptied, with an outflow of 4 liters per second, and predict the plan before running it.

Exercise 12.2.2. For each of these effects of a durative action, say whether it is a correct continuous effect, a correct discrete effect, or an error, and check your answers with `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))))`.

Exercise 12.2.3. The tank of `tank-p01.pddl` is filled by one action of 50 seconds. Change the goal to `(>= (level t1) 80)`, run the planner and explain the last plan it prints. Then say what you would change in the *domain* if a tap cannot be opened and closed every second.

Exercises for Section 12.3 — Reading a number while it changes

Exercise 12.3.1. In the plan for `greenhouse-p01.pddl` the tap opens at $t = 5$ for 30 seconds. By hand, show that no other start is valid for a fill of that length. Then find all the instants at which a single `fill` of 20 seconds could start, and check the two ends of your answer with `kairos validate`.

Exercise 12.3.2. The gardener wants 2 liters to remain in the tank at all times. Change the greenhouse domain, run `greenhouse-p01.pddl` again and explain how the plan moved.

Exercise 12.3.3. In the plan of Section 12.3 for the lamp, the charge starts at $t = 5$. Would a charge of 10 seconds starting at $t = 0$ be a plan? And one of 6 seconds starting at $t = 0$? Answer with the graph of the battery, then validate.

Exercises for Section 12.4 — The generator: when to start

Exercise 12.4.1. The tank of the generator of `generator-p01.pddl` now starts with 90 liters instead of 95. Compute the window of valid starts for the pouring, and say whether the jerrycan could be poured before the run.

Exercise 12.4.2. In the plan for `generator-p02.pddl` the second jerrycan is poured at 95.001. Give another valid plan in which no pouring is in progress when the generator stops, and validate it.

Exercise 12.4.3. A safety rule forbids pouring fuel into a running generator. Write the rule into the domain. Which of the problems `p01` and `p02` (the second with 60 liters in the tank) still has a plan, and why?

Exercises for Section 12.5 — The oven: when to stop

Exercise 12.5.1. In `oven-p01.pddl`, which durations of `heat` lead to a valid plan if the bread goes in when the heater stops? And if the bread may go in while the heater is still on, as in `oven-overlap.plan`? Validate that plan and explain why it is accepted.

Exercise 12.5.2. Give the oven a way to cool: an action `cool`, with the door open and the oven empty, that lowers the temperature at `(cool-rate)` degrees per second for a chosen time. Solve a problem in which the oven is at 150 degrees and a meringue bakes between 100 and 120.

Exercises for Section 12.6 — What the planner must find

Exercise 12.6.1. Run the first problem of the greenhouse and the second of the courier with `-v` and find in the output which of the two ways of the concept box Kairos used for each. Explain the difference from the conditions that read the level and the battery.

Exercise 12.6.2. A colleague removes the lower bound of `charge`, leaving only the upper one. What plans do you expect for the problem `p02` of the courier? Explain what the lower bound was doing, as a statement about the world and as a help for the search.

Exercises for Section 12.7 — The courier world

Exercise 12.7.1. The hospital wants the robots to keep a reserve: the battery must never go below 10 while moving. Change the domain

and run `p02` and `p05` again. Which plan changes and why?

Exercise 12.7.2. Draw, as in Figure 12.7, the battery of the robot during the last plan of `courier-battery-p03.pddl`. At which instant is the battery lowest? How many seconds of the first charge could be saved?

Exercise 12.7.3. In our model, picking and dropping cost no battery. Make them cost 0.5 units per second, as continuous effects. Does `pick` need an `over all` condition on the battery? Justify your answer with the question of the method box.

Exercises for Section 12.8 — The MODEL method: continuous change

Exercise 12.8.1. For each situation, answer the two questions (does the quantity change while an action runs, does anybody read or change it meanwhile) and choose between a discrete and a continuous effect: (a) a truck burns fuel on each trip and refuels only when parked; (b) a printer uses ink while printing, and a cartridge can be swapped only when it stops; (c) a water heater warms the water while somebody takes a shower from it; (d) a drone drains its battery while flying and while its camera records, and both can happen at once.

Exercise 12.8.2. Apply step L to the courier of this chapter: write the careful discrete version of `move` and `charge`, with `?duration` in the effect of the charge, and compare it with the continuous version on the five criteria. Which one would you give to a planner that does not support `:continuous-effects`?

13. Derived predicates and a world with a calendar

In every model so far, a fact is true for one of two reasons: the initial state says so, or an action made it true and no action has made it false since. Two situations of the hospital do not fit. The first: the floor has doors, and a robot at the dock wants to know whether it can get to the ICU. The answer is not written anywhere and no action sets it. It follows from which doors are open, and it changes, for many pairs of rooms at once, every time somebody opens or closes one door. The second: the pharmacy opens at eight and closes at two. No robot opens it. It opens because of the hour, and a courier that arrives early can only wait. Our models have no way to say "this is true because those other things are true", and no way to say "this will become true at that time, whatever anybody does".

PDDL 2.2 added one construct for each. A *derived predicate* is a fact defined by a rule over other facts: the planner works out its truth in every state, and no action touches it. A *timed initial literal* is a fact that the problem schedules to become true or false at a given time: a calendar that the world follows by itself. The two have nothing else in common, and that is why they share a chapter: they are the two ways in which a fact can change without an action changing it. One is a consequence of the state; the other is a consequence of the clock.

The first half of the chapter is about rules. We start with a fact that the courier world has maintained by hand since Chapter 2, `(free ?r)`, and turn it into a derived predicate; then come rules that call themselves, for rooms reachable through open doors, for towers of blocks and for electrical circuits; then what a planner does with rules, and what that costs. The second half is about the calendar: opening hours, deadlines and windows of visibility, on the courier and on a satellite, and numbers that change at known times. The method section adds one question to step O and one to step M, and compares a derived fact with the same fact kept by effects.

13.1 Facts that follow from other facts 282
13.2 Derived predicates . . . 283
Where a derived predicate may appear 284
13.3 Rules that call themselves 285
Two more recursions: towers and circuits . . . 287
Rules and negation . . . 290
13.4 What a planner does with rules 290
13.5 A world with a calendar: timed initial literals . . 291
13.6 Windows and deadlines 293
Deadlines 293
A satellite and its windows 295
13.7 Numbers on the calendar 296
13.8 The MODEL method: consequences and the calendar 297
Alternatives compared . 298
A deadline: timed literal or constraint 299
13.9 Common mistakes . . . 299
13.10 Summary 301
13.11 Further reading 301
13.12 Exercises 302

13.1. Facts that follow from other facts

Look at the typed courier domain of Chapter 4 with the eyes of the state table. It has two predicates about what a robot holds: `(carrying ?r ?i)` and `(free ?r)`. They are not independent. A robot is free exactly when it carries nothing; knowing every `carrying` fact, we know every `free` fact. The domain keeps both because `pick` needs to ask "does this robot carry nothing?", and in STRIPS a precondition can only ask for a fact. So `free` is stored, and the actions keep it in step with `carrying` by hand: `pick` deletes it, `drop` adds it, and the initial state must state it for every robot that starts with empty hands.

This works, and Chapter 7 named its risk: a redundant predicate that goes out of step. A problem whose initial state forgets `(free r1)` has a robot that can never pick. A new action that takes an item from a robot (say, a nurse unloads it) and forgets to add `free` leaves a robot that carries nothing and is not free. Nothing warns of either mistake: both states are perfectly good sets of facts.

With `free` the bookkeeping is light, one fact per action. For other facts it is hopeless. Take "the ICU can be reached from the dock". Opening one door may make it true for dozens of pairs of rooms, or for none, depending on all the other doors. An effect of `open-door` that kept such a fact up to date would have to contain the whole computation of paths in a graph.

The distinction we need is this. A **basic predicate** holds facts that are stated: by the initial state and by the effects of actions. A **derived predicate** holds facts that are computed: a rule says when they are true, in terms of other facts, and they are true in a state exactly when the rule says so.

Some facts are not decided by any action: they are consequences of other facts. Keeping them by hand in effects is fragile when easy and impossible when hard.

13.2. Derived predicates

Listing 13.1 is the typed courier domain with `free` turned into a derived predicate.

```
;; The typed courier world where (free ?r) is not kept by the
;; actions any more: a rule derives it from (carrying ?r ?i).
(define (domain courier-free)
  (:requirements :strips :typing :negative-preconditions
                 :existential-preconditions :derived-predicates)

  (: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))            ; derived: see the rule below

  ; A robot is free when there is no item that it carries.
  (:derived (free ?r - robot)
    (not (exists (?i - item) (carrying ?r ?i))))

  (: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)))

  ; pick and drop no longer mention (free ?r) in their effects
  (: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))
                 (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))))
```

Listing 13.1: The courier domain with `(free ?r)` defined by a rule.

The new construct is in lines 17 and 18:

```
  (:derived (free ?r - robot)
    (not (exists (?i - item) (carrying ?r ?i))))
```

A `:derived` definition, a **rule**, has two parts. The first is the **head**: the predicate with its variables, typed as in a parameter list. The second is the **body**: a condition, written with everything that Chapter 5 taught for preconditions (`and`, `or`, `not`, `exists`, `forall`, `imply`), whose free variables

are those of the head. The rule reads from right to left: *if* there is no item that `?r` carries, *then* `(free ?r)` holds. And since the world is closed, as always, it also reads the other way: if the body is false, `(free ?r)` is false. The domain declares `:derived-predicates`, and the predicate is still listed in `:predicates`, like any other.

Around the rule, three things disappeared. `pick` no longer deletes `free`, `drop` no longer adds it, and the initial state of `courier-free-p01.pddl` does not mention it. The precondition of `pick` did not change: it asks for `(free ?r)` as before, and the goal of the problem asks for `(free r1)` too.

```
$ 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)
```

Two items, one at a time. After the first `pick` the robot carries `meds2`, the body of the rule is false, and the second `pick` is not applicable until the `drop`: the rule does what the hand-kept fact did. The plan is the same that the domain of Chapter 4 gives for the same problem.

Where a derived predicate may appear

A derived predicate can be *read* wherever a basic one can: in preconditions, in the conditions of durative actions, in the condition of a `when`, in goals, and in the bodies of other rules. It can never be *written*. It may not appear in an effect:

```
$ 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
```

Nor may the initial state assert it. A problem that states `(free r1)` for this domain is rejected with `derived predicate 'free' cannot be initialized`. Both prohibitions are the same idea. If an action or a problem could set the fact, there would be states in which the fact and its rule disagree, and the rule would no longer be a definition.

A derived predicate is read like any other and never written: not in effects, not in the initial state. Its truth in each state is whatever its rules say.

A predicate may have several rules. Then it holds when the body of *any* of them holds: several rules are an `or`. A hospital where a room may be

entered with a keycard or with an escort could say it in two lines, one per reason, instead of one rule with a disjunction.

Axioms: out of PDDL and back in

Rules of this kind are much older than PDDL 2.2. The first definition of PDDL, in 1998 McDermott et al., 1998, already had them, under the name of *axioms*. Few planners supported them and later versions of the language left them out. Stefan Edelkamp and Jörg Hoffmann brought them back, as derived predicates, in PDDL 2.2, the language of the 2004 planning competition Edelkamp and Hoffmann, 2004; the other novelty of that version, timed initial literals, fills the second half of this chapter. Shortly after, Sylvie Thiébaux, Jörg Hoffmann and Bernhard Nebel argued why axioms deserve their place: they showed that a domain with axioms cannot in general be rewritten without them unless either the domain or the plans become exponentially larger Thiébaux et al., 2005.

13.3. Rules that call themselves

The rule for `free` could have been written inside `pick`; the derived predicate only gave the condition a name. The next rule cannot be written inside any precondition, because it has no fixed size.

The floor now has three doors: `d-pharmacy` between the lobby and the pharmacy, `d-corridor` between the lobby and the corridor, and `d-icu` between ward B and the ICU. We use the door predicates of Chapter 5, `(joins ?d ?a ?b)` and `(open ?d)`. And we change the level of detail at which the robot moves. A real courier robot has a navigation system that finds its own way through the building; the task planner does not need to choose every step, only to know *whether* the robot can get from one room to another. So `move`, between neighbors, becomes `go`, between any two rooms such that the second can be reached from the first (Listing 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)))
```

Listing 13.2: Reachability through open doors: the predicates, the two rules and the action `go` of `courier-reach-domain.pddl`.

There are two rules, one built on the other.

`passable` is one step: two connected rooms such that every door between them is open. The `forall` with `imply` is the "every ... that ..." of Chapter 5; if there is no door between two rooms, it is true of all zero of them, and the rooms are passable.

`reachable` is any number of steps, and its body mentions `reachable` itself: `?b` can be reached from `?a` if one step leads there, or if one step leads to some room `?c` from which `?b` can be reached. This is a **recursive rule**. It is the definition of a path that one would write in mathematics, and it has no equivalent with `exists` alone: a formula without recursion can speak of paths of two steps, or three, or any fixed number, but not of paths of whatever length the map happens to need.

How can a rule that refers to itself define anything? By building up from nothing. Start with no `reachable` fact at all. Apply the rule to everything it can derive: this gives the pairs joined by one passable step. Apply it again, now with those facts available: the pairs two steps apart. Go on until a round adds nothing new. What has been derived by then is true; everything else is false. Figure 13.1 shows the rounds on the map with the door of the ICU closed.

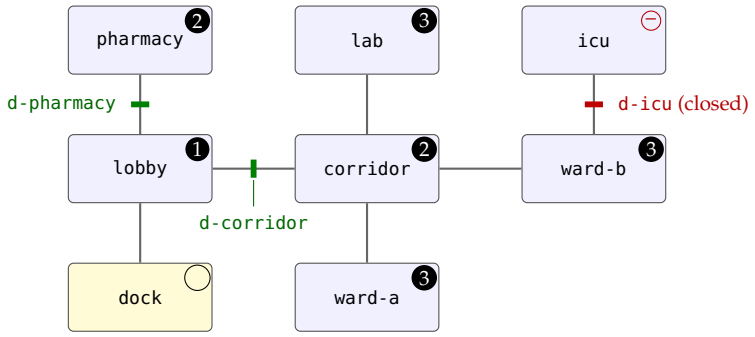


Figure 13.1.: Deriving `(reachable dock ?b)` with two doors open (green) and one closed (red). The number in each room is the round in which the fact for that room is derived: one passable step, then two, then three. A fourth round adds nothing, so `(reachable dock icu)` is false.

Three problems exercise the domain. In `courier-reach-p01.pddl` every door is open and the medicines go from the pharmacy to the ICU:

```
$ 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)
```

Four actions where the step-by-step courier needs eight. The second `go` crosses four rooms and three doors in one action, because `(reachable pharmacy icu)` was derived.

In `courier-reach-p02.pddl` all three doors start closed:

```
$ 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)
```

This plan shows what a derived fact is. No action in it has `reachable` in its effects, and yet each `open-door` changes which `go` actions are applicable next. After the first two, the robot can go from the pharmacy straight to ward B; it cannot go further, because of the third door, so it stops there, opens it, and goes on. The planner treats the doors as the means and reachability as their consequence, which is how we think about them.

The third problem uses the derived predicate in a goal, under a negation. The ICU must receive its medicines and then be isolated from the lobby again, with the robot back at the dock (`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)
```

The goal does not say which door to close, and the two plans close different ones. The first closes the door of the corridor, from the lobby side, which cuts the lobby from half of the floor; the second closes the door of the ICU on the way out. Both make `(reachable lobby icu)` false. A goal on a derived predicate states *what* must hold and leaves the *how* to the planner.

A recursive rule defines facts of unbounded depth, such as paths in a graph. Its facts are what repeated application of the rule derives, starting from nothing.

Two more recursions: towers and circuits

The same shape of rule appears wherever a relation is the repetition of a simpler one. In the blocks world, `(on ?a ?b)` says that a block lies directly on another. "Somewhere above" is its repetition (Listing 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))))
```

Listing 13.3: Blocks world with `clear` and `above` as derived predicates.

`clear`, which every blocks world of this book kept by hand in four effects, is here the absence of a block on top, the same pattern as `free`. `above` is the pattern of `reachable`. With them the goal can say what matters and no more. In `blocks-above-p01.pddl` the tower is `d` on `c` on `b` on `a`, and the goal is `(and (above a d) (clear a))`: `a` somewhere over `d`, with nothing on it. It does not say where `b` and `c` end.

```
$ 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)
```

The planner chose the cheapest arrangement that satisfies the goal: `a` directly on `d`, the other two on the table. A goal with `on` would have forced us to choose one arrangement ourselves. The problem `blocks-above-p02.pddl` asks for `a` above `b` and `b` above `a`; Kairos searches every arrangement of the three blocks and answers `unsolvable`.

The third domain has no robot and almost no actions. An electrical installation has a source, points joined by wires, and switches on some wires. A point has power if it is the source or if it is linked to a point that has power (Listing 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))))
```

Listing 13.4: A circuit: which points have power follows from the switches.

The only actions close and open switches; everything of interest is derived. In `circuit-p01.pddl` (Figure 13.2) the oven must have power while the kitchen, where an electrician is working, has none. Initially the two switches of the kitchen line are closed and the others open.

```
$ kairos circuit-domain.pddl circuit-p01.pddl
; cost = 5
(open-switch s2)
(open-switch s4)
(close-switch s3)
(close-switch s1)
(close-switch s5)
```

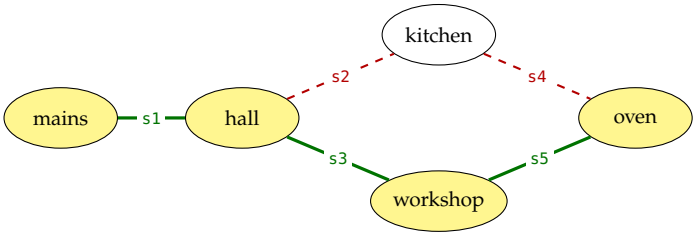


Figure 13.2.: The circuit of `circuit-p01` after the plan. Closed switches are solid green, open ones dashed red; the points with power are shaded. Power reaches the oven through the workshop. Look at `s4`: it had to open, because a closed switch conducts in both directions and the oven would feed the kitchen.

The plan opens both switches of the kitchen, not only the one on the side of the hall. The rule for `linked` is symmetric, so a powered oven behind a closed `s4` would give power to the kitchen from behind. The planner found a fact about circuits that a hurried electrician forgets, and it found it in the rules, not in any action.

Rules and negation

The rule for `free` has a negation in its body, and so could a rule of ours that used another derived predicate negated: "a room is isolated if it is not reachable from the lobby". What a rule cannot do is depend negatively on *itself*, directly or through other rules. The procedure that builds the facts round by round only makes sense if a fact, once derived, stays derived; a rule such as

```
(:derived (odd ?l - lamp) (not (odd ?l)))
```

would derive a fact because it is absent and thereby make its own reason false. Kairos rejects it: `derived predicate 'odd' depends negatively on itself`. Sets of rules without such cycles are called **stratified**: the predicates can be put in layers, each layer using the lower ones freely and itself only positively, and the layers are computed one after the other. All the rules of this chapter are stratified in two layers at most.

PDDL 2.2 is stricter than this: it does not allow a derived predicate under a negation in the body of a rule at all. Kairos and the planners built on Fast Downward accept any stratified set of rules.

13.4. What a planner does with rules

For the planner, a state is still a set of basic facts. Derived facts are not stored in it: each time the search reaches a new state, the rules are run on its basic facts until nothing new comes out, and only then are preconditions and the goal tested. The option `-v` shows the size of that work for the doors:

```
$ 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
...
```

Our two rules became 518 ground rules (the planner calls them axioms), built from the instances of the rules for particular rooms and doors. The numbers grow fast. With eight rooms there are 64 pairs `(reachable ?a ?b)`, and the recursive rule can derive each of them through any of eight intermediate rooms. A rule with three variables over n objects has in the order of n^3 instances, and they are evaluated in every state that the search visits. Derived predicates are not free: they move work from the modeler to the planner, and on a large map the modeler should know how much.

There is a second cost, less visible. The heuristics of Chapter 3 estimate the distance to the goal by looking at what actions add and delete. A derived fact is added and deleted by no action, so the heuristic must reason through the rules to see that opening a door brings the ICU closer. Planners differ in how well they do it. In the three problems of the doors the search expanded at most a few hundred states, so here it did well; a model in which everything of interest is derived, like the circuit, gives the heuristic less to hold on to.

Why not compile the rules away?

If a rule is only a name for a condition, a planner could replace the name by the condition and forget the rule. For `free` this works: it is what we would have done by writing the `exists` inside `pick`. For a recursive rule there is no condition to substitute. The alternative is to add actions that derive the facts step by step, and to force the planner to run them after every real action; plans then grow with the number of derivation steps. The result of Thiébaux, Hoffmann and Nebel Thiébaux et al., 2005 says that this is no accident of one compilation: any way of removing axioms makes the domain or the plans exponentially larger in the worst case. Planners that support derived predicates well, such as Fast Downward Helmert, 2006, therefore keep them as rules, organized in layers, and evaluate them in every state.

The third cost is the one of every feature beyond STRIPS: fewer planners. A model that uses a derived predicate only as a name, as with `free`, can be rewritten for a planner without them in a minute. A model built on a recursive rule cannot, and choosing it is choosing the planners that will run it.

Portability. In the check that the documentation of Kairos keeps (one recursive rule, ten seconds per planner), Kairos, LPG-td and LAMA returned a valid plan; OPTIC, POPF, TFD, ENHSP and Metric-FF did not. Derived predicates are rare among temporal and numeric planners.

Rules are grounded like actions and evaluated in every state. A rule with three variables over n objects costs in the order of n^3 ground rules.

13.5. A world with a calendar: timed initial literals

We leave the rules and turn to the clock. The temporal courier of Chapter 10 lives in a hospital where nothing happens unless a robot does it. Now the pharmacy has opening hours. What we need to say is simple: at time 30 the pharmacy opens. Nobody opens it, no action causes it, no condition has to hold. It is a fact about the future that is known today.

PDDL 2.2 writes it in the initial state of the problem:

```
(at 30 (is-open pharmacy))
```

This is a **timed initial literal**: a time and a literal. At that time the literal becomes true, whatever the plan is doing. A negative literal, `(at 50 (not (is-open pharmacy)))`, becomes false at its time. The times are counted from the start of the plan, in the same unit as the durations. The requirement is `:timed-initial-literals`, and it is all the syntax there is. `(is-open ?l)` is an ordinary predicate that the domain declares and that actions use in their conditions; the only special thing about it is that no action of the domain has it in an effect. The world changes it, on schedule.

Listing 13.5 shows the two actions of the domain that look at the calendar. The domain is the temporal courier with `(idle ?r)` and durations from distances, and `drop` has become `deliver`, which hands an item over at its destination.

```
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))
```

```
      (over all (robot-at ?r ?l))
      (over all (is-open ?l)))
    :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))))

  ; Handing the item over at its destination, before its deadline.
  (:durative-action deliver
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration 2)
    :condition (and
      (at start (idle ?r))
      (at start (carrying ?r ?i))
      (at start (destination ?i ?l))
      (over all (robot-at ?r ?l))
      (over all (is-open ?l))
      (at end (in-time ?i)))
    :effect (and
      (at start (not (idle ?r)))
      (at start (not (carrying ?r ?i)))
      (at end (item-at ?i ?l))
      (at end (delivered ?i))
      (at end (free ?r))
      (at end (idle ?r)))))
```

Listing 13.5: The actions `pick` and `deliver` of `courier-hours-domain.pddl`: a room must be open during the whole handover.

Both ask for the room to be open `over all`: during the whole action, not just when it starts. (The condition on `in-time` at the end of `deliver` is for the next section.) In the problem `courier-hours-p01.pddl` every room is open except the pharmacy. It is absent from the initial facts, and so closed, and it gets one timed literal:

```
    (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]
```

For the first time in the book a plan has a hole in it. The robot is at the pharmacy at 10.001 and does nothing until 30.000. No action called "wait" was needed: in a temporal plan, waiting is the absence of actions, and the planner chose the start time of `pick` to be the instant at which its condition becomes true. Without the timed literal this same problem took 42.006 seconds; the calendar costs 20 (Figure 13.3).

A timed initial literal makes a fact true or false at a fixed time, whatever the agent does. Plans adapt to it by placing actions inside the windows, and by waiting.

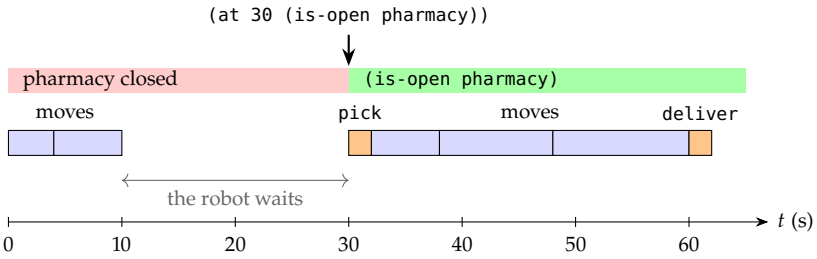


Figure 13.3.: The calendar (top) and the plan (bottom) of `courier-hours-p01`. The upper line is not chosen by the planner; the lower one is. Look at the gap between 10 and 30: the robot stands at the door of the pharmacy.

How does Kairos plan this? With `-v` it tells that its engine M3 takes the task, because with timed initial literals "actions must happen within time windows". Each timed literal is a happening with a fixed time, and the planner adds to the temporal constraints of the plan (Chapter 11) that the actions which need the fact come after it, and those which need its absence come before.

Portability. In the same check as before, Kairos, OPTIC and POPF planned with timed initial literals; TFD, LPG-td, ENHSP, Metric-FF and LAMA did not. The feature needs a temporal planner, and not every temporal planner has it.

13.6. Windows and deadlines

Two timed literals on the same fact, one that makes it true and a later one that makes it false, are a **window**: an interval in which something is possible. In `courier-hours-p02.pddl` the pharmacy is open only from 30 to 50, and the lab, where a sample from ward A must go, closes at 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]
```

There are two orders in which to serve the two items, and the windows leave one. Fetching the sample first would bring the robot to the pharmacy at 74, long after it closed. So the robot waits for the pharmacy, as before, and then has just enough time: the sample is handed over at 86.008, less than four seconds before the lab closes. The `over all` condition matters here. A delivery that started at 89 would start with the lab open and end with it closed, and the model rejects it, as the hospital would.

Deadlines

A **deadline** is half a window: something must happen before a given time. PDDL 2.2 has no construct for it, and needs none. The pattern is a fact

that means "it is still in time", true from the start and removed by a timed literal, and an action that needs the fact when it ends. In our domain the fact is `(in-time ?i)` and the action is `deliver`, with its condition `(at end (in-time ?i))`. The problem `courier-hours-p03.pddl` has two deliveries, medicines for ward A that must arrive before time 50 and a meal for ward B that can wait:

```
    (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]
```

The medicines arrive at 42.006, in time. The meal was nearer (it lies in the lobby, which the robot crosses at the start), and a plan that served it first has exactly the same length; it is in the file `courier-hours-mealfirst.plan`. The deadline is what tells the two apart:

```
$ 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
```

A deadline that cannot be met gives a problem without a plan, and for timed literals Kairos can often prove it. In `courier-hours-p04.pddl` the deadline is 20, and the medicines cannot be in ward A before 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)
```

The note is worth reading slowly. The planner computed the earliest time at which `(delivered meds1)` could possibly be achieved, in the optimistic way of the heuristics of Chapter 3 (ignoring deletes, with the shortest durations), and found it later than the instant at which `(in-time meds1)` disappears. When even the optimist is late, nobody arrives in time, and that is a proof.

A satellite and its windows

Windows are the daily bread of space missions, where the calendar is written by orbital mechanics. A satellite can photograph a target only while it flies over it, and can send its images only while a ground station is in sight. Both intervals are known long in advance (Listing 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)))))
```

Listing 13.6: A satellite that observes and transmits inside windows.

Nothing in the domain mentions time beyond the durations. The windows are in the problem, `satellite-p01.pddl`: the crater is visible from 10 to 20, the volcano from 12 to 30, and the station of Madrid is in sight from 35 to 50. The camera starts pointing at the volcano.

```
   (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]
```

The satellite does first what its initial pointing does not suggest: it turns away from the volcano to the crater, whose window closes sooner, and is ready before the window opens (Figure 13.4). Then it turns back for the volcano, and both images wait in memory for Madrid. The two transmissions take a little over 12 of the 15 seconds of the pass.

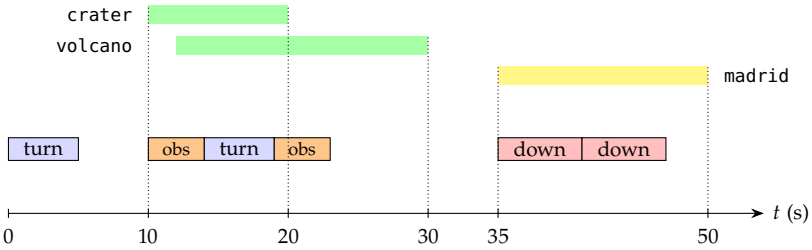


Figure 13.4.: The windows of `satellite-p01` (top: when each target is visible and when the station is in sight) and the plan (bottom; *obs* is `observe`, *down* is `downlink`). Every observation lies inside the window of its target and both transmissions inside the pass of the station.

In `satellite-p02.pddl` the crater is visible for only three seconds, and an exposure takes four. Kairos answers `unsolvable`, with the same note as for the deadline: the window takes `(visible crater)` away before `(sent crater)` can be reached.

13.7. Numbers on the calendar

What the calendar does to facts it can also do to numbers. During visiting hours the corridors of the hospital fill with people and the robot must go slowly. That is a number, the speed, changing at known times:

```
    (= (speed r1) 1)
    (at 20 (= (speed r1) 0.5))    ; a timed initial fluent
    (at 60 (= (speed r1) 1))
```

A **timed initial fluent** assigns a value to a numeric fluent at a time. It is not part of PDDL 2.2, which only schedules literals; it is a later extension with its own requirement, `:timed-initial-fluents`, that some temporal planners accept. Kairos lists it among its features:

```
$ kairos --features
...
  timed-initial-literals     full     :timed-initial-literals
  timed-initial-fluents      full     :timed-initial-fluents
...
```

The problem `courier-hours-p05.pddl` is the single delivery to ward A, with the pharmacy open all day and visiting hours from 20 to 60. The domain is unchanged: `move` already computed its duration from `(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]
```

The last move, 12 meters, takes 24 seconds: it starts at 28, when the speed is 0.5. The move before it is more interesting. It starts at 18.004, two seconds before the crowd arrives, and lasts its usual 10 seconds, although most of it happens in visiting hours. This is not an error of the planner. It is what the model says: the duration of a durative action is computed when the action starts, from the values of that instant (Figure 13.5). If a robot caught by the crowd half way should slow down, the model must be finer: shorter moves, or the continuous change of Chapter 12 with a rate that depends on the speed.

A timed initial fluent sets a number at a known time. Actions that read the number when they start are not affected by a change that comes while they run.

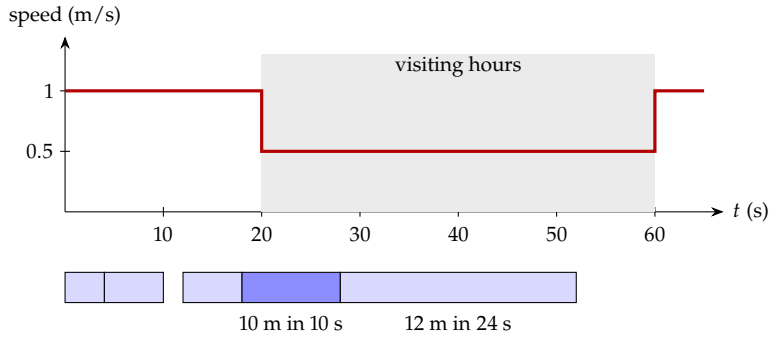


Figure 13.5.: The speed of the robot, set by two timed initial fluents, and the moves of the plan for the problem `p05`. Look at the darker move: it starts at 18 with speed 1 and keeps the duration computed then, although the speed drops at 20.

13.8. The MODEL method: consequences and the calendar

Both halves of the chapter are about changes that no action makes, and each adds a question to the method.

New questions for steps M and O

Map the situation. Does the world change at known times, whatever the agent does? Opening hours, deadlines, shifts, daylight, passes of a satellite. Each one goes into the model card under *Assumptions*, as a schedule: what changes, when, and that the agent cannot influence it. In PDDL it becomes timed initial literals (or fluents) in the problem, and a condition on the actions that depend on it: `over all` if the action needs the fact throughout, `at end` if it is a deadline.

Objects and state. For every property in the state table: is it a consequence of other properties? If knowing the rest of the state always tells its value, it is a candidate for a derived predicate, and

the table records the rule instead of the actions that would change it. Then ask how the consequence is computed: by looking at a fixed number of facts (like `free`), or by following a chain of unknown length (like `reachable`).

Step O on the courier with doors

The state table of `courier-reach` has a new kind of row.

Property	Changes when	Encoding
`(open ?d)`	a robot opens or closes the door	basic, dynamic
`(joins ?d ?a ?b)`	never	basic, static
`(passable ?a ?b)`	follows from `connected`, `joins`, `open`	derived
`(reachable ?a ?b)`	follows from `passable`, any number of steps	derived, recursive

Alternatives compared

The fact `(free ?r)` has now been encoded in three ways, and the three run on the same problem (two items from the pharmacy to ward A) with the same plan of 15 actions:

- *Kept by effects*: the domain of Chapter 4, `courier-domain.pddl`. `pick` deletes the fact, `drop` adds it, the initial state states it.
- *Derived*: `courier-free-domain.pddl`, Listing 13.1.
- *Written out*: `courier-inline-domain.pddl`, with no `free` predicate at all; `pick` has the precondition `(not (exists (?j - item) (carrying ?r ?j)))`.

	Kept by effects	Derived	Written out
Fidelity	Right only while every action and every initial state keep the two predicates in step	Right by construction	Right by construction
Size	27 facts, 46 actions, no rules	27 facts, 46 actions, 1 rule	26 facts, 46 actions, no rules
Guidance	Best: the fact is added and deleted by actions, which heuristics see	The heuristic must look through the rule	The heuristic must look through a negated `exists`
Clarity	Two predicates for one idea; the invariant is in the modeler's head	One definition, with a name, used anywhere	The condition is repeated wherever it is needed
Portability	Any planner	Needs `:derived-predicates`	Needs negative and existential preconditions

Table 13.1.: Three encodings of "the robot carries nothing". The sizes are those that `kairos -v` reports for the problem with two items.

Verdict. For a fact as simple as `free`, used by one action, keeping it by effects remains a good choice: it costs two effects, every planner accepts it, and it gives the search the most to work with. This is why the courier world of this book keeps it. The derived version earns its place when the fact is used in many actions, or when new actions keep being added to

the domain and each one is a chance to forget the bookkeeping. Writing the condition out is right when it is needed once and the planner has no derived predicates.

For `reachable` there is no table to draw, because one of the columns is empty: the fact cannot be kept by effects. The real alternative is another level of detail, the step-by-step `move` through passable doors of `courier-steps-domain.pddl`. On the problem with all doors closed it gives a plan of 11 actions where `go` needs 9, with 392 ground rules instead of 518 and 42 ground actions instead of 92. The recursive rule is not cheaper. It is the right choice when navigation is not the planner's business, and it lets goals and conditions speak of reachability itself, as in "leave the ICU isolated".

A deadline: timed literal or constraint

A deadline can also be written without touching the domain, with the plan constraints of PDDL 3 that Chapter 15 studies. The problem `courier-hours-p06.pddl` is `p03` without the timed literal and with one more section:

```
(:constraints (within 50 (delivered meds1))))
```

Kairos solves it with a valid plan that delivers the medicines first, and its validator rejects the plan that serves the meal first, with the message `the plan violates the problem :constraints`. The two encodings accept the same plans here. They differ in the other criteria. The constraint is clearer: it says "within 50" in one line of the problem, where the timed literal needs a helper predicate, a condition in an action and two facts per item. The timed literal is more portable among temporal planners, and it can express what `within` cannot: a window that opens, closes and opens again, or a fact that an action needs *throughout*. Use `within` for a plain deadline on a goal when the planner supports it, and timed literals for everything that is a schedule of the world.

13.9. Common mistakes

Writing a derived predicate

The habit of keeping facts by hand survives the introduction of the rule: the modeler adds `:derived` and leaves an old effect, or an old initial fact, in place.

```
$ 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]
...
```

In a problem the message is `derived predicate 'free' cannot be initialized`. The fix is always to delete the offending line: the rule already says it.

A rule that only looks one step ahead

Writing `reachable` without the recursion,

```
(:derived (reachable ?a ?b - location)
  (or (passable ?a ?b)
      (exists (?c - location)
        (and (passable ?a ?c) (passable ?c ?b)))))
```

gives a model that works on every test with rooms at most two steps apart and fails on the first long corridor, with a plan full of short hops or with no plan. No message warns of it: the rule is legal and means something else. The test set of step E needs a problem that is deeper than the rule would be without recursion.

The fact is already true

A timed literal that makes a fact true has no effect if the fact is true from the start. In `courier-hours-wrong-p01.pddl` the modeler listed every room as open, the pharmacy too, and then added its opening time:

```
    (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]
...
```

The robot picks at 10, twenty seconds before the pharmacy opens. The file is correct PDDL and the check accepts it. For every fact on the calendar, decide its value at time zero, and write the first timed literal as the first *change*.

A window checked only at the start

With `(at start (is-open ?l))` instead of `over all`, an action may begin an instant before the room closes and finish inside a closed room. Whether that is wrong depends on the world: a customer who is inside when the shop closes is still served, a satellite whose target leaves the field of view half way through the exposure has no image. Decide it on purpose, with the three questions of Chapter 10 (what must hold at the start, throughout, at the end), and test it with a problem whose window is shorter than the action, like `satellite-p02.pddl`.

A requirement left out

Timed literals are written in the problem, but the requirement belongs with the others, and it is easy to forget because the domain itself does not change. Kairos plans anyway and warns in the check:

```
warning: uses timed-initial-literals but :requirements does not
    include :timed-initial-literals [semantic.
    undeclared_requirement]
```

The same warning exists for `:derived-predicates`. A problem may

declare requirements of its own, as `courier-hours-p05.pddl` does for `:timed-initial-fluents`.

13.10. Summary

PDDL 2.2 gave facts two new ways to be true. A derived predicate is true because a rule over the current state says so; a timed initial literal makes a fact true or false because the clock reached a time written in the problem. Neither is under the control of the agent, and both let a model say directly something that before had to be simulated with bookkeeping.

Table 13.2.: What this chapter added.

Construct or idea	What to remember
`(:derived (p ?x) body)`	`p` holds exactly when the body holds; requirement `:derived-predicates`
Where it may appear	Read anywhere (conditions, goals, other rules); never in effects or in the initial state
Several rules, one head	Any of them is enough
Recursive rule	Facts are built round by round from nothing; for paths, towers, networks
Negation	Allowed if no predicate depends negatively on itself (stratified)
Cost	Ground rules evaluated in every state; weaker heuristics; fewer planners
`(at 30 (p))` in `:init`	`p` becomes true at time 30 whatever the plan does; `(at 50 (not (p)))` makes it false; requirement `:timed-initial-literals`
Window	A literal that opens and one that closes; actions need the fact `over all`
Deadline	A fact removed at the deadline, needed `at end` by the action that must be in time
`(at 20 (= (f) 0.5))`	A timed initial fluent: a number set at a known time; an extension, `:timed-initial-fluents`

13.11. Further reading

The report that defines PDDL 2.2 Edelkamp and Hoffmann, 2004 is short and introduces both constructs, with the restrictions that the 2004 competition placed on them. The case for derived predicates, with the proof that they cannot be compiled away cheaply and a discussion of stratified rules, is the paper by Thiébaux, Hoffmann and Nebel Thiébaux et al., 2005. How a planner evaluates rules in layers during the search is described with Fast Downward Helmert, 2006. For timed initial literals in a temporal planner, see POPF Coles et al., 2010 and OPTIC Benton et al., 2012, whose forward search orders the actions of a plan around the fixed happenings. The `within` constraint used in the last comparison belongs to PDDL 3 A. Gerevini and Long, 2005 and to Chapter 15. The book by Haslum and colleagues Haslum et al., 2019 covers both features of PDDL 2.2 with further examples.

13.12. Exercises

Exercises for Section 13.1 — Facts that follow from other facts

Exercise 13.1.1. For each property, say whether it is basic or a consequence of others, and of which: (a) a room is empty (no robot and no item in it); (b) a door is open; (c) a robot is overloaded (it carries more than its capacity); (d) two robots are in the same room; (e) an item is in the pharmacy.

Exercise 13.1.2. A new robot model can carry two items. In the domain of Chapter 4, what would `pick` and `drop` have to do to keep `(free ?r)`, meaning "can take one more item", correct? Explain why this bookkeeping is harder than for one item, without writing the PDDL.

Exercises for Section 13.2 — Derived predicates

Exercise 13.2.1. Add to the typed courier domain a derived predicate `(tidy ?l)`: no item lies in the room. Write a problem in which two items lie in the corridor and the goal asks for a tidy corridor and a tidy lobby, and read the plan: where did the planner put the items?

Exercise 13.2.2. A colleague receives the domain of Listing 13.1 with its problem and adds `(free r1)` to the initial state "to be safe". Predict the message of `kairos check` and explain why the language forbids what looks like a harmless redundancy.

Exercise 13.2.3. A restricted room may be entered by a robot that has a keycard, and any other room by any robot. Write `(may-enter ?r ?l)` in two ways: with one rule and an `or`, and with two rules. Which reads better?

Exercises for Section 13.3 — Rules that call themselves

Exercise 13.3.1. In the map of Figure 13.1, suppose that `d-corridor` is closed and the other two doors are open. Compute by rounds the rooms `?b` for which `(reachable ward-a ?b)` holds. Is `(reachable ward-a ward-a)` among them? Does that matter for the action `go`?

Exercise 13.3.2. A garden has an inlet and points joined by pipes, each pipe with a valve. Water reaches a point if it is the inlet or if an open valve joins it to a point with water. Model it, with actions that open and close valves, and solve: the garden can be fed through the pool or through the shed; give it water while the pool, which is being repaired, stays dry.

Exercise 13.3.3. In the blocks domain of Listing 13.3, write the goal "`a`, `b` and `c` are in the same tower, in any order" using only `above`. Then explain why `blocks-above-p02.pddl` is unsolvable, and why Kairos can prove it here while it often could not in Chapter 12.

Exercise 13.3.4. Explain what is wrong with the pair of rules "a lamp is on if it is not off" and "a lamp is off if it is not on", both as derived predicates. What would the round-by-round procedure do with them?

Exercises for Section 13.4 — What a planner does with rules

Exercise 13.4.1. The doors domain exists with `go` and with `move` (the files `courier-reach` and `courier-steps`). Run the problem `p02` of each with `-v` and compare the lines that start with `[ground]`. Explain where the difference in the number of rules and in the number of actions comes from.

Exercise 13.4.2. Estimate how many ground rules `reachable` would have on a hospital with 100 rooms. Propose a model for that hospital in which the robot still does not plan every step. (Hint: are the doors opened by the robot?)

Exercises for Section 13.5 — A world with a calendar

Exercise 13.5.1. The pharmacy opens twice: from 0 to 8, and from 40 to 60. Write the problem (one delivery to ward A, as in `p01`) and predict the makespan before running it. Could the robot use the first window?

Exercise 13.5.2. In the plan of `courier-hours-p01.pddl` the robot goes to the pharmacy first and waits there. Is a plan in which it waits at the dock and arrives exactly at 30 also valid? Is it better or worse by the makespan? Give a reason, outside the model, to prefer one of the two.

Exercises for Section 13.6 — Windows and deadlines

Exercise 13.6.1. In `courier-hours-p02.pddl`, what is the earliest closing time of the lab for which the problem still has a plan? What is the latest opening time of the pharmacy window (keeping its length of 20 seconds) for which the lab, closing at 90, can still be served?

Exercise 13.6.2. A rover drives and takes pictures only by daylight. A drive between neighbor sites takes 30 seconds and a picture 10. Model it with a fact `(daylight)` on the calendar: two days of 100 seconds with a night of 50 between them. Three sites lie in a line, base, dune and rock; the rover starts at the base and must photograph the dune and the rock and return. Read the plan: what does the rover do during the night?

Exercise 13.6.3. In the satellite problem, the plan observes the crater first. Write by hand the plan that observes the volcano first and validate it. Which condition fails, and at what time?

Exercises for Section 13.7 — Numbers on the calendar

Exercise 13.7.1. In `courier-hours-p05.pddl`, move the visiting hours: the crowd arrives at 10 and leaves at 30. Predict what a robot in a hurry should do, then run the planner and read all the plans it prints.

Exercise 13.7.2. Name two quantities of a hospital, other than the speed of a robot, that change at known times and that a planner of deliveries would want as timed initial fluents.

Exercises for Section 13.8 — The MODEL method

Exercise 13.8.1. In the domain of Listing 13.5, `(delivered ?i)` is added by the action `deliver`. Could it be a derived predicate, "the item is

at its destination"? Write the rule, and say what would be lost with respect to the deadline.

Exercise 13.8.2. Classify each change by who causes it, the agent (an action), the state (a rule) or the calendar (a timed literal): (a) the cafeteria closes at four; (b) a room becomes crowded when more than three robots are in it; (c) a robot locks a door; (d) the night shift starts; (e) a ward is supplied when it has its medicines and its meal.

Part V. Preferences and constraints: PDDL 3

14. Soft goals and preferences

Every goal written so far was an order. The planner either reached all of it or answered that there was no plan, and a model with one goal too many was a model without plans. Real requests are seldom like that. The morning list of the courier says: medicines to ward A, medicines to the ICU, a sample to the lab. The battery holds 100 units and nobody has repaired the charger. If the three deliveries need more than 100, a planner that reads the list as one goal answers `unsolvable` and the robot stays at the dock all morning, although it could have served two wards out of three.

PDDL 3.0 adds the construct for that situation: the **preference**, a condition that the plan should satisfy and may violate, at a price that the modeler sets. With preferences a problem has two kinds of goals, the ones that must hold and the ones that are wished, and the metric of Chapter 9 says how much each wish is worth. The planner then solves a different problem: no longer "reach the goal", but "reach what is required, and give up as little as possible of what is wished".

The chapter starts from the list that does not fit in the battery and turns its deliveries into wishes. It then gives the wishes different weights, sets them against the meters that granting them costs, writes one preference for many objects with `forall`, and moves preferences from the goal into the preconditions of an action, where they speak about the way and not about the end. Two other worlds, a tourist's day and a satellite with more requests than time, use the same tools. The last sections look at what a planner does with a preference, add the wishes to the MODEL method and compare a preference with its two neighbors: the hard goal and the action cost.

14.1 More wishes than battery 306
14.2 A preference in the goal 307
14.3 What each wish is worth 309
A wish about the robot itself 310
Hard and soft goals together 311
14.4 Wishes against effort . 312
14.5 One preference, many instances 313
14.6 Wishes about the way: preferences in preconditions 314
14.7 Two more worlds 316
A tourist's day 316
A satellite with more requests than time . . . 317
14.8 What the planner does with a preference 319
14.9 The MODEL method: wishes and their worth 320
Alternatives compared . 321
14.10 Common mistakes . . . 322
14.11 Summary 324
14.12 Further reading 324
14.13 Exercises 325

14.1. More wishes than battery

We take the typed courier world of Chapter 4 and the battery of Chapter 8, in its simplest form: every move consumes as many units of battery as meters it covers, and today there is no `charge` action. The only new line of the domain is its requirement, `:preferences`, which the first problem does not use yet.

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

The problem `courier-battery-p01.pddl` holds the morning list. The robot `r1` starts at the dock with a battery of 100; `meds1` and `meds2` wait in the pharmacy and `sample1` in ward B (Figure 14.1). The goal asks for everything:

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

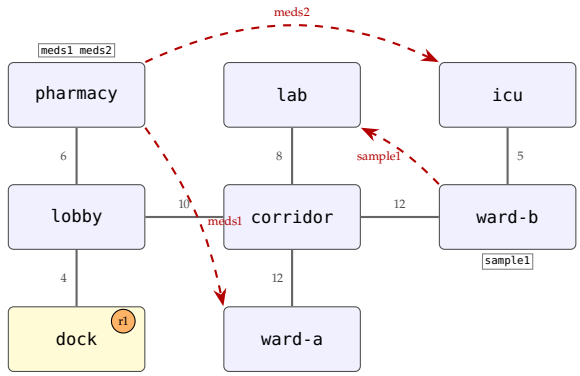


Figure 14.1.: The morning list. Dashed arrows are the three deliveries asked for; the numbers are meters, and each meter costs one unit of a battery of 100. The robot carries one item at a time.

```
$ 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)
```

The answer is correct and useless. Correct, because no order of the three deliveries fits in 100 units: Figure 14.2 shows what each subset of them consumes. Useless, because the question that the hospital has is another one: which deliveries should be made with the battery that there is.

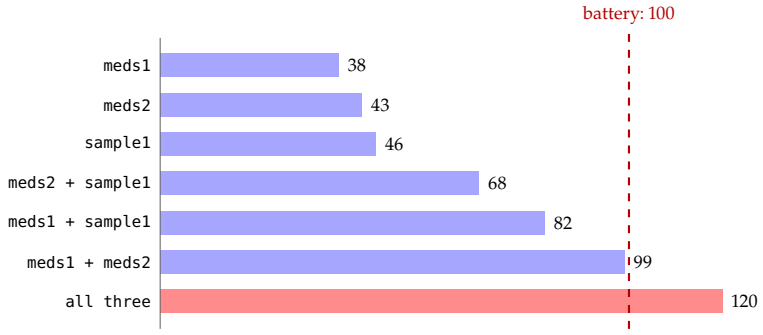


Figure 14.2.: Battery needed to make each set of deliveries, starting at the dock. Any two deliveries fit; the three together need 120.

A problem with more goals than its resources allow is called **over-subscribed**. It is the normal case whenever requests come from several people and resources (battery, time, fuel, memory) come from only one robot. The classical remedy is to delete goals by hand until a plan appears, but that makes the modeler do the planner's work: choosing *which* goals to delete is exactly the hard part, and it depends on distances, on the battery and on the order of the trips.

A goal is all or nothing. When the requests exceed the resources, a model made only of hard goals has no plan, and says nothing about the best partial answer.

14.2. A preference in the goal

A **preference** is a condition with a name, written `(preference name condition)`. Inside a goal it means: this condition should hold at the end of the plan, but a plan in which it does not hold is still a plan. A goal that is a preference is also called a **soft goal**; the ordinary ones are **hard goals**. The requirement is `:preferences`.

The problem `courier-battery-p02.pddl` is the morning list with its three deliveries turned into soft goals (Listing 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))))
```

Listing 14.1: The three deliveries as preferences, all worth the same.

Two things changed. In the goal, each delivery is wrapped in a preference and has a name: `meds-a`, `meds-icu`, `sample-lab`. And the problem has a `:metric`, which is where the names are used. The expression `(is-violated meds-a)` is a number: 0 if the plan satisfies the preference `meds-a`, 1 if it does not. The metric adds the three, so its value is the number of deliveries *not* made, and the planner is asked to minimize it.

A plan **satisfies** a goal preference when its condition holds in the last state, and **violates** it otherwise. The word is stronger than what it means: violating a preference breaks no rule, it only costs what the metric says. Since `is-violated` counts what went wrong, metrics with preferences are almost always `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)
```

The output is the anytime search of Chapter 9 at work, and it is worth reading from the top. The first plan has cost 3 and *no actions*: nothing follows its cost line. With only soft goals, doing nothing is a valid plan, the worst one: it violates the three preferences. The second plan delivers `meds1` and costs 2. The third adds `sample1` and costs 1, and there the search stops by itself: no plan violates fewer than one preference, since the three deliveries do not fit. The cost that Kairos prints is the value of the metric.

With preferences the planner always has a plan, the empty one, and the search becomes a search for better plans. The cost printed is the value of the metric.

A plan written by hand is validated as always, and `kairos validate` reports the metric that it reaches. The file `courier-battery-p03-lazy.plan` holds the seven actions that deliver only `meds1`:

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

Valid, with a metric of 2: a lazy plan is not a wrong plan. What preferences change is the meaning of "valid" against "good". Validity is still decided by preconditions and hard goals; everything about wishes lives in the metric.

Step M: required or wished

The model card of the courier world has had one line for *Success* since Chapter 2. For an over-subscribed day that line splits in two:

- **Required**: what a plan must achieve to be acceptable at all. Today, nothing; tomorrow, perhaps the sample that the lab is waiting for.
- **Wished**: what should be achieved if possible, each wish with its worth.

The test is to imagine the plan that leaves the condition out. If that plan is *unusable*, the condition is required. If it is only *worse*, it is a wish.

14.3. What each wish is worth

In `p02` the three wishes were worth the same, and the planner dropped the delivery to the ICU. The head nurse disagrees: the ICU comes first. The metric is an arithmetic expression, so each `is-violated` can be multiplied by a **weight**, the price of violating that preference. The problem `courier-battery-p03.pddl` only changes the metric:

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

Leaving the ICU without its medicines now costs 5, ward A 3 and the lab 1. Table 14.1 gives the metric of every set of deliveries that fits in the battery; the planner must find the row with the smallest value.

Deliveries made	Battery	`p02` (1, 1, 1)	`p03` (3, 5, 1)
none	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**

Table 14.1.: The metric of each feasible set of deliveries, with equal weights and with the weights of the head nurse. With equal weights three sets tie at 1; with weights, only one is best.

```
$ 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)
```

The costs 9, 4, 3 and 1 are four rows of the table: nothing, `meds2` alone, `meds2` with `sample1`, and finally both medicines. The last plan arrives at the ICU with one unit of battery left. Same domain, same initial state, same wishes: only their worth changed, and with it the plan.

A wish about the robot itself

A preference can wrap any goal condition, not only deliveries. The technicians would like the robot to end the morning at the dock, where they can look at the charger. That is one more soft goal, `(preference parked (robot-at r1 dock))`, and `courier-battery-p04.pddl` gives it a weight of 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))))
```

The best plan of `p03` ended in the ICU with a battery of 1: it cannot go home. Coming back competes with the deliveries for the same battery, and the planner has to weigh them:

```
$ 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)
```

The robot serves the ICU and the lab and returns to the dock with 10 units left. Ward A is given up (cost 3) because the alternative, both medicines and a robot stranded in the ICU, costs $1 + 4 = 5$. Note that the empty plan still costs 9 and not 13: the robot starts at the dock, so the plan that does nothing satisfies `parked`.

Hard and soft goals together

Hard goals and preferences mix freely inside the same `and`. If the lab must have its sample today, whatever happens to the rest, the problem `courier-battery-p05.pddl` leaves that delivery outside any preference:

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

Now the empty plan is not a plan. Kairos prints three plans, of cost 8, 5 and 3; the first one delivers only the sample and the last one adds `meds2`. The lazy plan of before, validated against `p05`, is now `plan invalid`: the goal `(item-at sample1 lab)` does not hold at its end.

A weight is the price of violating a preference. The metric adds the prices of everything that the plan gives up, and hard goals stay outside the bargain.

Choosing weights

Only the relation between weights matters: (3, 5, 1) and (30, 50, 10) describe the same problem. Three patterns cover most needs.

- **Equal weights**: maximize the number of wishes granted.
- **Worth**: each weight is what the wish is worth in a common unit (euros, minutes, meters). Two small wishes can then beat a large one, and that is intended.
- **Strict priority**: a wish of a higher class must beat *any number* of wishes of the lower classes. Give each class a weight larger

than the sum of all the weights below it. With weights 5, 3 and 1 the ICU beats ward A and the lab together ($5 > 3 + 1$).

14.4. Wishes against effort

So far a wish was dropped only when it did not fit. But a wish can fit and still not be worth its effort: nobody sends a robot 60 meters to deliver a napkin. To model that, the effort and the wishes have to be in the same metric, in the same unit.

The domain `courier-cost` forgets the battery and uses the action costs of Chapter 9: a move adds its meters to `total-cost`, and a `pick` or a `drop` adds 1. Its items have the subtypes `medicine`, `sample` and `meal`. In `courier-cost-p01.pddl` there are two wishes, a medicine for ward A and a meal for the ICU, both waiting in the pharmacy:

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

The metric reads as a sentence: meters traveled, plus 60 if the medicine does not arrive, plus 20 if the meal does not. The weights are now *in meters*: the medicine is worth a trip of up to 60, the meal one of up to 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)
```

Doing nothing costs $60 + 20 = 80$. Delivering the medicine costs 40 (38 meters and two handlings) and saves 60: the plan drops to $40 + 20 = 60$. Delivering also the meal would add 63 to the cost (back to the pharmacy and up to the ICU) to save 20, so the planner leaves the meal where it is, and proves that no plan is cheaper. Nothing prevented that delivery; it was just a bad deal. In `courier-cost-p02.pddl` the meal is worth 100, and the same domain gives a plan of cost 103 that makes both deliveries: the plans did not change, their price did.

When costs and violations share the metric, a wish is granted only if it is worth more than the effort it adds. The weights must be in the unit of the cost.

A metric that sets the worth of what is achieved against the cost of achieving it is called a **net benefit** metric. It has a consequence that surprises at first: with a net benefit metric there is no reason to expect that the planner achieves "as much as possible". It achieves as much as pays.

14.5. One preference, many instances

Tomorrow the pharmacy sends five medicines to the ICU, and next week twenty. Writing one preference per item works, but the problem file grows with the list, and the universal goals of Chapter 5 solved that for hard goals. Preferences have the same tool: a `forall` *around* a preference.

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

This goal of `courier-cost-p03.pddl` declares one preference per medicine, all under the name `meds-icu`. They are independent wishes: the plan may satisfy some and violate others. And `(is-violated meds-icu)` is no longer 0 or 1: it is the *number* of medicines that are not in the ICU at the end. The problem has two medicines and a meal in the pharmacy (the meal is not a `medicine`, so no preference speaks of it). The problem declares `:universal-preconditions` for the `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)
```

The empty plan costs 100: two violations at 50 each. The best plan delivers *one* medicine: 45 of effort plus one violation, 95. The second medicine would need 68 more (the way back and the way up again) to save 50. The planner does not care which medicine it takes: the two instances are worth the same.

The position of the `forall` matters. The other order, `(preference all-meds (forall (?m - medicine) (item-at ?m icu)))`, is a different wish: *one* preference whose condition is "every medicine is in the ICU". It is violated once, however many medicines are missing, and a plan that delivers one of two gets no credit for it. Table 14.2 sets the two side by side.

	`forall` outside	`forall` inside
Written as	`(forall (?m) (preference p ...))`	`(preference p (forall (?m) ...))`
Preferences	one per object	one
`(is-violated p)`	number of objects that fail	1 if any object fails
Partial success	counts	does not count

Table 14.2.: A `forall` around a preference and a `forall` inside it are two different wishes.

A forall around a preference makes one instance per object under one name, and is-violated counts the instances that fail.

The same counting happens whenever a name is used more than once. Two preferences written separately with the same name, for instance `(preference meds (item-at meds1 ward-a))` and `(preference meds (item-at meds2 icu))`, are added together: `(is-violated meds)` can

be 0, 1 or 2. A name identifies a *class* of wishes with one price, not a single wish.

14.6. Wishes about the way: preferences in preconditions

All the wishes so far were about the last state. Some wishes are about how the plan gets there. Visitors wait in the lobby, and the hospital would rather not have robots crossing it with biological samples. It is not forbidden: if there is no other way, the sample must still reach the lab. A soft goal cannot say this, because at the end of the plan the robot is no longer in the lobby, whatever it did on the way.

PDDL 3.0 allows a preference inside the precondition of an action. The action remains applicable when the preference does not hold; applying it in that case counts as one violation. The domain `courier-lobby` adds one to `move`, with `lobby` declared as a constant of the domain (Chapter 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))))
```

The first two conditions are hard, as always. The third is the wish: "the destination is not the lobby, or the robot carries no sample". A move that enters the lobby with a sample is legal and violates `clean-lobby`.

For a precondition preference, `(is-violated clean-lobby)` is the number of times that *any* action of the plan was applied while the preference did not hold. A robot that crosses the lobby three times with samples violates it three times.

On the fixed map of the floor this wish leaves no choice, because the lobby is the only way between the pharmacy and the rest. So the problems of this section open a back door between the pharmacy and the lab, 30 meters long, where the way through the lobby has 24 (Figure 14.3). A sample waits in the pharmacy and must (hard goal) reach the lab. In `courier-lobby-p01.pddl` one crossing is priced at 10 meters:

```
(:goal (item-at sample1 lab))
(:metric minimize (+ (total-cost)
                     (* 10 (is-violated clean-lobby))))
```

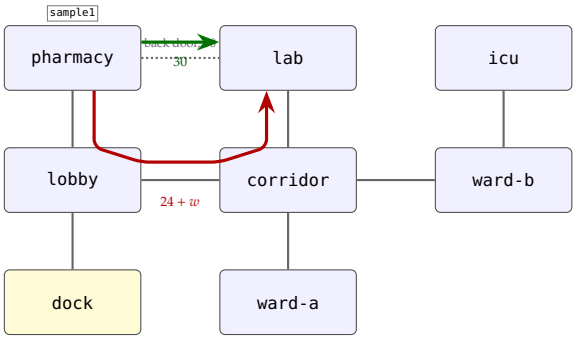


Figure 14.3.: Two ways to take the sample from the pharmacy to the lab. The red one is shorter, but it enters the lobby with a sample and pays the weight w of `clean-lobby`. The green one is the back door that this section adds to the map.

```
$ 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)
```

The robot goes around: 10 meters to the pharmacy, 30 through the back door and two handlings, 42 in all. The way through the lobby would cost $36 + 10 = 46$. In `courier-lobby-p02.pddl` the same crossing is priced at 3, and the answer turns:

```
$ 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)
```

The second plan violates the preference once, in its fourth action, and is still the better one: $36 + 3 = 39$. Figure 14.4 draws the two plans as functions of the weight: they cross at $w = 6$.

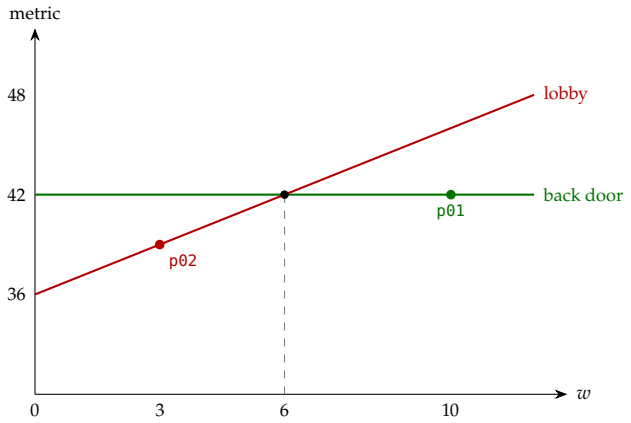


Figure 14.4.: The metric of the two routes as the weight w of `clean-lobby` grows. Below 6 the planner crosses the lobby; above 6 it takes the back door. The dots are the plans that Kairos returned for the two problems.

When there is no way around, the preference is paid as many times as it is broken. `courier-lobby-p03.pddl` has no back door and two samples.

Kairos returns a single plan of 15 actions and cost 106: 86 of effort, and two entries into the lobby with a sample at 10 each.

A preference in a precondition never blocks the action. It is violated once each time the action is applied without it, and the plan pays for every time.

A preference in a precondition is not a precondition

Reading `courier-lobby-domain.pddl` quickly, the third condition of `move` looks like a rule of the hospital. It is not: the planner will break it whenever that is cheaper, and it will break it for free if the metric does not mention `clean-lobby`. If a condition must never be broken, write it without `preference`. With the lobby rule as a hard precondition and no back door (`courier-lobby-hard-p02.pddl`), the sample cannot leave the pharmacy:

```
$ 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. Two more worlds

A tourist's day

A tourist has 240 minutes before dinner and four sights around the hotel. Walking between places takes time, every visit takes time, and a bus joins any two places in 5 minutes. The tourist must be back at the hotel for dinner, would like to see everything, and would rather walk. Listing 14.2 is the domain: plain numeric planning, with a fact `(used-bus)` that remembers whether the bus was taken.

```
;; 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)))
```

```
33
34  (:action visit
35    :parameters (?s - sight)
36    :precondition (and (at ?s)
37                       (>= (minutes-left) (visit-time ?s)))
38    :effect (and (seen ?s)
39                 (decrease (minutes-left) (visit-time ?s)))))
```

Listing 14.2: The tourist domain: minutes are the resource that runs out.

The goal of `tourist-p01.pddl` has the three kinds of condition of this chapter: a hard goal, positive wishes with different worths, and a wish for something *not* to have happened.

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

The visits alone add up to 220 minutes, so the four sights do not fit on foot. Kairos prints four plans, of cost 14, 5, 4 and 3; the last one is:

```
$ 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)
```

The tourist sees the three sights that are worth most, skips the market (2) and takes the bus once (1), for the way back: on foot from the tower the hotel is 30 minutes away, and only 5 are left. The wish `on-foot` is a soft goal about a fact that records history. It costs the same for one bus ride or for four, because the last state only knows whether `(used-bus)` is true. A tourist who minds *each* ride would put the preference in the precondition of `bus` instead.

For a tourist who really dislikes buses, `tourist-p02.pddl` raises the weight of `on-foot` to 6, more than any single sight. The best plan then costs 4: museum, market and cathedral, all on foot, and the tower is left for another day.

A satellite with more requests than time

Preferences are not tied to instantaneous actions. An observation satellite passes over a region for 60 seconds. In that time it can turn its camera to a target (a `slew`, whose duration depends on the angle) and take an image (10 seconds, during which it must keep pointing at the target and the region must be in view). There are five requests, two of them urgent. Listing 14.3 uses the durative actions of Chapter 10, and the problem

closes the pass with a timed initial literal of Chapter 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)))))
```

Listing 14.3: The satellite domain: durative actions inside a pass that ends.

The targets have two subtypes, and the goal has one preference per class, each under a `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))))
```

With three routine targets, a weight of 5 makes the urgent class a strict priority ($5 > 3 \times 1$).

```
$ 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]
```

The second plan images the two urgent targets and stops at 47 seconds with nothing else in reach. The third one is cleverer: going from `fire` to `flood` *through* `harbor` takes the same 15 seconds of turning as going directly, and the stop in the middle leaves room for one routine image (Figure 14.5). Two routine requests stay unserved, and the cost is 2. The cost and the makespan are now different numbers: the metric counts violations, not time.

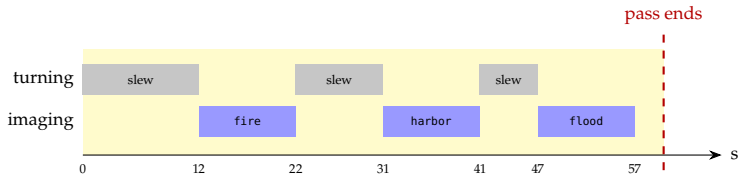


Figure 14.5.: The best plan for the satellite: three images inside the 60 seconds of the pass. The routine image of `harbor` fits because the harbor lies on the way from one urgent target to the other.

14.8. What the planner does with a preference

A classical planner searches for a state that satisfies the goal. A preference is not part of that question, so planners rewrite the problem into one they can search. The usual rewriting, due to Keyder and Geffner Keyder and Geffner, 2009, turns a soft goal into ordinary actions with costs (Figure 14.6):

- A new action *ends* the plan. It requires the hard goals and, once applied, no action of the domain is applicable any more.
- After the end, every preference must be settled by one of two new actions: **collect**, which requires that the condition of the preference holds and costs nothing, or **forgo**, which requires nothing and costs the weight of the preference.
- The new goal is that every preference is settled.

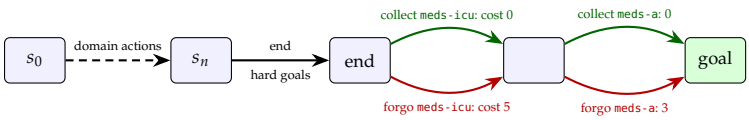


Figure 14.6.: Soft goals compiled into costs. After the real plan ends, each preference is either collected for free, if its condition holds, or forgone at the price of its weight. The cheapest path to the new goal is the best plan of the original problem.

The result is a problem with hard goals and action costs only, which any cost-aware planner of Chapter 9 can solve, and whose heuristic "sees" the wishes: a state from which `meds-icu` can still be collected looks cheaper than one from which it must be forgone. The collect and forgo actions are removed from the plan before it is printed.

Kairos follows this scheme. `kairos --features` describes how far:

```
$ 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
...
```

In plain words: preferences in goals, alone or under a `forall`, are compiled as in the figure. Preferences written directly in the `and` of a precondition are compiled too, by making copies of the action (one that satisfies the preference and one that violates it and pays), for up to four preferences per action. Any other preference (a fifth one in the same action, or one nested deeper in a condition) is not compiled but *counted*: the search does not see its cost, each plan found is priced exactly, and the anytime loop keeps only the plans that improve. The answer is the same kind of answer; it may take longer to get good. The preferences "over constraints" are the subject of Chapter 15.

Two consequences are visible from outside. The search starts from the plan that is cheapest to *find*, with soft goals often the empty one, and improves from there. And when a sequential search ends by itself, as in all the courier examples of this chapter, the last plan is optimal for the metric: `kairos -v` reports `solved (proven optimal)`. On temporal problems such as the satellite the search ends without that proof.

PDDL3 and the fifth competition

Preferences and the trajectory constraints of the next chapter arrived together in PDDL3, the language that Alfonso Gerevini and Derek Long defined in 2005 for the fifth International Planning Competition, held in 2006 (A. Gerevini & Long, 2005). Until then the competition had measured plan quality by length, cost or makespan; PDDL3 let a problem say which goals and which properties of the plan were desirable, and how much. The report of that competition (A. E. Gerevini et al., 2009) describes the language, the benchmark domains written for it and how the planners of the time coped with them.

14.9. The MODEL method: wishes and their worth

Preferences change the first step of the method more than any other. Until now "what counts as solved" had one answer. From now on, step M asks for a list.

New questions for step M (Map the situation)

- **Which goals are required and which are wishes?** For each condition of success, imagine the plan that leaves it out: unusable (hard goal) or only worse (preference).
- **What is each wish worth?** In which unit, and compared to what: to the other wishes only, or also to the effort (distance, time, money) of the plan.
- **Is any wish about the way and not about the end?** Something that should not happen during the plan, even if it leaves no

trace in the last state.
- **What is the plan when nothing is granted?** If the empty plan is acceptable, say so; if it is not, some goal is hard and is missing from the list.

The model card gains a *Wishes* table under *Success*: name, condition, weight, and the reason for the weight.

Steps O and D barely change: a preference uses the predicates and actions that are already there. They are touched only when a wish is about the way and nothing in the state records it, as with `(used-bus)`: then step O adds the fact, and step D makes the action set it. Step E gains two tests of its own. The first is to run the problem with all weights at zero or without metric and check that the plan obtained is the one expected when nobody wishes anything. The second is the *price test*: for each wish, raise its weight until the plan changes, and check that the weight at which it changes makes sense (Figure 14.4 is such a test).

Alternatives compared

A wish about the end has two encodings, and a wish about the way has three. We compare the second case on the rule of the lobby, with the back door open. The three models are in the examples of the chapter, and the three were run on the same problem.

Hard precondition (`courier-lobby-hard`). The condition is written without `preference`: a robot with a sample cannot enter the lobby.
Preference (`courier-lobby`). The model of Section 14.6, with weight 10.
Action cost (`courier-lobby-cost`). No preference: `move` has a conditional effect (Chapter 6) that adds 10 to `total-cost` when it enters the lobby with a sample.

```
(when (and (= ?to lobby)
           (exists (?s - sample)
             (carrying ?r ?s)))
  (increase (total-cost) 10))))
```

All three return the same five actions through the back door, with cost 42. They differ in everything else (Table 14.3).

	Hard precondition	Preference	Action cost
Fidelity	Loses solutions: without the back door there is no plan at all	Exact: allowed, at a price	Exact, the same plans and the same prices
Size	No growth	One copy of `move` per compiled preference, plus the collect and forgo actions of goal preferences	No growth; one conditional effect
Guidance	Best: forbidden moves are pruned	Good when compiled: the heuristic sees the price	Depends on the planner: Kairos does not use a conditional cost to guide the search
Clarity	A rule, read as a rule	The wish and its price are separate: the domain says what is wished, each problem says how much	The price is buried in the domain, the same for every problem
Portability	Negative and disjunctive preconditions	`:preferences`: Kairos, OPTIC, POPF and ENHSP among the planners tested	`:conditional-effects` and `:action-costs` together

Table 14.3.: Three encodings of "do not cross the lobby with a sample", judged on the five criteria.

The Guidance cell of the third column comes from running it. Kairos solves `courier-lobby-cost-p01.pddl`, but adds a note:

```
$ 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]
```

A cost that depends on the state cannot be given to the search as the cost of an action, so the search falls back to the shortest plan, which here happens to be the best one.

Verdict. Use a *hard* goal or precondition when the plan that breaks the condition is unusable: safety, physics, the law. A preference with a very large weight is not a substitute: it still allows the violation, and it makes the search weigh huge numbers against small ones. Use a *preference* when breaking it is acceptable at a price, and above all when the price changes from one problem to the next: the weights live in the problem file, and the domain is written once. Use an *action cost* when the thing to avoid is the action itself, always, whatever the state (a toll, a slow elevator), and not a condition under which it is applied. And when a wish about the end competes only with other wishes, a soft goal is the only one of the three that expresses it: neither a hard goal nor an action cost can say "deliver as many as possible".

14.10. Common mistakes

Preferences without a metric

A preference that the metric does not mention costs nothing to violate. The file `courier-battery-nometric.pddl` is `p02` without its `:metric`:

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

An empty plan of cost 0, and no warning: the PDDL is correct. Without a metric Kairos minimizes the number of actions, and the shortest plan that reaches no required goal is no plan. *Fix*: every preference appears in the metric, or it should not be in the model. The same happens to a preference without a name, such as `(preference (item-at meds1 ward-a))`: the syntax allows it, but nothing can refer to it.

A name in the metric that no preference has

A typing error in a name is caught by `kairos check`, which is the reason to run it first:

```
$ 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
```

The opposite error, a name misspelled in the same way *both* in the goal and in the metric, is not an error at all. Read the names once more.

Maximizing the violations

`is-violated` counts what went wrong. Writing `maximize` because "we want the most deliveries" asks for the plan that fails the most:

```
$ 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]
```

The empty plan violates all three preferences and is already the best plan there is. *Fix*: `minimize` a sum of violations. To maximize a benefit, write it as a constant minus the violations.

Taking the first plan

With soft goals the first plan is often the empty one. Asking for a single plan returns it:

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

Fix: let the anytime search run and take the last plan, or give it a goal with `--target-cost`. A system that calls the planner with a short timeout must know that what it gets may be far from the best.

Weights in the wrong unit

In a net benefit metric the weights are compared with the cost of the actions. A weight of 3 for a medicine, copied from a problem where only wishes were compared, means "worth three meters" in `courier-cost`, and the robot will never move for it. The symptom is a plan that does less than expected and is `proven optimal`. *Fix*: do the price test of Section 14.9, and write next to each weight what it is measured in.

14.11. Summary

A preference is a named condition that a plan should satisfy and may violate. In a goal it speaks about the last state; in a precondition, about each application of an action. The metric gives each violation a price, and the planner looks for the plan that pays least. With that, a problem that asks for more than its resources allow has an answer: the best subset of what was asked.

Table 14.4.: The constructs of this chapter.

Construct	Meaning
`:preferences`	The requirement
`(preference p C)` in the goal	Soft goal: `C` should hold in the last state
`(preference p C)` in a precondition	`C` should hold each time the action is applied; it never blocks the action
`(forall (?x) (preference p C))`	One preference per object, all named `p`
`(is-violated p)`	In the metric: how many preferences named `p` are violated (goal), or how many times (precondition)
`(* w (is-violated p))`	A weight: the price of each violation
`(+ (total-cost) ...)`	Net benefit: wishes against effort, in one unit

For the modeler, the lesson is in step M: separate what is required from what is wished, and give every wish a worth that can be defended. Preferences about the way opened a door that the next chapter goes through: Chapter 15 introduces conditions on all the states that a plan visits, hard or as preferences.

14.12. Further reading

The definition of preferences, with their syntax and the exact meaning of `is-violated`, is in the report that introduced PDDL3 (A. Gerevini & Long, 2005); Sections 2 and 3 cover what this chapter used. The article on the fifth competition (A. E. Gerevini et al., 2009) adds the benchmark domains with preferences, good sources of modeling ideas, and the results of the planners. The textbook by Haslum et al. (2019) treats preferences together with trajectory constraints and discusses their compilation into costs. OPTIC (Benton et al., 2012) is a temporal planner built around preferences, and its article explains what makes them hard when time is involved. For how Kairos handles them, see its documentation (Martín Rico et al., 2026).

14.13. Exercises

Exercises for Section 14.1 — More wishes than battery

Exercise 14.1.1. Without the planner, and using the distances of Figure 14.1, compute the battery that `r1` needs to deliver `meds2` to the ICU and then `sample1` to the lab, starting at the dock. Check your number against Figure 14.2.

Exercise 14.1.2. Find, with Kairos, the smallest initial battery for which `courier-battery-p01.pddl` has a plan. Before running anything, compute by hand the battery needed by two different orders of the three deliveries. Which order does the plan of Kairos follow, and why is it the cheap one?

Exercises for Section 14.2 — A preference in the goal

Exercise 14.2.1. Suppose the goal of `courier-battery-p02.pddl` gains a fourth preference, `(preference at-lobby (robot-at r1 lobby))`, added to the metric with weight 1: the day shift would like to find the robot in the lobby. What is the cost of the empty plan? And of the best plan? Answer first, then run it.

Exercises for Section 14.3 — What each wish is worth

Exercise 14.3.1. Using Table 14.1, choose weights for the three wishes of `p03` so that the best plan delivers `meds1` and `sample1` and gives up the ICU. Verify with Kairos.

Exercise 14.3.2. A hospital has three classes of deliveries: 2 urgent, 4 normal and 10 optional. Give one weight per class so that the classes are strict priorities: no number of optional deliveries is worth one normal delivery, and no number of normal and optional ones is worth an urgent one.

Exercise 14.3.3. In `courier-battery-p04.pddl` the wish `parked` has weight 4. For which weights does the best plan end at the dock? Reason with the table of the section, then test a weight on each side of your answer.

Exercises for Section 14.4 — Wishes against effort

Exercise 14.4.1. In `courier-cost-p01.pddl` the meal is worth 20 and is not delivered; in `p02` it is worth 100 and is. Find the exact weight at which the planner changes its mind, and explain what Kairos returns when the weight is exactly that one.

Exercises for Section 14.5 — One preference, many instances

Exercise 14.5.1. In `courier-cost-p03.pddl`, raise the weight of `meds-icu` to 70. Predict the cost of the best plan and the value of `(is-violated meds-icu)` in it; then run it.

Exercise 14.5.2. Rewrite the goal of `courier-cost-p03.pddl` with the `forall` inside the preference. Without running it: what is the metric of the plan that delivers one medicine? Which plans can be optimal now?

Exercises for Section 14.6 — Wishes about the way

Exercise 14.6.1. According to Figure 14.4, the two routes tie when the weight of `clean-lobby` is 6. Run `courier-lobby-p01.pddl` with weights 6 and 7 and explain both outputs.

Exercise 14.6.2. The night shift. Patients sleep in ward A and ward B, and the hospital would rather not have robots entering those rooms at night, loaded or not. Write a domain `courier-night` from `courier-cost`, with a static predicate `(quiet ?l - location)` and a preference in `move`. Then write a problem in which `r1` must take `meds2` from the pharmacy to the ICU and be back at the dock, with each entry into a quiet room priced at 25 meters. How many times is the preference violated in the best plan, and why can it not be fewer?

Exercise 14.6.3. A student models the rule of the lobby as a soft goal: `(preference clean-lobby (not (robot-at r1 lobby)))`. Explain what this wish means and why it does not express the rule.

Exercises for Section 14.7 — Two more worlds

Exercise 14.7.1. In `tourist-p01.pddl`, being back at the hotel is a hard goal. Turn it into a preference `dinner` with weight 1 and run the problem. What does the tourist do now, and what does that say about the weight?

Exercise 14.7.2. Extend `satellite-p01.pddl`: the pass lasts 80 seconds and there is a fourth routine target, `volcano`, 5 seconds of slew away from every other direction. How many requests are left unserved in the best plan that Kairos finds?

Exercises for Section 14.9 — The MODEL method: wishes and their worth

Exercise 14.9.1. Apply step L to the wish “end the morning at the dock” of `courier-battery-p04.pddl`. Compare three encodings on the five criteria: a hard goal, the preference of the chapter, and no goal at all with a robot that is simply sent home by a second problem afterwards. When is each one right?

Exercise 14.9.2. Step M from a story. “Tonight I have three hours. The lab report is due tomorrow (70 minutes). I should do the exercises of algebra (50), I would like to read the chapter for Friday (40), go to the gym (45) and do the laundry (15). After the gym I am tired, and I work badly when tired unless I rest for 20 minutes.” Write the *Success* part of the model card with its table of wishes, then a domain and a problem. Let the planner decide the evening and read its plan.

15. Constraints on the whole plan

A goal judges a plan by its last state, and a precondition judges one action at the moment it is applied. Many rules of a real workplace are neither. "No waste in the ICU" is not about where things end up: the waste ends at the dock in any case. "Sanitize before entering the lab" is not about one action: it relates something that happens now to something that must have happened earlier. "Visit the dock at most once" counts. The preference in a precondition of Chapter 14 looked at the way for the first time, but only through the window of a single action. A hospital that hands its rulebook to the modeler expects every plan to respect it from the first step to the last.

PDDL 3.0 has a section for the rulebook: `:constraints`. A **trajectory constraint** is a condition on the whole sequence of states that a plan goes through, written with one of ten operators, from `always` and `sometime` to deadlines such as `within` and windows such as `hold-during`. A plan that reaches the goal and breaks a constraint is not a plan. And a constraint wrapped in a preference becomes a rule that may be broken at a price.

The chapter starts with a plan that breaks a rule which the model had no way to state. It then defines the sequence of states of a plan, on which everything else rests, and introduces the operators in three groups: what must always or sometimes be true, what must happen in a certain order, and what must happen by a certain time. One section sets the ten side by side, each with its exact meaning and with a plan that satisfies it and a plan that violates it. Then come constraints in the domain and in the problem, constraints as preferences, a rover and a workshop, what a planner does with a constraint, and the comparison that the MODEL method asks for: a constraint, preconditions on every action, or an automaton written by hand.

15.1 A rule that the plan breaks 327
15.2 The states of a plan: always 329
15.3 What must happen, and in which order 330
sometime and at end . . 330
at-most-once 330
sometime-before and sometime-after 331
15.4 Rules with a clock . . . 332
Counting steps 332
Counting seconds 333
15.5 The ten operators, side by side 334
15.6 Where rules live, and rules that bend 336
In the domain or in the problem 336
Constraints as preferences 337
15.7 Two more worlds 337
A rover and a crater . . 337
A workshop with an oven 338
15.8 What the planner does with a constraint 339
15.9 The MODEL method: rules about the whole plan 340
Alternatives compared . 341
15.10 Common mistakes . . . 342
15.11 Summary 344
15.12 Further reading 344
15.13 Exercises 345

15.1. A rule that the plan breaks

The courier of this chapter has a basket. It can carry any number of items, so the predicate `(free ?r)` of the typed domain disappears and `pick` loses that precondition. Items have three subtypes, `medicine`, `sample` and `waste`. The robot can also sanitize its gripper where there is a gel dispenser, and picking anything up dirties it again. Listing 15.1 is the whole domain, `courier-rules`: classical STRIPS with types. It declares `:constraints` because its problems will use them.

```
;; 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)))
```

Listing 15.1: The domain of the chapter: a courier with a basket and a gel dispenser. Nothing in it mentions a rule.

In `courier-rules-p01.pddl` a box of medicines and a bag of waste wait in the pharmacy. The medicines go to the ICU, the waste to the dock, and the robot must end at the dock.

```
$ 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)
```

Fourteen actions, the shortest plan there is, and no hospital would accept it: the robot loads everything in one go and rides into the intensive care unit with a bag of waste in its basket. The model has no fault that step E could have found. Every action is legal, the goal is reached, and nothing false was assumed. What is missing is a rule of the hospital, *no robot is ever in the ICU while it carries waste*, and the question is where such a rule can be written.

Not in the goal: in the last state the waste is at the dock. A precondition of `move` could forbid entering the ICU with waste, but waste can also be picked up *inside* the ICU, so `pick` needs its own precondition, and so does every action that a later version of the domain adds. The rule is one sentence about states, and it would be scattered over all the actions that might produce such a state.

Some rules are not about the end of the plan nor about one action: they are about every state on the way. Goals and preconditions can only express them piece by piece.

15.2. The states of a plan: always

A plan of n actions, applied from the initial state, goes through $n + 1$ states:

$$S_0, S_1, S_2, \ldots, S_n$$

where S_0 is the initial state and S_i is the state after the i-th action. This sequence is the **trajectory** of the plan. Note that it includes both ends: the initial state is part of the trajectory, and so is the last one.

The plan above has 14 actions and a trajectory of 15 states. Figure 15.1 draws two facts along it: whether the robot is in the ICU, and whether it carries the waste. The rule of the hospital is broken in the states where both are true, S_8 and S_9.

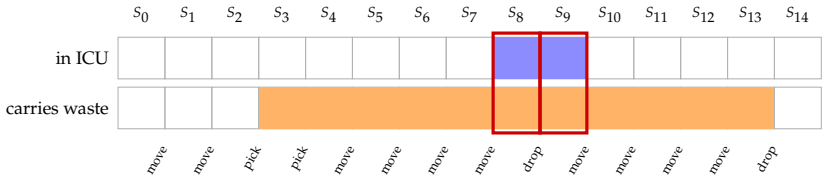


Figure 15.1.: The trajectory of the 14-action plan. Each column is a state; a filled cell means that the fact holds in it. A goal looks only at S_{14}. The rule "never in the ICU with waste" looks at every column, and fails in the two framed in red.

A **trajectory constraint** is a condition on the trajectory. The simplest one says that a condition holds in every state: `(always C)`. Constraints go in a section of their own, `:constraints`, which in a problem is written after the goal. The requirement is also called `:constraints`. `courier-rules-p02.pddl` is `p01` plus the rule:

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

The `forall` makes one constraint for every robot and every item of type `waste`; a section with several constraints joins them with `and`. The condition inside `always` is an ordinary one, of the kind used in goals: here a negated conjunction, which is why the problem also declares `:negative-preconditions` and `: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)
```

The robot still loads both items at the pharmacy, but now it gets rid of the waste at the dock before it goes up to the ICU. The plan is two actions

longer: that is the cost of the rule. The domain did not change by a single character.

The validator checks constraints too. The short plan of `p01`, saved as `courier-rules-p02-bad.plan`, is valid for `p01` and invalid for `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
```

The trajectory of a plan is the sequence of its states, the initial one included. An always constraint must hold in every one of them.

Because S_0 belongs to the trajectory, an `always` that is false in the initial state can never be satisfied: the problem is unsolvable before the first action. When a problem with constraints has no plan, the initial state is the first place to look.

15.3. What must happen, and in which order

`always` is the operator of prohibitions. Four others speak about things that must happen at some point, how often, and in which order. Each is shown here on a small problem of `courier-rules`; the plans that satisfy and violate each one are collected in Section 15.5.

sometime and at end

`(sometime C)` asks that `C` holds in at least one state of the trajectory. In `courier-rules-p03.pddl` the robot collects waste from ward B, and the ICU nurse wants to see it pass by during the round:

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

The plan of Kairos has 10 actions instead of 8: on the way it steps into the ICU and out again. A `sometime` is a goal that does not have to last. No goal could say this: at the end the robot is at the dock, not in the ICU.

`(at end C)` asks that `C` holds in the last state. That is what a goal does, and the operator exists so that goals and constraints can be written, and preferred, in the same section. A condition written in `:constraints` without any operator means the same.

at-most-once

`(at-most-once C)` asks that `C` becomes true at most one time: it may hold during one stretch of consecutive states, of any length, and never again after that stretch ends. It may also never hold. The door of the dock is slow and noisy, and the technicians want it used once per round. In `courier-rules-p04.pddl` the robot starts in the lobby, waste waits in ward A for the dock, and medicines wait at the dock for ward A:

```
(:constraints
  (at-most-once (robot-at r1 dock))))
```

The short plan would fetch the medicines first (the dock is one step away), deliver them, come back with the waste: 11 actions and two visits. With the constraint Kairos returns 12 actions: the robot walks to ward A with an empty basket, brings the waste, and in a single visit to the dock drops it and picks the medicines up.

The stretch may be long. The robot stays at the dock for three states (it arrives, drops, picks) and that is one visit. What is forbidden is to leave and come back.

sometime-before and sometime-after

Two operators relate two conditions.

`(sometime-before A B)`: whenever `A` holds in a state, `B` held in some *earlier* state. "Before you do `A`, you must have done `B`." If `A` never holds, nothing is asked.

`(sometime-after A B)`: whenever `A` holds in a state, `B` holds in that state or in some *later* one. "If you do `A`, you must do `B` afterwards."

The lab asks for robots to sanitize before they come in. The problem `courier-rules-p05.pddl` sends a sample from ward A to the lab under that rule:

```
  (: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)
```

The plan has the `sanitize` action that the rule forced. Read where it is, though. The robot sanitizes in the corridor on its way *out*, then picks the sample, which dirties the gripper again, and enters the lab with a dirty gripper. The constraint is satisfied to the letter: `(sanitized r1)` held in a state earlier than the first state in the lab. It is the rule that was written wrongly.

Step E: the cheat test for a rule

The fourth test of step E tempts the planner to cheat, and a rule is one more thing to cheat on. For every constraint, look for the cheapest way to satisfy its words without doing what was meant. `sometime-before` asks for "at some earlier time", however long ago and whatever happened in between. If the hospital means "clean *when* it enters", the rule is about a single state, and the operator for that is `always`: `(always (imply (robot-at r1 lab) (sanitized r1)))`. Exercise 15.3.3 runs it.

`sometime-after` is the mirror image. In `courier-rules-p06.pddl`, a robot that has been in the ICU must go to the dock afterwards to be disinfected:

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

The goal is reached with the `drop` in the ICU, after 7 actions. The plan of Kairos has 11: four more moves take the robot to the dock, although no goal asks for it. The constraint created an obligation in the middle of the plan that the plan must discharge before it may end.

sometime: at least once. at-most-once: one stretch at most. sometime-before: only after B has happened. sometime-after: if A happens, B follows.



Figure 15.2.: Four operators as patterns over a trajectory of 12 states (time runs to the right; a filled cell means that the condition holds). On the right: `C` comes back after a gap; the first `A` has no `B` before it; the last `A` has no `B` after it; the second `A` waits 4 steps for its `B`.

15.4. Rules with a clock

The remaining four operators carry numbers: deadlines and windows. To give them a meaning, every state of the trajectory needs a time. Write t_i for the time at which state S_i begins, with $t_0 = 0$.

In a temporal plan (Chapter 10) the states change at the happenings, the starts and ends of the durative actions, and t_i is the time of the i-th happening, in the unit of the durations. In a sequential plan there is no clock, and Kairos uses the position in the plan: state S_i is at time i. A deadline of 7 in a sequential plan means "within the first 7 actions".

- `(within t C)`: `C` holds in some state at time t or earlier. A deadline.
- `(always-within t A B)`: whenever `A` holds in a state, `B` holds in that state or in a later one that is at most t later. A deadline that starts to run every time `A` is true.
- `(hold-during t1 t2 C)`: `C` holds in every state whose time is at least t_1 and less than t_2. A window.
- `(hold-after t C)`: `C` holds in every state whose time is later than t. A window that never closes.

Counting steps

Four problems of `courier-rules` use them, with time in steps.

`p07`: the medicines must be in ward A within 7 steps, `(within 7 (item-at meds1 ward-a))`, while a bag of waste in the lobby also has to go to the dock. Seven steps is exactly what the delivery needs from the dock, so

the robot must leave the waste for later, although it walks past it at the first step.

`p08`: a sample that has been picked up must be in the lab at most 3 steps later, `(always-within 3 (carrying r1 sample1) (item-at sample1 lab))`. From ward A that allows the two moves and the drop, and nothing else: no stop on the way.

`p09`: the corridor is mopped from step 3 to step 6, `(hold-during 3 6 (not (robot-at r1 corridor)))`, and the medicines must go from the pharmacy to ward A, through the 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)
(move r1 pharmacy lobby)
(move r1 lobby corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

The direct plan has 7 actions and is in the corridor at step 5 (Figure 15.3). A sequential plan cannot wait, because time only passes when an action is applied. So the planner *invents* a wait: the third and fourth actions walk to the lobby and back to the pharmacy, two steps that do nothing but let the clock run. The robot then enters the corridor at step 7, when the window has closed.

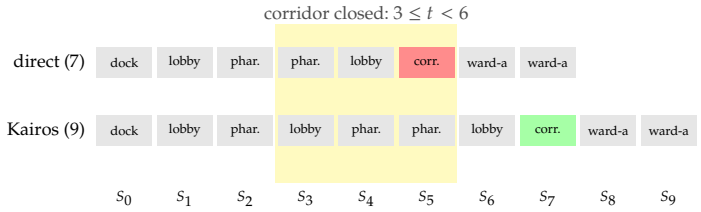


Figure 15.3.: Where the robot is in each state of two plans for `courier-rules-p09`. The window covers the states at times 3, 4 and 5. The direct plan is in the corridor in S_5; the plan of Kairos wastes two moves and enters it in S_7.

`p10`: two bags of waste in the lobby, and a curfew: from step 4 on the robot is at the dock, `(hold-after 4 (robot-at r1 dock))`. The states after time 4 are S_5, S_6 and so on. Carrying one bag at a time puts the robot back in the lobby at step 5; the plan of Kairos takes both bags in one trip, six actions, and is at the dock from step 4.

Counting seconds

With durative actions the numbers are times of the clock. The domain `courier-timed` is the same courier with the durative actions of Chapter 10: a move lasts its distance in seconds, a pick or a drop 2 seconds, and `(idle ?r)` keeps the robot to one action at a time. `courier-timed-p01.pddl` asks for the two errands of `p07` with a deadline on the clock:

```
(: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]
```

The medicines are in ward A at 44.007, when the `drop` ends, inside the deadline. This time the robot does pick the waste on its way: it costs 2 seconds, and 45 leaves room for them. Taking the waste to the dock first would deliver the medicines at second 54; that plan is in `courier-timed-waste-first.plan`, and `kairos validate` rejects it.

A window in seconds is written the same way. In `courier-timed-p02.pddl` the corridor is closed to robots from second 20 to second 40, `(hold-during 20 40 (not (robot-at r1 corridor)))`. The best plan that Kairos finds takes the waste to the dock first and enters the corridor after the window:

```
$ 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]
```

A temporal plan can wait: the robot simply stands in the lobby between 34 and 40. The plan is valid but not the fastest one: the plan in `courier-timed-waste-first.plan` does the same errands without the pauses, in 54 seconds, and is valid too. Chapter 10 already warned that the makespan of a temporal plan is good, not proven minimal.

In a sequential plan the time of a state is its position; in a temporal plan, the clock. within is a deadline, always-within a deadline that restarts, hold-during and hold-after are windows.

Chapter 13 modeled a deadline with a timed initial literal: a fact that becomes false at the deadline and that the delivering action requires. `within` says the same in one line and without touching the domain, and it also works in sequential plans, which have no clock for a timed literal to follow. The timed literal is more widely supported. Section 15.10 shows a case where the two do not behave the same.

15.5. The ten operators, side by side

Table 15.1 gives the meaning of the ten operators over a trajectory $S_0, \ldots, S_n$ whose states begin at times $t_0 = 0 \leq t_1 \leq \ldots \leq t_n$. We

write $S_i \models C$ for "the condition `C` holds in state S_i".

Constraint	Holds for the trajectory when
`(at end C)`	$S_n \models C$
`(always C)`	$S_i \models C$ for every i
`(sometime C)`	$S_i \models C$ for some i
`(at-most-once C)`	if $S_i \models C$, there is $j \geq i$ such that C holds in $S_i, \ldots, S_j$ and in no state after S_j
`(sometime-before A B)`	for every i with $S_i \models A$ there is $j < i$ with $S_j \models B$
`(sometime-after A B)`	for every i with $S_i \models A$ there is $j \geq i$ with $S_j \models B$
`(within t C)`	$S_i \models C$ for some i with $t_i \leq t$
`(always-within t A B)`	for every i with $S_i \models A$ there is $j \geq i$ with $S_j \models B$ and $t_j - t_i \leq t$
`(hold-during t1 t2 C)`	$S_i \models C$ for every i with $t_1 \leq t_i < t_2$
`(hold-after t C)`	$S_i \models C$ for every i with $t_i > t$

Table 15.1.: The trajectory constraints of PDDL 3.0 as conditions on the sequence of states. The last two rows are stated as Kairos checks them, over the states that begin inside the window; Section 15.10 discusses the edges.

Three details of the table are easy to miss. In `sometime-before` the index j is *strictly* smaller than i: `B` in the same state as `A` is not before. In `sometime-after` and `always-within` it is $j \geq i$: `B` in the same state is enough. And every operator that starts with "if `A` holds" is satisfied by a plan in which `A` never holds, the empty plan included.

Table 15.2 is the same list as an experiment. Each row is a problem of `courier-rules` with one constraint and two plans written by hand: every file `-ok.plan` is accepted by `kairos validate`, and every file `-bad.plan` is rejected as a plan that violates the constraints of the problem. The bad plan is always a valid plan of the same problem without its constraint.

Problem and constraint	`-ok.plan`	`-bad.plan`
`p02` `always` not in the ICU with waste	drops the waste at the dock, then goes to the ICU	carries both items into the ICU (S_8, S_9)
`p03` `sometime` in the ICU	steps into the ICU (S_4)	collects the waste and returns
`p04` `at-most-once` at the dock	one visit (S_6 to S_8)	two visits (S_1, S_2 and S_{10}, S_{11})
`p05` `sometime-before` lab, sanitized	sanitized in S_6, in the lab in S_7	never sanitized
`p06` `sometime-after` ICU, dock	ends at the dock (S_{11})	ends in the ICU
`p07` `within 7` medicines in ward A	delivered in S_7	delivered in S_{11}
`p08` `always-within 3` carrying, in the lab	picked in S_4, in the lab in S_7	stops to sanitize: S_4 to S_8
`p09` `hold-during 3 6` not in the corridor	enters the corridor in S_7	in the corridor in S_5
`p10` `hold-after 4` at the dock	at the dock from S_4	back in the lobby in S_5

Table 15.2.: One problem per operator, each with a plan that satisfies its constraint and a plan that violates it. The files are `courier-rules-pNN.pddl`, `courier-rules-pNN-ok.plan` and `courier-rules-pNN-bad.plan`; all 18 plans are in the tests of the chapter. (`at end` is the goal.)

```
$ 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
```

The message names the section, not the constraint. With several constraints in a problem, the way to find the broken one is to validate the plan against copies of the problem that keep one constraint each.

What the operators cannot say

The ten operators are a small, fixed vocabulary. In PDDL 3.0 they cannot be nested: there is no `(always (sometime C))`, and the conditions inside them are ordinary conditions on one state. So "*every time* the robot touches waste it sanitizes before the *next* time it enters the lab" has no direct translation: `sometime-before` forgets what happened in between, as `p05` showed. Such rules are written by adding a fact that remembers (the gripper is `(sanitized r1)` or not) and constraining that fact with `always`. The operators are in fact a fragment of *linear temporal logic*, the logic that Amir Pnueli proposed in 1977 Pnueli, 1977 for stating properties of programs over their runs; the full logic allows the nesting that PDDL 3.0 leaves out.

15.6. Where rules live, and rules that bend

In the domain or in the problem

A `:constraints` section can be written in the problem, as so far, or in the domain, after the predicates and before the actions. The meaning is the same; the scope is not. A constraint of the domain holds for every problem of that domain. A plan must satisfy both sections.

The rule of the ICU is not today's rule: it is the hospital's. `courier-safe` is `courier-rules` with the rule moved into the domain. To name the ICU there, the domain declares it as a constant:

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

`courier-safe-p01.pddl` states only the errands of `p02` and gets the same 16-action plan. Whoever writes the next problem cannot forget the rule.

That is also the danger. A rule in the domain is applied to problems that its author did not imagine. Step E asks for a test that must have a plan, and here is one that should: `courier-safe-p02.pddl` is the same morning with the bag of waste lying in the ICU itself.

```
$ 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)
```

The rule says that no robot is ever in the ICU carrying waste, and picking the bag up in the ICU produces exactly that state. Read literally, the rule forbids cleaning the ICU. The hospital meant "do not bring waste *into* the ICU", which is a different sentence, and finding the difference is what the test was for.

Step M: whose rule is it

For every rule in the story, the model card records its **scope**: a law of the world (nothing can break it: it belongs in preconditions and effects), a rule of the organization (every problem: a constraint of the domain), or a condition of today (this problem: a constraint of the problem). And its **strength**: must it hold, or should it?

Constraints as preferences

A constraint can be wrapped in the `preference` of Chapter 14. It then becomes a **soft constraint**: a plan may violate it, and `(is-violated name)` is 1 in the metric if the trajectory does not satisfy it, 0 if it does. Under a `forall`, each instance counts.

`courier-rules-p11.pddl` has the morning of `p02` with the hard rule of the ICU and two wishes about time that pull in opposite directions:

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

Eight steps are what the medicines need if the robot goes straight to the ICU; six are what the waste needs if it goes straight to the dock. Both cannot be first. Kairos prints three plans, of cost 5, 3 and 2. The last one delivers the medicines first, in 8 steps, and the waste at step 16: it violates `early-waste` and pays 2. The hard rule is respected by all three.

Soft constraints are how a rulebook with exceptions is written: the rules that protect patients stay hard, and the rules of convenience get a price.

A constraint of the domain binds every problem; one of the problem, only that problem. Wrapped in a preference, a constraint may be violated and is paid in the metric.

15.7. Two more worlds

A rover and a crater

A rover drives between four waypoints and collects samples. Its domain, `rover-domain.pddl`, has two actions, `drive` and `collect`. Its mission is to bring the samples of the plain and of the crater (Figure 15.4). Without rules, `rover-p01.pddl` is solved in four actions through the plain. The mission control of `rover-p02.pddl` adds three rules about the crater, none of which the domain knows about:

```
(: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)
```

Each rule left its mark. The rover goes around by the ridge (first rule), enters the crater once and leaves on the other side (second), and comes home although the goal does not ask for it (third). The first rule deserves a second look: the domain has no fact "the crater has been surveyed", and none was needed. The constraint speaks about *having been* at the ridge, which is history, and the trajectory holds the history.

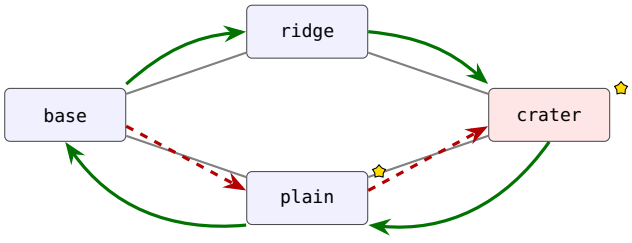


Figure 15.4.: The rover's mission: two samples (stars). Dashed red, the four-action plan without rules. Green, the plan under the three rules: by the ridge, once through the crater, and back to the base.

A workshop with an oven

A workshop bakes parts in an oven: open the door, load, close, heat (which bakes whatever is inside), cool, open, unload. Two safety rules belong to the workshop, not to any particular order of work, so they are constraints of the domain (Listing 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))))
```

Listing 15.2: The workshop: two safety rules as constraints of the domain.

The actions that follow have only the preconditions of physics: a part is loaded through an open door, the oven is heated when it is cold. In particular, nothing in `heat` or in `open-door` mentions the other. The problem `workshop-p01.pddl` asks for three parts baked and outside:

```
$ 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)
```

The `close-door` before `heat` and the `cool` before the second `open-door` come from the first rule; the three parts in one batch come from the second. Without the constraints, the same domain gives a plan of 8 actions that heats with the door open and unloads from a hot oven. A hasty plan that bakes the parts one by one (`workshop-p01-hasty.plan`) is rejected by the validator, and this time the message points at the domain: `the plan violates the domain :constraints`.

15.8. What the planner does with a constraint

A forward search builds a plan state by state. To respect a constraint it does not need the whole trajectory: it needs to remember, in each state, just enough of the past to know whether the constraint can still be satisfied. That memory is a small automaton, a **monitor**, and each operator has its own (Figure 15.5). For `always` there is nothing to remember: the condition is checked in every state. For `sometime` one bit is enough, "it has happened". For `at-most-once` there are three situations: not yet, during the stretch, after it.

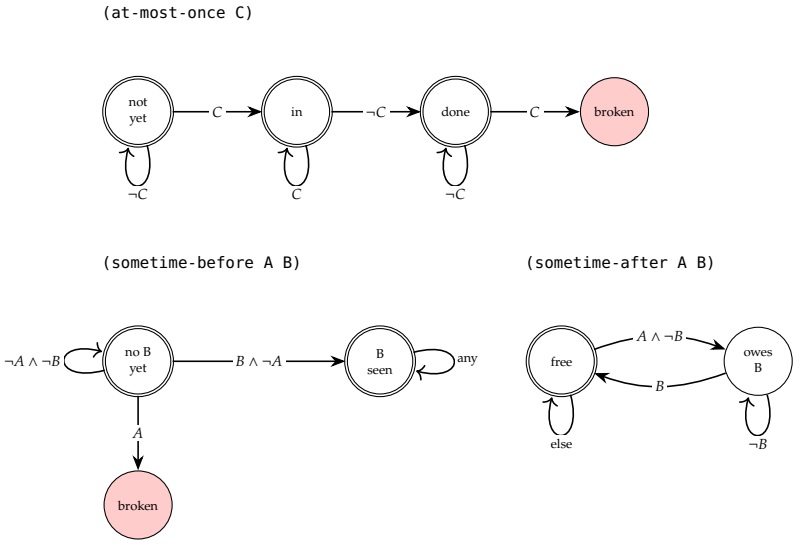


Figure 15.5.: Monitors for three operators. Each state read from the trajectory moves the monitor along one arrow. A plan may end only in a double circle; a red state can never be left. The monitor of `sometime-after` has no red state: an obligation can always still be discharged, but the plan may not end while it is owed.

A planner can run the monitors beside the search, or write them into the problem. Kairos does the second: it compiles the constraints away before searching. What a constraint forbids becomes a precondition of every action, since every state but the last is the state before some action; what it must remember becomes a new fact, kept up to date by conditional

effects added to every action; and what must be true when the plan ends is added to the goal. For the constraints with times in sequential plans it adds a counter of steps. In temporal plans the deadlines and windows become flags set at their times, the same mechanism as the timed initial literals of Chapter 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)
```

Compiling has two consequences for the modeler. The search and its heuristic see the rules as ordinary preconditions and goals, so a rule prunes the search instead of rejecting finished plans. And the cost is paid in size: each constraint adds conditions to *every* action, and a `forall` over two types adds one monitor per pair of objects.

Few planners accept this part of the language. Among those compared in the documentation of Kairos (Martín Rico et al., 2026), only Kairos and OPTIC returned valid plans for a problem with a trajectory constraint; POPF returned a plan that broke it, and the other planners rejected the file. That weighs on the Portability criterion below.

Rules for the search before rules for the plan

Temporal formulas entered planning before PDDL3, for a different purpose. TLPlan, the planner of Fahiem Bacchus and Froduald Kabanza, published in 2000 Bacchus and Kabanza, 2000, let the modeler write formulas of temporal logic that said which partial plans were not worth extending, and pruned the search with them: advice for the planner, not requirements on the answer. PDDL3 turned the same kind of formula into part of the problem itself: something a plan must satisfy, whoever finds it and however (A. Gerevini & Long, 2005).

15.9. The MODEL method: rules about the whole plan

New questions for steps M and D

Step M (Map the situation). List the rules of the story apart from its goals. For each one:

- **Is it about one action or about the whole plan?** "The robot cannot pick what is not there" is about one action. "Never in the ICU with waste" is about every state.
- **Does it look at one state or at history?** A rule about one state at a time is an `always`. A rule with "before", "after", "once", "by" needs memory.
- **Whose rule is it, and can it bend?** Domain or problem; hard or preference.

Step D (Dynamics). When an action gets a precondition, ask whether

it describes what the action *needs* to be possible or what somebody *wants* of the plan. The first is dynamics. The second is a rule, and it may be better written once, as a constraint, than repeated in every action that could break it.

The model card gains a table of rules: the sentence as the client says it, its scope, its strength, and the operator chosen, with the cheat test of step E run on each (Section 15.3).

Alternatives compared

A rule about the whole plan has three encodings. All three were written and run for this section.

A constraint. The models of the chapter: `courier-safe` for the rule of the ICU, `courier-rules-p04` for the rule of the dock.

Preconditions on every action. Possible when the rule looks at one state at a time. The domain `courier-icu-pre` writes the rule of the ICU into the two actions that can produce the forbidden state:

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

It returns the same 16-action plan as the constraint, and the same `unsolvable` when the waste lies in the ICU. But it is right only because *both* actions were found. The domain `courier-icu-leaky` has the precondition on `move` alone, the first one anybody thinks of, and on that problem it finds a 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)
...
```

Whether or not the hospital wants that plan, the model no longer says what its author believed it said.

An automaton with extra predicates. Needed without `:constraints` when the rule looks at history. The modeler writes by hand the monitor shown in Figure 15.5. For "the dock at most once", the domain

`courier-dock-auto` adds a fact `(left-dock ?r)`, sets it when the robot leaves the dock, and forbids entering the dock once it is set:

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

On the errands of `p04` it returns the same 12 actions as `(at-most-once (robot-at r1 dock))`.

Table 15.3.: Three encodings of a rule about the whole plan, judged on the five criteria.

	Constraint	Preconditions on every action	Automaton with extra predicates
Fidelity	Says the rule once, over all states; cannot leak. Can be *too* strong (waste in the ICU)	Exact only if no action is forgotten, now and in later versions; state rules only	Exact if the automaton is right and the initial state sets it right; any rule
Size	The planner adds the monitors: conditions on every action	Only the actions concerned	One or two facts per rule and object, effects on the actions concerned
Guidance	Good in Kairos, which compiles to the other two	Good: plain preconditions	Good: plain facts
Clarity	Best: the rule is one line, apart from the dynamics, and can be made soft	The rule is scattered and mixed with physics	Worst: the rule is hidden in bookkeeping
Portability	Poor: `:constraints` is accepted by few planners	Good: ADL preconditions	Good: ADL with conditional effects

Verdict. Write the rule as a constraint first, even if the final model cannot use one: it is the specification, and `kairos validate` with the constraint is the test for any other encoding. Keep the constraint when the planner supports it, when rules change from problem to problem, or when some must be soft. Move a rule into preconditions when it is about one state, few actions can break it and the model must run on classical planners. Write the automaton when the rule is about history and the planner has no `:constraints`; then validate its plans against the constraint version.

15.10. Common mistakes

Forgetting that the initial state counts

`courier-rules-once-start.pddl` asks for the dock at most once, with the robot starting *at* the dock and waste to bring to it:

```
$ 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 is part of the trajectory, so the one visit is spent before the first action, and coming back is a second one. The same holds for `sometime-before`: a `B` that is true in S_0 counts for every later `A`, but

an `A` true in S_0 has nothing before it and breaks the constraint at once. *Fix*: decide what the rule means for the state the day starts in, and write it for that.

An operator outside its section

The operators exist only inside `:constraints`. In a goal they are read as predicates that nobody declared (`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
```

A warning, not an error; then the planner answers `unsolvable`, because the goal contains a fact that is never true. *Fix*: move the condition to `:constraints`, and read the warnings of `kairos check`.

A window that the plan never reaches

`courier-rules-short.pddl` delivers medicines to ward A under a curfew, `(hold-after 10 (robot-at r1 dock))`. The plan returned has 7 actions and ends in ward A. It is valid: the constraint speaks about the states after time 10, and this plan has none. A window constrains the states that fall inside it; it does not make the plan last until the window opens. *Fix*: if the robot must be at the dock, say it also in the goal.

Standing still through a window

In a temporal plan a state begins at a happening and lasts until the next one. Kairos decides a window by the times at which states *begin*. The plan `courier-timed-standing.plan` drives the robot into the corridor at second 14 and leaves it standing there until second 45, with no happening in between:

```
$ kairos validate courier-timed-domain.pddl courier-timed-p02.
    pddl courier-timed-standing.plan
plan valid, makespan 105.01, metric 105.01
```

No state begins between 20 and 40, so `(hold-during 20 40 ...)` has nothing to check, although the robot is in the closed corridor during the whole window. *Fix*: when a window must hold at every instant, give its edges a happening of their own. In `courier-timed-p03.pddl` two timed initial literals (Chapter 13) switch a fact `(closed corridor)` on at 20 and off at 40, and the rule becomes an `always`:

```
(at 20 (closed corridor)) (at 40 (not (closed corridor)))
...
(forall (?l - location)
  (always (not (and (closed ?l) (robot-at r1 ?l)))))
```

Now a state begins at second 20 with the robot inside, and the same plan is rejected as `plan invalid`.

Steps taken for seconds

`(within 45 ...)` in a sequential problem gives the plan 45 *actions*, which is almost never a restriction, and the plan found looks as if the constraint were ignored. In a temporal problem `(within 7 ...)` means 7 seconds. *Fix*: a number in a constraint has the unit of the plan: positions if the domain has instantaneous actions, the time unit of the durations if it has durative ones. If a sequential model needs real time, it needs durative actions first.

A rule stronger than the world

A constraint cuts plans away, and with one rule too many it cuts all of them. The symptom is `unsolvable` on a problem that looked easy, as with the waste in the ICU. *Fix*: the debugging procedure of Chapter 7 with one more step at the start: remove the constraints one at a time until a plan appears, then validate that plan against each constraint alone to see which one it breaks, and where.

15.11. Summary

A plan goes through a sequence of states, its trajectory, and PDDL 3.0 lets a model state conditions on the whole of it. The section `:constraints`, in the domain or in the problem, holds them; the requirement has the same name.

Operator	Says	Typical rule
`always`	in every state	safety: never this
`sometime`	in some state	pass by, check in
`at end`	in the last state	a goal
`at-most-once`	one stretch at most	do not come back
`sometime-before A B`	`A` only after `B`	a prerequisite
`sometime-after A B`	after `A`, `B`	an obligation
`within t`	by time *t*	a deadline
`always-within t A B`	after `A`, `B` within *t*	a reaction time
`hold-during t1 t2`	in the states of a window	a closure
`hold-after t`	in the states after *t*	a curfew

Table 15.4.: The ten operators and the kind of rule each one writes.

Constraints are hard unless they are wrapped in a preference. They keep the rulebook out of the dynamics: the domain says what can be done, the constraints say what may be done. Where the planner does not accept them, the same rules are written as preconditions or as a hand-made automaton, and the constraint remains as their specification and their test.

15.12. Further reading

The report on PDDL3 (A. Gerevini & Long, 2005) defines the trajectory constraints with their semantics over timed sequences of states; compare its definitions of `hold-during` and `hold-after` with the last rows of Table 15.1. The article on the fifth competition (A. E. Gerevini et al., 2009) presents the domains written with constraints and how the competing planners treated them. Haslum et al. (2019) devote a section to trajectory constraints and to their compilation into automata. The validator VAL (Howey et al., 2004) is the usual reference for checking plans against

PDDL semantics. OPTIC (Benton et al., 2012) is the other planner of the comparison that handles constraints and preferences over them in temporal problems.

15.13. Exercises

Exercises for Section 15.1 — A rule that the plan breaks

Exercise 15.1.1. For each rule, say whether it can be written as a goal, as a precondition of one action, or needs the whole plan: (a) the robot ends at the dock; (b) the robot never carries more than three items; (c) a sample is never left in the corridor, not even for a moment; (d) the robot cannot drop what it does not carry; (e) the robot passes by the pharmacy at least once; (f) ward A is visited before ward B.

Exercises for Section 15.2 — The states of a plan: always

Exercise 15.2.1. Hygiene asks for more: waste and medicines must never be in the basket together. Add that constraint to the problem `courier-rules-p01.pddl`, with a `forall` over robots, waste and medicines, and run it. How does the plan differ from the one of `p02`?

Exercise 15.2.2. Without running anything: what does Kairos answer if the constraint of `courier-rules-p02.pddl` is replaced by `(always (not (robot-at r1 dock)))`? Why?

Exercises for Section 15.3 — What must happen, and in which order

Exercise 15.3.1. Add to `courier-rules-p03.pddl` the constraint that no room is visited twice: `(forall (?l - location) (at-most-once (robot-at r1 ?l)))`. Predict the answer from the map, then run it.

Exercise 15.3.2. What do these constraints mean? (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))` in a problem whose initial state has `(sanitized r1)`.

Exercise 15.3.3. Replace the constraint of `courier-rules-p05.pddl` by `(always (imply (robot-at r1 lab) (sanitized r1)))`. The plan printed in the section is saved as `courier-rules-p05-early.plan`: check that it is now invalid, and read the new plan. Which requirement does `imply` need?

Exercises for Section 15.4 — Rules with a clock

Exercise 15.4.1. In `courier-rules-p07.pddl`, change the deadline to 6 steps. Explain the answer of Kairos by counting actions.

Exercise 15.4.2. In `courier-timed-p01.pddl`, how tight can the deadline be? Add up the durations of the actions that the delivery needs, try a deadline of 43, and explain why 42 cannot be met although the durations add up to 42.

Exercise 15.4.3. Explain the difference between `(within 5 (sanitized r1))` and `(always-within 5 (robot-at r1 icu) (sanitized r1))`. Give a plan, in words, that satisfies the first and violates the second.

Exercises for Section 15.5 — The ten operators, side by side

Exercise 15.5.1. Take the 16 actions of `courier-rules-p02-ok.plan` and write down its trajectory: where the robot is and what it carries in $S_0, \ldots, S_{16}$. Decide by hand which of these constraints it satisfies: (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)))`. Then check with `kairos validate`.

Exercises for Section 15.6 — Where rules live, and rules that bend

Exercise 15.6.1. The workshop of Section 15.7 has "one heating cycle per shift" in its domain. Give a reason to move that rule to the problem, and a reason to keep the other rule in the domain.

Exercise 15.6.2. In the problem `courier-rules-p11.pddl`, give the wish `early-waste` a weight of 4. Predict the plan and its cost, then run it.

Exercises for Section 15.7 — Two more worlds

Exercise 15.7.1. Remove the third rule from `rover-p02.pddl` and predict the plan. Which of its actions is still due to a rule?

Exercise 15.7.2. Parts must not stay long in the oven. Add this constraint to the problem `workshop-p01.pddl`, and find the smallest `N` for which there is a plan:

```
(forall (?p - part)
  (always-within N (in-oven ?p) (outside ?p)))
```

What does the constraint ask of the order in which the parts are unloaded?

Exercises for Section 15.9 — The MODEL method: rules about the whole plan

Exercise 15.9.1. Write the rover's first rule, "the ridge before the crater", as an automaton with extra predicates, in a domain `rover-auto` without `:constraints`. Make it general: any waypoint can be marked as risky and any as a lookout in the problem. Compare its plan with the one of the constraint, and say what your encoding does if the rover *starts* at the ridge.

Exercise 15.9.2. Step M on a story. "In our kitchen the cook washes hands after touching raw meat and before touching anything else. The oven is never left on with nobody in the kitchen. Each dish is tasted before it is served. The fridge door is opened as few times as possible. Everything is served by 14:00." For each rule give its scope, its strength and the operator (or the reason why no single operator fits).

16. Object fluents

Since Chapter 2 the position of the courier has been a predicate, `(robot-at ?r ?l)`. A predicate is a relation: for every robot and every room it is true or false, and nothing in it says that a robot is in *one* room. That a robot has exactly one position is a fact about the world which the model keeps true by discipline: every action that adds a `robot-at` deletes another one. Chapter 7 gave the pattern a name, the at-most-one invariant, and listed its failure, the forgotten delete, among the anti-patterns. When the discipline fails, the planner does not complain. It uses the robot that is now in two rooms.

PDDL 3.1 offers a way to say it once, in the declaration. An **object fluent** is a function whose value is an object: `(location-of ?r)` is *the* room of robot `?r`. It has one value at a time, an action changes it by assigning a new one, and the old value is gone without anybody deleting it. Chapter 8 introduced functions whose values are numbers; this chapter gives them objects as values, with the same keywords: `:functions`, `=` and `assign`.

The chapter starts with the bug, then rewrites the courier world with two object fluents. It uses fluents as terms inside other expressions, meets the fluent without a value, `undefined`, and visits the blocks world and a map to be colored. It then looks at what planners build from a model, state variables with finite domains, which is what an object fluent states directly. The MODEL method gains a question for step O and a comparison between predicate and fluent. A last section closes Part V with the features of PDDL 3 in one table.

16.1 One robot in two rooms 347
16.2 A function whose value is an object 348
16.3 Fluents as terms 350
16.4 No value: undefined . . 351
16.5 Two more worlds 352
Blocks: one fluent instead of four predicates 352
Coloring a map 353
16.6 State variables: what the planner sees 354
16.7 The MODEL method: one value or many . . . 355
Alternatives compared . 356
16.8 PDDL 3 at a glance . . . 356
16.9 Common mistakes . . . 357
16.10 Summary 358
16.11 Further reading 358
16.12 Exercises 359

16.1. One robot in two rooms

Take the typed domain of Chapter 4 and a round trip: bring the medicines from the pharmacy to ward A and come back to the dock (`courier-p02.pddl`). The plan has ten actions: seven for the delivery and three moves to return.

Now a slip of the kind every modeler makes. In `courier-twin-domain.pddl` the action `move` has lost one line:

```
(: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` finds nothing wrong, because nothing is wrong with the PDDL. The planner finds a 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)
```

Six actions for a round trip that needs ten. Read the fourth: the robot picks the medicines in the pharmacy and leaves *from the lobby*. It never walked back, because it never left: after two moves `(robot-at r1 dock)`, `(robot-at r1 lobby)` and `(robot-at r1 pharmacy)` are all true. And the plan has no return trip at all, since the robot is still at the dock (Figure 16.1). The validator accepts the plan: it checks the plan against the model, and the model is the problem.

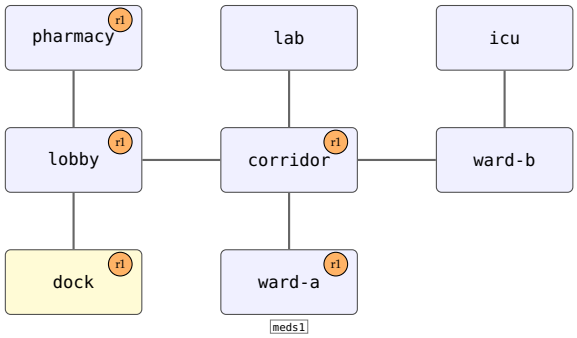


Figure 16.1.: The last state of the six-action plan in the domain with the forgotten delete. The robot is in five rooms at once. Nothing in a predicate says that this state is impossible.

The cheat test of step E catches this bug if somebody reads the plan. But the cause is deeper than a missing line. The predicate `robot-at` can describe 2^8 situations for one robot on this floor, any subset of the eight rooms, and only 8 of them make sense. The model relies on its actions to stay inside those 8. A representation in which the other 248 cannot even be written would not need the discipline.

A predicate lets a robot be in any number of rooms. That it is in exactly one is an invariant which only the effects of the actions keep, and one forgotten delete breaks it silently.

16.2. A function whose value is an object

An **object fluent** is declared in `:functions`, like the numeric fluents of Chapter 8, with one addition: after the function comes the type of its value.

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

`(location-of ?r)` takes a robot and gives a location. For each robot it has, in every state, at most one value. The requirement is `:object-fluents`. Three constructs use a fluent, and all three are already known:

- **Initial value**, in `:init`: `(= (location-of r1) dock)`.
- **Test**, in a precondition or a goal: `(= (location-of ?r) ?from)` is true when the value of the fluent is the object `?from`. This is the equality of Chapter 4, so the domain also declares `:equality`.
- **Change**, in an effect: `(assign (location-of ?r) ?to)` gives the fluent a new value. The old one is replaced. There is nothing to delete.

Listing 16.1 is the courier world rewritten. The position of a robot is `(location-of ?r)`. For items the typed domain had two predicates, `item-at` and `carrying`, with a second hidden invariant: an item is in one room or in one robot's gripper, never both, never neither. A single fluent says it, `(holder ?i)`: what holds the item. Its value can be a room or a robot, so the domain gives the two types a common supertype, `place` (Chapter 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))))
```

Listing 16.1: The courier world with object fluents. `robot-at`, `item-at` and `carrying` are gone.

Compare each action with its version of Chapter 4. `move` had two effects, a delete and an add; now it has one assignment, and there is no line to forget. `pick` had three effects on the item and the robot; now the item changes holder, from the room to the robot. `(free ?r)` is still a predicate: it is a property that is true or false, not a value (Exercise 16.2.2 removes it).

The problem `courier-of-p01.pddl` states the initial values and tests one in the goal:

```
    (= (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)
```

The plan is the one of Chapter 4, action for action: the world did not change, its description did. The round trip (`courier-of-p02.pddl`) takes ten actions, as it should. And the bug of the previous section cannot be written: there is no way to assign a second value to `(location-of r1)` and keep the first (Figure 16.2).

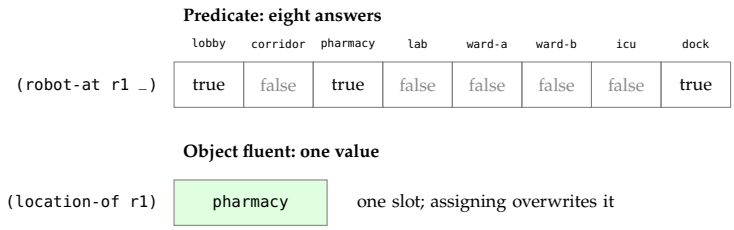


Figure 16.2.: The same question, "where is `r1`?", asked of a predicate and of a fluent. The predicate answers once per room, and three answers can be `true`, as in the state that the buggy domain reached. The fluent has one slot.

An object fluent is a function with an object as value: declared with its type, given a value in the initial state, tested with =, changed with assign. One value at a time, by construction.

Functions are not predicates

A fluent never appears alone as a condition, and it does not take its value as an argument. The habit of years of `robot-at` produces `(location-of r1 dock)`. In the initial state of `courier-of-oldstyle.pddl` it gives only a warning:

```
$ 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
...
```

The fact is ignored, the robot starts nowhere, and the planner answers `unsolvable`. *Fix*: a fluent is always compared or assigned: `(= (location-of r1) dock)`.

16.3. Fluents as terms

In `(= (location-of ?r) ?from)` the parameter `?from` has one job: to carry the position of the robot from the precondition to the rest of the action. With a predicate there was no other way to name "the room where `?r` is". With a fluent there is: the expression `(location-of ?r)` *is* that room, and it can be written wherever an object can, as an argument of a predicate or on either side of an equality.

The domain `courier-terms` uses it in two actions:

```
; 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` reads: the room of the robot is connected to `?to`. `pick` reads: the holder of the item is the room of the robot. Both sentences are closer to how the action would be described in words than their versions with an extra parameter. `drop` keeps its parameter `?l`, which names the room that becomes the holder.

```
$ 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)
```

The plan is the same and its lines are shorter: an action is named by its parameters, and these actions have fewer. That is a gain for the planner too. Chapter 3 showed that parameters multiply ground actions. `pick` with `?l` had one ground action per robot, item and room; without it, one per robot and item. For this problem Kairos grounds 30 actions from `courier-of` and 23 from `courier-terms`, and the difference grows with the number of rooms and items.

A fluent can be written where an object is expected. Parameters whose only job was to name a value disappear, and with them ground actions.

The price is paid by whoever reads the plan: `(move r1 lobby)` no longer says where from. For a robot that executes the plan that is harmless, since it knows where it is. For a person who debugs it, the redundant parameter was documentation.

16.4. No value: undefined

A fluent that is not given a value in `:init` has none: it is **undefined**. An effect can also take the value away, with the keyword `undefined` in an assignment. Both are legitimate tools: some properties have a value only part of the time.

The domain `courier-cargo` keeps `carrying` as a predicate and makes the room of an item a fluent, `(room-of ?i)`. While a robot carries the item, the item is in no room:

```
  (: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` assigns the room again. The problem `courier-cargo-p01.pddl`, two deliveries, is solved in 11 actions as expected.

An undefined fluent is equal to nothing. A test such as `(= (room-of ?i) ?l)` cannot hold while the item is carried, whatever `?l` is, so nobody can pick an item out of another robot's gripper. The same applies at the end of the plan. The plan `courier-cargo-p01-carried.plan` stops one action early, with the sample still in the gripper, in the lab:

```
$ 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 is not a value to compare with

`undefined` can be assigned, but it cannot be tested: there is no `(= (room-of ?i) undefined)`. The domain `courier-cargo-undef` tries it in a precondition:

```
$ 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
```

Fix: when the model must *ask* whether a property has a value, do not use `undefined` for the missing one. Keep a fact that says it, as `(carrying ?r ?i)` does here, or give the fluent a value for that case, as `holder` does with the robot.

That is the design choice between the two courier models of this chapter. In `courier-of` the fluent `holder` is *total*: every item always has a holder, a room or a robot, and no test ever meets an undefined value. In `courier-cargo` the fluent `room-of` is *partial*, and a second piece of state, `carrying`, covers the gap; two pieces of state that must agree are an invariant again. Prefer total fluents. Use `undefined` when "no value" is a real situation that nothing needs to ask about: a region that has no color yet, in the next section.

A fluent without a value is undefined. It can be assigned undefined but never compared with it. A fluent that always has a value is easier to keep right.

16.5. Two more worlds

Blocks: one fluent instead of four predicates

The blocks world of Chapter 2 had `on`, `on-table`, `clear` and `holding`, and most of the effects of its four actions existed to keep them in step. But the whole state of a table of blocks is one fact per block: what it rests on. Listing 16.2 models exactly that, `(on-top-of ?b)`, whose value is another block or the table. The table is a constant of the domain, of a type `surface`, and `block` and `surface` share the supertype `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)))
```

Listing 16.2: Blocks world with a single object fluent. "Clear" is not stored: it is read from the fluent with an existential condition.

A block is clear when no block has it as its support: `(not (exists (?c - block) (= (on-top-of ?c) ?b)))`. With that the domain needs two actions and each has a single effect. Reversing a tower of three (`blocks-of-p01.pddl`):

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

There is no state that this model can reach in which a block is on two things, or clear and covered at once: there is nothing to go out of step. The price is in the preconditions, which now quantify over all blocks, and in granularity: the hand that held a block between two actions is gone (Chapter 7 discussed that choice).

Coloring a map

Give each region of a map a color so that no two regions with a common border share one. A region has exactly one color, or none yet: an object fluent that starts undefined (Listing 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)))
```

Listing 16.3: Map coloring. The problem gives `color-of` no initial value: every region starts without a color.

The precondition of `paint` looks at all the neighbors: none of them has the color `?c`. A neighbor without a color passes the test, because an undefined fluent is not equal to `?c`. The goal of `coloring-p01.pddl` asks that every region has *some* color, which is how a condition says "is defined" without the forbidden comparison:

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

The map has four regions in a ring around a fifth (Figure 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)
```

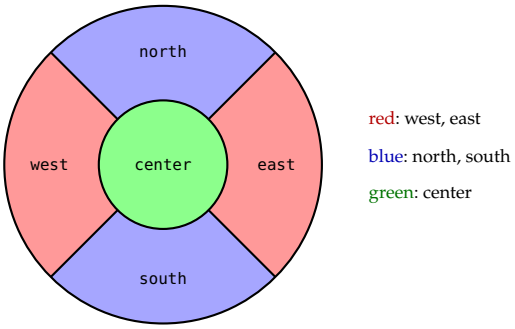


Figure 16.3.: The map of `coloring-p01` with the colors of the plan. Opposite regions of the ring do not touch and can share a color; the center touches all four and needs the third.

With two colors (`coloring-p02.pddl`) Kairos searches the 94 reachable states and answers `unsolvable`, which here is a small proof: this map is not two-colorable. Map coloring is a constraint satisfaction problem, and a planner is not the best tool for it. As a model, though, it shows the fit: one variable per region, one value per variable, a condition between neighbors.

16.6. State variables: what the planner sees

Chapter 3 described a state as a set of facts. That is the view of the language. Most modern planners work with another one: a state is an assignment of values to a fixed list of **state variables**, each with a finite domain of values. "The position of `r1`" is a variable with eight values; "the holder of `meds1`" is a variable with nine (Figure 16.4).

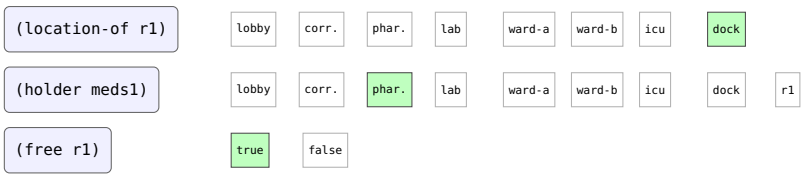


Figure 16.4.: The initial state of `courier-of-p01` as three state variables. Each row is a variable with its domain; the value in green is the current one. A state is one choice per row: $8 \times 9 \times 2 = 144$ combinations, against 2^{18} sets of facts in the predicate model.

A planner that receives predicates has to *discover* the variables. It analyzes the actions, proves that certain facts can never be true together (a **mutex group**: the eight facts `(robot-at r1 _)`), and merges each group into one variable. `kairos -v` reports the result in its last line. For the predicate model of Chapter 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
```

Eighteen facts, in which the planner found two mutex groups (the position of the robot; the whereabouts of the item) and ended with 3 variables. For the model with object fluents:

```
$ 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
```

The same 3 variables, the same 30 actions, the same search, state for state. But no mutex group had to be found: the two fluents *were* the variables, and the only fact left is `(free r1)`. And the domain with the forgotten delete? The same line of `kairos -v` for `courier-twin-p02` says `10 variables (1 mutex groups)`. The analysis could no longer prove that the robot is in one room, because it is not true, and each `robot-at` fact became a variable of its own. An unexpected number in that line is a symptom worth knowing: a model whose invariants the planner cannot find is often a model whose invariants do not hold.

Planners turn sets of facts into state variables by proving which facts exclude each other. An object fluent hands them the variable, and the invariant with it.

Finite-domain representations

The representation with multi-valued variables is known in the planning literature as SAS$^+$ or *finite-domain representation*. The Fast Downward planner (Helmert, 2006) made it the standard internal form: a translator turns PDDL into finite-domain variables before any search, and the invariant analysis behind that translation is described in detail by Helmert (2009). Many heuristics are defined on these variables and are more accurate the fewer and the larger they are: a planner that knows that a robot has one position does not waste effort on states where it has two.

PDDL 3.1

Object fluents arrived with PDDL 3.1, the language of the sixth International Planning Competition, in 2008. The same revision split the old requirement `:fluents` in two, `:numeric-fluents` and `:object-fluents`, and added the `:action-costs` of Chapter 9. The complete grammar of PDDL 3.1 was written down by Daniel Kovacs (Kovacs, 2011), and is the reference for the syntax used in this chapter.

16.7. The MODEL method: one value or many

New questions for step O (Objects and state)

For every dynamic property in the state table, ask how many values it has for one object:

- **Exactly one, always?** The room of a robot, the holder of an item, the state of a door. It is a function. Write it as an object fluent, or as a predicate with an invariant that you state in the table and test.
- **At most one?** The color of a region, the item in a gripper. A function that may be undefined. Decide who needs to ask whether it has a value.
- **Any number?** The rooms a robot has visited, the items on a cart, the rooms connected to a room. It is a relation: a predicate,

and nothing else will do.
- **A number?** A numeric fluent (Chapter 8).

The state table gains a column, *values per object*: one, at most one, many. Every "one" written as a predicate is an invariant, and step E owes it a test.

Alternatives compared

Three encodings of where things are in the courier world, all in the examples of the chapter and all run on the same delivery: the **predicates** of Chapter 4 (`courier-domain.pddl`), the **object fluents** with every parameter kept (`courier-of`), and the fluents used **as terms** (`courier-terms`).

	Predicates	Object fluents	Fluents as terms
Fidelity	Right only while every action keeps the invariant; a forgotten delete gives impossible plans	The invariant cannot be broken	The same
Size	18 facts, 30 ground actions	1 fact, 2 fluents, 30 ground actions	23 ground actions; fewer parameters
Guidance	3 variables, after the planner finds 2 mutex groups	3 variables, given	The same
Clarity	Familiar; the invariant is implicit	The declaration documents the invariant	Shortest; plans say less `((move r1 lobby))`
Portability	Every planner	`:object-fluents`: of the planners compared with Kairos, only Kairos	The same

Table 16.1.: Predicates against object fluents for the position of the robot and of the items. The numbers are those of `kairos -v` for the delivery of `p01`.

Verdict. On every criterion but the last, the fluent is equal or better, and on Fidelity it is better in the way that matters most: a whole class of bugs disappears. The last criterion decides in practice. The comparison of planners in the documentation of Kairos (Martín Rico et al., 2026) finds no other planner among those tested that accepts object fluents, and a model that must run elsewhere has to use predicates. So: think in fluents always, since the question "one value or many" belongs to step O whatever the syntax. Write fluents when the planner is Kairos or another that supports PDDL 3.1 fully. Write predicates otherwise, and then record each single-valued property as an invariant and test it; the count of variables and mutex groups of `kairos -v` is one cheap test.

16.8. PDDL 3 at a glance

This chapter closes Part V. PDDL 3.0 added to the language two ways of talking about the quality and the shape of a plan, and PDDL 3.1 added the fluent of this chapter and tidied the requirements. Table 16.2 collects them.

Feature	Requirement	Version	Where
Soft goals, preferences in preconditions, `is-violated`	`:preferences`	3.0	Chapter 14
Trajectory constraints, in the domain and in the problem	`:constraints`	3.0	Chapter 15
Soft constraints (a preference around a constraint)	both of the above	3.0	Chapter 15
Functions with objects as values, `undefined`	`:object-fluents`	3.1	Chapter 16
Functions with numbers as values (the old `:fluents`)	`:numeric-fluents`	3.1	Chapter 8
Numeric and object fluents together	`:fluents`	3.1	
`total-cost` increased by actions	`:action-costs`	3.1	Chapter 9

Table 16.2.: The features of PDDL 3.0 and 3.1, their requirements and where the book treats them.

The three chapters of this part share an idea. Preferences, constraints and object fluents add no new kind of action and no new kind of world: everything they describe could be, and for years was, encoded with the tools of Parts II and III, as costs, as bookkeeping predicates, as invariants kept by hand. What they add is a place to say directly what the modeler means: this is wished, this is a rule, this has one value. A model that says what it means is shorter, and its mistakes are easier to see. The cost is that fewer planners read it, and the Portability criterion has been the last word of each comparison.

With Part VI the language grows in the other direction again. PDDL+ brings a new kind of world: one that changes by itself.

16.9. Common mistakes

A value of the wrong type

A fluent has a type, and `kairos check` enforces it in assignments, initial values and comparisons. `courier-of-badtype-domain.pddl` assigns an item where a location is expected:

```
$ 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
```

This is the fluent working for the modeler: the same slip with predicates, `(robot-at ?r ?i)`, is caught only if the predicate is typed too. *Fix*: read the message; when a fluent must take values of two types, as `holder` does, give them a common supertype.

Two initial values

A fluent has one value, so `:init` may give it only one. `courier-of-twice.pddl` puts the robot at the dock and in the lobby:

```
$ 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
...
```

A warning, and a plan that starts from the lobby. With predicates the same file would have put the robot in two rooms without a word. *Fix*: read the warnings; problem files written by a program are where this one appears.

A fluent that nobody initialized

Leaving `(= (location-of r1) dock)` out of `:init` is not an error: the fluent is undefined, every precondition that tests it is false, and the planner answers `unsolvable` after searching one state. Nothing points at the missing line. *Fix*: when a problem with fluents is unsolvable at once, check first that every fluent of every object has its initial value, unless it is meant to start undefined.

A fluent for something that is a relation

After this chapter everything looks like a function. "The room connected to the lobby" is not one: the lobby has three neighbors. "The item a robot carries" is one only while the robot has no basket. A property forced into a fluent loses the values that do not fit, and the model loses solutions. *Fix*: the question of step O, asked honestly: exactly one, at most one, or many. Only the first two are fluents.

16.10. Summary

An object fluent is a function from objects to an object, with at most one value in every state. It is declared in `:functions` with the type of its value, initialized and tested with `=`, and changed with `assign`; the requirement is `:object-fluents`.

- A fluent replaces a predicate and the invariant "one value per object" that the predicate needed. There is no delete to forget.
- A fluent is a term: it can be an argument or one side of an equality, and parameters that only carried its value can be dropped.
- A fluent may be undefined, initially or by assignment. `undefined` cannot be tested; prefer fluents that always have a value.
- Planners build state variables from predicates by finding mutex groups. A fluent gives them the variable.
- Step O asks of every property: one value, at most one, or many. Few planners accept object fluents, so the answer is often written as a predicate with a tested invariant.

16.11. Further reading

The grammar of object fluents, with `assign` and `undefined`, is in the BNF of PDDL 3.1 by Kovacs (2011). The textbook of Haslum et al. (2019)

discusses object fluents as part of its account of PDDL's state variables. For what planners do internally, the description of Fast Downward (Helmert, 2006) explains the finite-domain representation and why heuristics benefit from it, and Helmert (2009) gives the algorithm that finds the invariants in a model written with predicates. The state-variable view of planning is also the one adopted by Ghallab et al. (2016) from their first chapter.

16.12. Exercises

Exercises for Section 16.1 — One robot in two rooms

Exercise 16.1.1. The typed courier domain (`courier-domain.pddl`) keeps three invariants with its effects: a robot is in one room; an item is in one room or carried by one robot; a robot is free exactly when it carries nothing. For each one, name the effects of each action that keep it.

Exercise 16.1.2. In `courier-twin-domain.pddl` the robot multiplies. Break the second invariant instead: remove `(not (item-at ?i ?l))` from `pick` in a copy of the typed domain, and describe, before running anything, a goal for which the planner would return an impossible plan.

Exercises for Section 16.2 — A function whose value is an object

Exercise 16.2.1. In `courier-of` the holder of an item can be a robot, so a goal can ask for it. Write a problem with two robots, `r1` at the dock and `r2` in the lab, and a sample in ward A, whose goal is that `r2` holds the sample and is in the lab. Which robot fetches it?

Exercise 16.2.2. `(free ?r)` repeats what `holder` already knows: a robot is free when it is the holder of no item. Write a domain `courier-nofree` without that predicate. Is the new condition safe with respect to undefined values?

Exercises for Section 16.3 — Fluents as terms

Exercise 16.3.1. For one robot, eight rooms with their 14 connections and one item, count by hand the ground actions of `courier-of` that survive the static precondition `connected`, action by action, and compare with the 30 that Kairos reports. Then explain where the 23 of `courier-terms` come from.

Exercise 16.3.2. Two robots must never be in the same room. Extend `courier-of` so that `move` refuses to enter a room where another robot is, comparing the fluents of the two robots. Test it with `r2` standing in the corridor, in the way of a delivery that `r1` would make, and read the plan.

Exercises for Section 16.4 — No value: undefined

Exercise 16.4.1. In `courier-cargo`, could the plan end with the goal `(= (room-of sample1) lab)` satisfied while the robot still carries `sample1`? Why not? What would be needed to ask, in a goal, that the robot ends carrying nothing?

Exercise 16.4.2. Compare `holder` (total) and `room-of` with `carrying` (partial) with step L: which invariants does each model leave to the actions?

Exercises for Section 16.5 — Two more worlds

Exercise 16.5.1. Write a problem for `blocks-of` with two towers, `b` on `a` and `d` on `c`, whose goal is both towers upside down. How many actions does the best plan have? If Kairos prints a longer plan first, explain why.

Exercise 16.5.2. Color the map of Australia with three colors: Western Australia, Northern Territory, South Australia, Queensland, New South Wales, Victoria and Tasmania. (South Australia borders every mainland region.) What does the planner do with Tasmania?

Exercises for Section 16.6 — State variables: what the planner sees

Exercise 16.6.1. Run `kairos -v` on `courier-p02` and on `courier-twin-p02`, and explain the two counts of variables and mutex groups. Which mutex group survives in the buggy domain?

Exercises for Section 16.7 — The MODEL method: one value or many

Exercise 16.7.1. Classify each property as a predicate, an object fluent, an object fluent that may be undefined, or a numeric fluent: the floor an elevator is on; the people inside an elevator; the state of a door (open, closed, locked); which rooms a door joins; the battery level; the cart a robot is pushing; the items on a cart; the robot that is charging at the only charger.

Exercise 16.7.2. The doors of Chapter 5 had the predicates `(open ?d)` and `(locked ?d)`. Model a door with one fluent `(state-of ?d)` whose values are three constants, with actions to unlock, open and close, and a problem. Which impossible state of the predicate model can no longer be written?

Exercises for Section 16.8 — PDDL 3 at a glance

Exercise 16.8.1. For each need, name the feature of Part V and its requirement: (a) the robot should avoid the lobby, but may cross it; (b) the robot must never be in the lobby with a sample; (c) every item has exactly one holder; (d) three deliveries are asked and the battery allows two; (e) the delivery should happen in the first ten minutes, and it is bad, but not fatal, if it does not.

Part VI. Hybrid systems: PDDL+

17. Processes and events

In every model so far, the world changed only because the agent did something. A fact became true as the effect of an action; a number went up or down while a durative action ran. Even the battery of Chapter 12, which drained continuously, drained only during a `move`, and stopped the moment the move ended. Leave that robot in the corridor, switched on, with nothing to do for an hour, and its battery is exactly as it was: no action is running, so nothing changes. A real robot would be dead. A sample left on a bench warms up whether or not somebody plans to carry it, a tank goes on filling after the agent has turned to something else, and a ball that has been let go falls without asking. The timed initial literals of Chapter 13 let the world change by itself, but only at times written in the problem. Here the changes depend on the state: *while* the robot is on, *when* the tank is full.

This chapter makes the step from "the agent changes the world" to "the world changes by itself". PDDL+ adds two constructs to the language. A *process* is a continuous change that runs for as long as its condition holds, with nobody starting it and nobody able to refuse it. An *event* is an instantaneous change that happens at the moment its condition becomes true, again with no choice for anybody. Actions stay in the language, but their role narrows: they are the only decisions of the agent, and everything else follows from the laws of the world. With the three together we model a tank that fills and overflows, a ball that falls and bounces, a car that the driver steers only through its acceleration, and a courier robot whose battery drains whenever it is on and whose samples spoil if it dawdles.

The chapter starts with the simplest process and the simplest event, on a water tank, and reads its plans, which look different from the ones we know. Then it sets the three kinds of change side by side and follows them through two classic domains, the bouncing ball and the car. The courier world comes next: moving in PDDL+, the battery, and a sample that warms up. A section explains how Kairos plans and validates these models, because what a planner can and cannot find matters more here than anywhere else in the book. The chapter ends with what PDDL+ adds to the MODEL method, the usual mistakes, and exercises.

17.1 A world that does not wait 362
17.2 Processes: change while a condition holds 363
The first plan 364
The plan is not unique, and neither is the time . 365
17.3 Events: change at an instant 366
An event must switch itself off 366
A problem without a plan 367
17.4 Three kinds of change . 367
The bouncing ball 368
The car 370
17.5 The courier world in PDDL+ 372
A move is an action, a process and an event . . 373
The battery: a process and an event 373
Exercising the model . . 374
The cheat test: a robot that coasts 375
A sample that warms up 376
17.6 How Kairos plans PDDL+ 378
Search in steps of time . 378
Validation in continuous time 378
What it can miss 379
17.7 The MODEL method: who causes each change 380
Alternatives compared . 380
17.8 Common mistakes . . . 381
17.9 Summary 382
17.10 Further reading 383
17.11 Exercises 383

17.1. A world that does not wait

Take a water tank with a valve. The agent opens the valve, water comes in at two liters per second, and the agent wants eight liters. With the tools of Part IV we would write a durative action `fill` whose duration the planner chooses, with a continuous effect on the level. That model is correct as long as the story is about filling. It stops being correct as soon as we ask what happens if the agent forgets the valve: in the model, nothing, because a durative action always ends, and when it ends the water stops. The model has no way to say "the water keeps coming until somebody closes the valve", and so it cannot say "and then the tank overflows" either.

The trouble is not the continuous effect. It is *who owns the change*. In a durative action, the change belongs to the agent: it starts when the agent starts the action and it cannot outlive the action. In the world, the change belongs to the tank: water flows while the valve is open, and the valve is a fact of the state, not an activity of the agent. The agent decides two things, when to open and when to close. Between those two decisions the world runs on its own.

In PDDL+ the agent only decides. Change that goes on by itself is a process; change that happens by itself at an instant is an event.

A system with both discrete state (the valve is open or closed) and continuous state (the level) that evolves in time is called a **hybrid system**. **PDDL+** is the extension of PDDL for hybrid systems, defined by Maria Fox and Derek Long Fox and Long, 2006. It keeps everything of PDDL 2.1 and adds two kinds of things that can happen, next to actions:

- A **process** changes numbers continuously, for as long as its condition holds. It is not chosen: whenever the condition is true, the process is running.
- An **event** changes the state at one instant, the first instant in which its condition is true. It is not chosen either: when the condition becomes true, the event fires.

Processes and events are the laws of the world. Actions are what is left for the agent, and in a pure PDDL+ model they are instantaneous: opening a valve, releasing a ball, pressing a pedal. A plan is the list of those decisions with their times. What the world does in between is not in the plan, because it is not a decision; it follows from the model.

From PDDL 2.1 to PDDL+

PDDL 2.1 Fox and Long, 2003 was designed in levels, and its authors, Maria Fox and Derek Long, already had in mind a last level in which change could be started by the world and not only by the agent. That level was not used in the 2002 competition. Fox and Long published it in full in 2006 in the *Journal of Artificial Intelligence Research*, under the title "Modelling Mixed Discrete-Continuous Domains for Planning" Fox and Long, 2006. The paper gives PDDL+ a formal meaning by translating every domain into a *hybrid automaton*, the model that control engineers and computer scientists use to describe and verify systems such as thermostats, engines and flight controllers. A PDDL+ plan is, in those terms, a way of driving the automaton to a goal.

17.2. Processes: change while a condition holds

Listing 17.1 is the tank. Read it in three blocks: what the agent decides (two actions), what the world does while a condition holds (one process), and what the world does at an instant (one event, which the next section explains).

```
;; 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)))
```

Listing 17.1: The tank: two actions, one process and one event (`examples/ch17/tank-domain.pddl`).

A process is written like an action, with the keyword `:process`:

```
(:process fill
  :parameters ()
  :precondition (valve-open)
  :effect (increase (level) (* #t (inflow))))
```

The differences with an action are in the meaning, not in the syntax.

- The `:precondition` is not something the planner must achieve in order to use the process. It is the condition under which the process *is running*. While `(valve-open)` is true, `fill` runs; when it is false, `fill` does not.
- The `:effect` can only be continuous: `increase` or `decrease` of a numeric fluent by an expression multiplied by `#t`, as in the continuous effects of Chapter 12. Here `#t` is the time that the process has been running, and `(inflow)` is the **rate**: liters per second. A process cannot add or delete a fact, and it cannot `assign`.
- A process never appears in a plan. Nobody applies it.

The requirement for processes and events is `:time`[1], together with `:numeric-fluents`, because a process has to change some number. The domain also uses `:negative-preconditions`, for `(not (valve-open))`.

1: Some PDDL+ files in the wild declare `:processes` and `:events` or nothing at all. Kairos accepts a domain with processes and no `:time`, with a warning (Section 17.8).

Notice how the agent controls the process: it does not. It controls the fact `(valve-open)`, with two ordinary actions, and the fact switches the process on and off. This indirect control is the pattern of every PDDL+ model: *actions change facts, facts enable processes, processes change numbers*.

A process is an action that nobody chooses: its precondition says when it runs, and its effect, always with #t, says how fast it changes a number.

The first plan

The first problem asks for at least eight liters, with the valve closed at the end and no flood.

```
;; 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)))))
```

Listing 17.2: Eight liters, valve closed, no flood (`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]
```

The plan has the format of the temporal plans of Chapter 10: a time, an action, and a duration in brackets, which is zero because both actions are instantaneous. Three things are new.

First, the gap. Nothing is written between 0 and 4 seconds, and yet that is where the plan does its work: the valve is open, so `fill` runs, and the level goes from 0 to $2 \times 4 = 8$ liters. The plan says *when* the agent acts; the model says what happens meanwhile. To read a PDDL+ plan we must replay the processes in our head, or on paper: Figure 17.1 does it for this plan.

Second, the line `@PlanEND`. A PDDL+ plan may need to let time pass after its last action, until the world reaches the goal on its own, so the plan states where it ends. Here the end coincides with the last action.

Third, the note. Kairos looks for plans with few actions, and reports as *cost* the duration of the plan. It does not minimize that duration: we will see plans that are valid and longer than needed. The note is printed with every PDDL+ plan and we leave it out of the listings from here on.[2]

2: The outputs of this chapter and the next were obtained with the options of the test script of the book, in particular `-threads 1`. With more threads Kairos runs several searches at once and may print other plans, as valid as these.

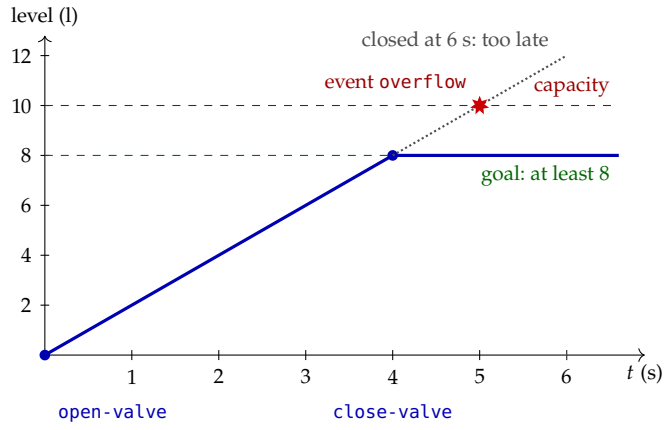


Figure 17.1.: The level of the tank over time. Thick line: the plan of Kairos. The slope between the two actions is the rate of the process `fill`. Dotted: the same plan with the valve closed two seconds later; on its way the level crosses the capacity and the event fires.

The plan is not unique, and neither is the time

Closing the valve at 4 seconds is the earliest moment that reaches the goal, but not the only one. Any time from 4 seconds up to just before 5 will do. We can check it with a plan written by hand: the validator of Kairos replays any plan in continuous time.

```
$ 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
```

In classical planning a plan was a choice of *which* actions. With durative actions it became also a choice of *order and overlap*. In PDDL+ the times themselves are the plan: the same two actions solve the problem at 4.5 seconds and fail at 3 or at 6. The planner searches over instants, which is why PDDL+ planning is expensive and why this chapter keeps its problems small.

17.3. Events: change at an instant

The second law of the tank is the flood.

```
(:event overflow
  :parameters ()
  :precondition (and (>= (level) (capacity)) (not (flooded)))
  :effect (flooded))
```

An event has the syntax of an action and the effects of an action: it may add and delete facts, and `assign`, `increase` or `decrease` numbers by a fixed amount. What changes is, again, who decides. An event **fires** at the first instant in which its precondition is true. The planner cannot apply it earlier, cannot delay it and cannot skip it.

An event is an action that nobody chooses and nobody can avoid: it fires at the instant its condition becomes true. The agent can only keep that condition from becoming true.

Events are how a PDDL+ model says that something goes wrong, or simply that something happens. The planner avoids a bad event in the only way it can: by acting so that its condition never holds. In the plan of the previous section the valve is closed at 4 seconds, the level stays at 8, and `overflow` never fires. Close it too late, and it does:

```
$ 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
```

Both actions of that plan were applicable. Nothing in it is "illegal". It fails because, at 5 seconds, the level reached the capacity and the world added `(flooded)`, which the goal forbids (the dotted line of Figure 17.1). This is the usual way of excluding a bad outcome with an event: the event leaves a mark, and the goal (or the precondition of the actions that matter) refuses the mark. Chapter 18 shows another way, the global constraint.

An event must switch itself off

Look at the second conjunct of the precondition, `(not (flooded))`. Without it the event would be true at 5 seconds, fire, and still be true: the level is still at the capacity. It would have to fire again, and again, infinitely

many times in the same instant. PDDL+ forbids this: **an event must make its own precondition false**. The simplest way is the one used here, a fact that the event adds and that its precondition negates. The event then fires once.

An event may also be the goal of the agent. Problem `tank-p02` asks for the flood:

```
$ kairos tank-domain.pddl tank-p02.pddl
; cost = 5
; makespan = 5.000
0.000: (open-valve) [0.000]
5.000: @PlanEND
```

One action, and then five seconds in which the agent does nothing. The plan ends at `@PlanEND`, when the event has fired. Here the end of the plan matters: a plan with `open-valve` alone, ending at time zero, would not reach the goal.

A problem without a plan

Problem `tank-p03` asks for 12 liters in a tank of 10 without a flood. No timing of the valve can do that.

```
$ 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)
```

The answer is not the `unsolvable` of the classical chapters, and the exit code is 14, not 10. Kairos says *no plan found*: it tried every way of timing the valve that it considers and none worked, but it does not consider every instant, so it does not claim a proof. Section 17.6 explains what exactly it tries. For the test set of step E this means that "must have no plan" is checked, in PDDL+, as "Kairos finds none", and that we should also convince ourselves by reasoning, as we just did.

17.4. Three kinds of change

Table 17.1 puts the three constructs side by side. The syntax is almost the same; the rows that differ are the ones about choice.

	Action	**Process**	**Event**
Who causes it	the agent	the world	the world
When	when the planner decides, if the precondition holds	all the time the precondition holds	at the first instant the precondition holds
How long	an instant	as long as the condition lasts	an instant
Effects	facts and numbers, in jumps	numbers only, continuously (`#t`)	facts and numbers, in jumps
In the plan	yes, with its time	no	no

Table 17.1.: Actions, processes and events.

The three feed each other in a cycle, drawn in Figure 17.2. An action changes a fact. The fact makes the condition of a process true, and the process starts to move a number. The number reaches a threshold, which makes the condition of an event true, and the event changes facts. Those facts may stop the process, start another one, or enable an action that the agent was waiting for.

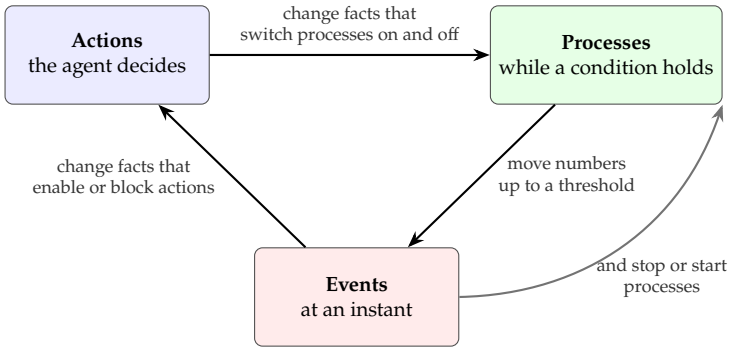


Figure 17.2.: How the three kinds of change feed each other. Only the box on the left is under the control of the planner; the other two are the laws of the world.

What happens between two actions

Between one decision of the agent and the next, a PDDL+ state evolves by itself, and the rules are few. (1) While no event condition is true, time flows: every process whose condition holds changes its numbers at its rate, and the facts stay as they are. (2) The moment some event condition becomes true, time stops, the event fires, and its effects apply. This may make other event conditions true; they fire too, in the same instant, until no event can fire. (3) Then time flows again, with the processes that the new state enables. An action of the plan is applied at its time to the state that this evolution has reached, and its effects may again trigger events before time goes on. If two processes change the same number, their rates add up. If a state would let events fire for ever in one instant, the model is wrong, not the plan.

A useful way to remember the difference with Chapter 12: a durative action with a continuous effect is a process whose start is chosen by the agent and whose end comes at a time fixed when it starts, or chosen. A PDDL+ process starts and ends when the *state* says so. To give the agent the control that a durative action had, PDDL+ models use three pieces: an action that starts the activity (it adds a fact), a process that runs while the fact holds, and either an action or an event that ends it (it deletes the fact). This **start-process-stop** pattern is in the tank (`open-valve`, `fill`, `close-valve`) and will be in every model of the chapter.

The bouncing ball

The bouncing ball is the oldest example of a hybrid system: a continuous law (gravity) interrupted by an instantaneous one (the bounce). In Listing 17.3 the agent holds a ball at five meters, lets it go, and must catch it later with a hand that waits at two meters.

```
;; 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))))
```

Listing 17.3: The bouncing ball (`examples/ch17/ball-domain.pddl`).

Three details deserve a look.

One process, two effects. The process `fall` changes the height at the rate `(velocity)` and the velocity at the rate `(gravity)`, at the same time. The rate of the first effect is not a constant: it is a fluent that the second effect is changing. The height is therefore not a straight line in time but a parabola. The continuous effects of Chapter 12 had constant rates; a rate that is itself changing is something we get almost for free in PDDL+, and Chapter 18 studies it.

An event that assigns. The event `bounce` fires when the ball is on the floor and going down. Its effect turns the velocity around and keeps 80% of it: `(assign (velocity) (* -1 (* (restitution) (velocity))))`. After it, the velocity is positive, so the conjunct `(< (velocity) 0)` is false: the event has switched itself off without any extra fact.

An action with a narrow window. The agent can `catch` only when the ball is within ten centimeters of the hand and slower than 2 m/s. The planner must find an instant in which the world, running by itself, satisfies that.

Problem `ball-p01` drops the ball from 5 m with gravity 10 and asks to have it in the hand again after at least two bounces.

```
$ 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
```

Two actions again, and everything interesting is in between. Let us replay it. From 5 m the ball takes one second to reach the floor ($5 = \frac{1}{2} \cdot 10 \cdot t^2$), and arrives at -10 m/s. The event fires: the velocity becomes +8. The ball rises for 0.8 s to 3.2 m, and is back on the floor 1.6 s after the first bounce, at $t = 2.6$, with -8 m/s. Second event: +6.4 m/s. Half a second later, at $t = 3.1$, the ball is at $6.4 \cdot 0.5 - 5 \cdot 0.5^2 = 1.95$ m and moving at $6.4 - 5 = 1.4$ m/s: inside the window of `catch`. Figure 17.3 shows the whole flight.

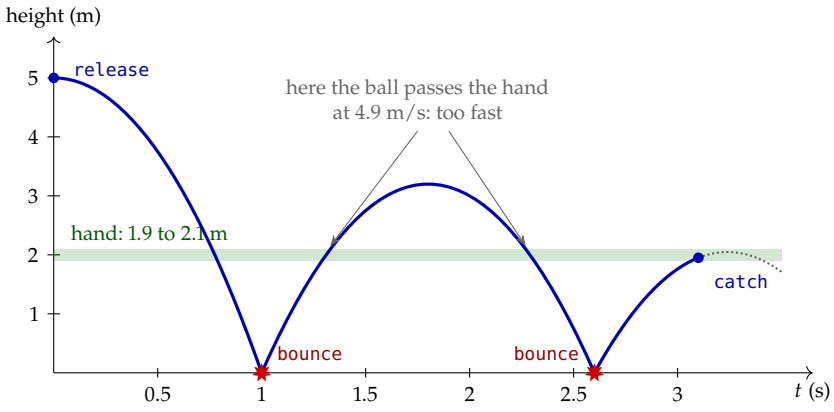


Figure 17.3.: The flight of the ball in the plan of Kairos. Each arc is the process `fall`; the stars are the event `bounce`. The ball crosses the band of the hand five times, and only near the top of the third arc is it slow enough to be caught.

The figure also shows why the plan does not catch the ball earlier. After the first bounce the ball crosses the two meters twice, but at 4.9 m/s. Only the third arc, whose top is at 2.05 m, passes the hand slowly. The planner did not "know" this: it found the one stretch of time in which the precondition of `catch` is true. We could not have told where that stretch is without doing the physics, and that is the point of a PDDL+ model: we write the laws, and the planner finds when to act.

The car

The second classic is the car of Fox and Long Fox and Long, 2006: a car on a straight road whose driver can only change the acceleration. Listing 17.4 is our version of it, with a 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
```

```
16
17  (:action accelerate
18    :parameters ()
19    :precondition (and (running) (< (acceleration) 1))
20    :effect (increase (acceleration) 1))
21
22  (:action brake
23    :parameters ()
24    :precondition (and (running) (> (acceleration) -1))
25    :effect (decrease (acceleration) 1))
26
27  ;; Parking needs the car at rest and the pedals released
28  (:action park
29    :parameters ()
30    :precondition (and (running) (= (acceleration) 0)
31                       (>= (velocity) -0.05) (<= (velocity) 0.05))
32    :effect (and (not (running)) (parked)
33                 (assign (velocity) 0)))
34
35  (:process moving
36    :parameters ()
37    :precondition (running)
38    :effect (and
39      (increase (distance-run) (* #t (velocity)))
40      (increase (velocity) (* #t (acceleration)))))
41
42  (:event radar
43    :parameters ()
44    :precondition (and (> (velocity) (speed-limit))
45                       (not (fined)))
46    :effect (fined)))
```

Listing 17.4: The car: the driver changes the acceleration, a process does the rest (`examples/ch17/car-domain.pddl`).

The actions `accelerate` and `brake` do not move the car. They add or subtract one unit of acceleration, which can be −1, 0 or 1. The process `moving` turns the acceleration into velocity and the velocity into distance, for as long as the engine runs. The event `radar` marks the car as `fined` if it ever goes faster than the limit, and `park` needs the car at rest, give or take 5 cm/s, with no acceleration.

Problem `car-p01` asks to cover at least ten meters and park, with a limit of 2 m/s and no fine. Kairos is an anytime planner: it prints a first plan and then better ones.

```
$ 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
```

The first plan is correct and absurd. The driver accelerates for three tenths of a second, releases the pedal (the `brake` at 0.3 takes the acceleration from 1 back to 0), and then crawls at 0.3 m/s for more than half a minute before braking. It has five actions, the least possible, and that is what the search was looking for. The second plan takes 7.5 seconds: up to 1.8 m/s, three seconds of cruising, and a braking interrupted three times. Every action is applicable and the radar never fires, so it is valid; a driving instructor would not sign it.

Can we do better by hand? The limit suggests the plan: two seconds of acceleration to reach exactly 2 m/s (2 m covered), three seconds at the limit (6 m more), two seconds of braking (the last 2 m). Ten meters in seven seconds:

```
$ kairos validate car-domain.pddl car-p01.pddl car-p01-clean.plan
plan valid, makespan 7
```

Figure 17.4 draws both. And a driver in a hurry, who keeps the pedal down for 3.2 seconds and then brakes? The car covers 10.24 m in 6.4 s, and the validator answers `(fined) is true`: the velocity passed the limit at $t = 2$ and the event fired.

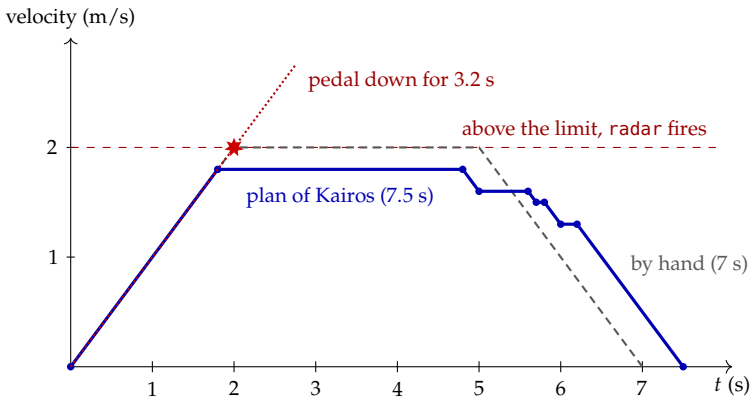


Figure 17.4.: The velocity of the car in three plans. Each dot is an action of the driver; between dots the slope is the acceleration. The area under each line is the distance: 10 m for the two valid plans.

Step D for the car: who causes each change

The story says "the driver drives ten meters". Step D asks what can be done or can happen, and in a hybrid model we must first split the story by cause. *Decided by the agent:* pressing and releasing the pedals, parking. *Done by the world while a condition holds:* the car gains velocity while there is acceleration and covers distance while it has velocity, as long as the engine runs. *Done by the world at an instant:* the radar takes the picture the moment the limit is passed. Each line becomes one construct: three actions, one process, one event. If we had written `drive-ten-meters` as one action we would have given the agent a power it does not have, that of deciding where the car is.

17.5. The courier world in PDDL+

The courier robot has had a battery since Chapter 8, and since Chapter 12 a battery that drains continuously during a move. Now the battery drains *whenever the robot is on*: moving, waiting, picking, or doing nothing at all. That one sentence changes the model more than it seems, because

it makes time matter for every action, and so moving must take time in a way that processes can see.

Only two lines of the model card change. In *Scope*: the battery is spent by being on, not by moving; a robot whose battery is empty switches off where it stands. In *Assumptions*: the drain is constant while the robot is on, and the speed is constant while it moves.

A move is an action, a process and an event

In a pure PDDL+ model there are no durative actions, so a move that takes time is built with the start-process-stop pattern. Listing 17.5 shows the three pieces.

```
;; --- 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)))
```

Listing 17.5: Moving in PDDL+: the agent starts the move, a process advances the robot, an event ends the move (part of the domain `courier-battery`).

- The action `start-move` is the decision. It takes the robot out of `?from`, records that it is `(moving ?r ?from ?to)` and sets the odometer `(traveled ?r)` to zero. While it moves, the robot is in no room: no `robot-at` fact holds, so it cannot pick, drop or start another move.
- The process `travel` advances the odometer at the speed of the robot, while the robot is on its way *and switched on*.
- The event `arrive` fires when the odometer reaches the distance. It puts the robot in `?to` and deletes `moving`, which switches off both the process and the event itself.

The end of the move is an event, not an action, on purpose: a robot that has covered the distance *is* in the next room, whether the agent likes it or not. The section on alternatives compares this with an action `stop-move`.

The battery: a process and an event

The battery needs two laws (Listing 17.6) and two actions, `power-off` and `power-on`, which give the agent the only control it has over the drain. `(powered ?r)` is a precondition of every action of the robot, and `power-on` needs some charge left.

```
;; The battery drains whenever the robot is on
(:process drain
  :parameters (?r - robot)
  :precondition (powered ?r)
  :effect (decrease (battery ?r) (* #t (drain-rate ?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))))
```

Listing 17.6: The battery drains while the robot is on, and an empty battery switches the robot off (part of the domain `courier-battery`).

The process `drain` has nothing to do with moving: its condition is only `(powered ?r)`. The event `shutdown` deletes `(powered ?r)`, which is its own condition, the condition of `drain`, and half of the condition of `travel`. A robot that runs out of battery between two rooms stays there: `moving` holds, `travel` does not run, `arrive` will never fire, and `power-on` is not applicable with an empty battery. The planner cannot let this happen to a robot it needs, and nowhere did we write a precondition that says "have enough battery for the trip". The laws of the world do that job.

Step 0: the state of a hybrid model

The state table grows in two ways. First, facts that exist only to switch processes on and off: `(powered ?r)` and `(moving ?r ?from ?to)` are not things the story mentions as goals; they are the conditions of `drain` and `travel`. Second, numbers that measure a process: `(traveled ?r)` is there so that an event can tell when the move is over. For every process ask: which fact starts and stops it, and which number tells how far it has gone? If either is missing from the table, the process cannot be written.

Exercising the model

The typical problem is the delivery of the first chapters: `r1` is at the dock, the medicines are in the pharmacy, and they must go to ward A. The battery is at 30 and drains at 0.5 per second, so the robot can be on for 60 seconds. The trip is 38 m at 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
```

The plan lists only the starts of the moves. The arrivals are events: we read them off the distances. The robot leaves the dock at 0 and the lobby at 4, so it arrived at the lobby at 4 (4 m at 1 m/s); it leaves the lobby again at 16, six seconds after leaving the pharmacy. Each `start-move` comes at the very instant of the previous arrival. At the end the battery is at $30 - 0.5 \times 38 = 11$. Figure 17.5 shows the first ten seconds in three lanes: what the agent does, what the processes do, and when the events fire.

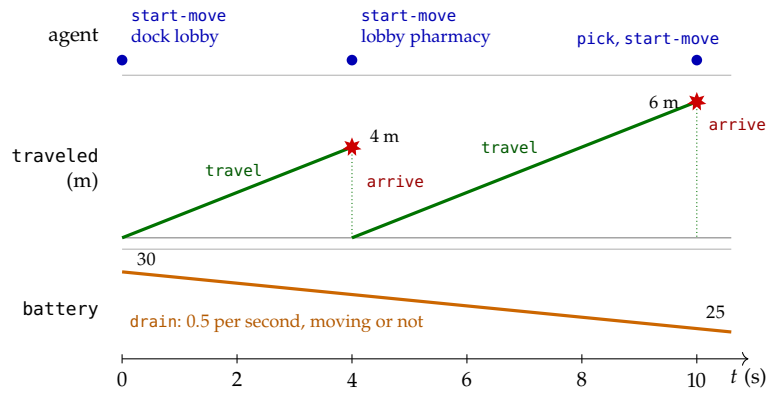


Figure 17.5.: The first ten seconds of the plan. Top: the decisions of the agent. Middle: the process `travel` raises the odometer until the event `arrive` fires, and the next `start-move` resets it. Bottom: the process `drain` does not care about any of this.

Problem `courier-battery-p02` is the same delivery with the battery at 8: enough for 16 seconds. Kairos answers `no plan found`, after trying 221 664 states. That is the problem "that must have no plan" of the test set. The fourth problem of the set, the one that tempts the planner to cheat, deserves its own subsection.

The cheat test: a robot that coasts

Suppose that we had written the process `travel` with the condition `(moving ?r ?from ?to)` alone, forgetting `(powered ?r)`. The model looks fine and solves `p01` with the same plan. The domain `courier-battery-coast` has that bug, and here is what Kairos does with it on the problem that should have no 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
```

The robot starts a move and switches itself off. Off, its battery does not drain, and the wrong process keeps pushing it along the corridor. It switches on for an instant at each room to start the next move. The battery, 8 at the start, is at 0.3 when the robot first switches off, and that is enough for the rest of the day. The planner has found a perpetual motion machine, because the model contained one.

With the correct domain the same plan is rejected at the first step that depends on the free ride: the robot stopped 60 cm short of the lobby when it was switched off.

```
$ 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
```

In a hybrid model the planner looks for free rides: a process that runs when it should not, or one that fails to run when it should. Test with a problem whose only plans would need one.

The lesson generalizes. In classical models the cheat was a missing precondition. In hybrid models it is a missing *condition of a process*: any stretch of time in which a good process runs for free, or a bad process is switched off, will be found and used. The cheat test for a PDDL+ model is a problem that is impossible in the world and would be possible only through such a stretch.

A sample that warms up

The second law of nature in the hospital is about samples. A blood sample must be kept cold. In a fridge it is safe; out of a fridge it warms up, and above a limit it is spoiled. The domain `courier-cold` leaves the battery out, to keep it small, and keeps the same three-piece move. Listing 17.7 shows what is new.

```
  ;; Picking an item takes it out of the fridge, if it was in one
  (: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) (not (chilled ?i))))
  ;; Dropping a sample into the fridge of the room
  (:action store
    :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)))
```

Listing 17.7: The cold chain: picking takes a sample out of the fridge, storing puts it in one, and in between it warms up (part of the domain `courier-cold`).

The process `warm-up` has no fact that an action "starts". Its condition is about where the sample is: it runs for every sample that is not `(chilled ?i)`. No action of the robot mentions the temperature at all. The agent controls the process only through the facts in its condition: `pick` takes a sample out of the cold, and `store` (a `drop` into the fridge of a room that has one) puts it back.

In problem `courier-cold-p01` the robot is in the corridor, with medicines at hand that must go to ward A. A sample waits on a bench of the lab at 4 degrees, already warming up at 0.1 degrees per second, and must end in the fridge of the ICU. It is spoiled at 8 degrees: it has 40 seconds out of a fridge. Both the lab and the ICU have fridges.

A planner that counts actions is tempted to deliver the medicines first, since they are right there: ten actions in all. We write that plan by hand and validate it.

```
$ 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
```

Every action in it is applicable; it fails because of something no action did. The robot reaches the lab at 32 seconds and the ICU at 57, and the sample was spoiled at 40. Now the plan of 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
```

The planner has found something we had not thought of when writing the problem. The robot goes to the lab and, at 8 seconds, picks the sample from the bench and stores it in the fridge *of the lab*. That stops the process. With the sample safe at 4.8 degrees, the robot delivers the medicines without hurry, comes back at 48 seconds, and only then takes the sample out for the 25 seconds of the trip to the ICU, where it arrives at 7.3 degrees. It is what a sensible nurse would do. The plan is not the shortest in time: taking the sample straight to the ICU and then delivering the medicines needs 62 seconds and eleven actions, and the validator accepts it too (`courier-cold-p01-sample-first.plan`). Figure 17.6 draws the temperature in the three plans.

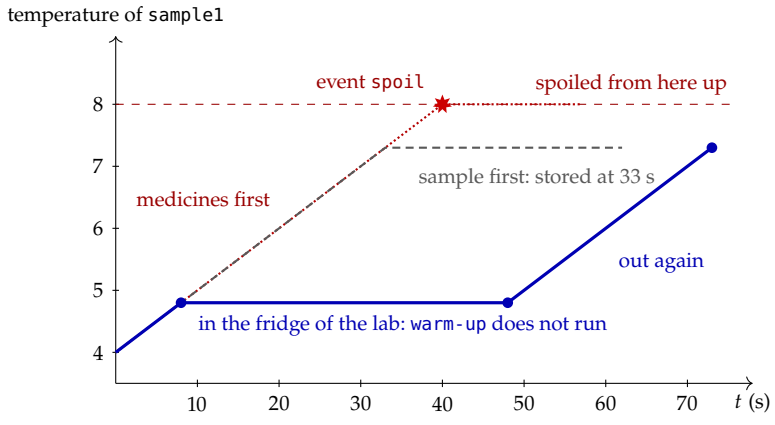


Figure 17.6.: The temperature of the sample in three plans. Dotted: the medicines first; the sample spoils on the bench at 40 s. Dashed: the sample first, by hand. Thick: the plan of Kairos, which parks the sample in the fridge of the lab. The slope is always the same, 0.1 degrees per second: the plans differ only in *when* the process runs.

Problem `courier-cold-p02` is the one without a plan: only the sample, on a hot day in which it lasts 10 seconds on the bench, with the robot 8 seconds away from the lab and the lab 33 seconds from the ICU. Kairos answers `no plan found`.

17.6. How Kairos plans PDDL+

A classical planner chooses among a finite set of actions. A PDDL+ planner must also choose *when*, and time is a continuum. Every PDDL+ planner tames the continuum in some way, and the way determines which plans it can find. This section describes what Kairos does; the facts come from its documentation Martín Rico et al., 2026 and from the runs of this chapter.

Search in steps of time

Kairos plans PDDL+ **in steps of time**. To the actions of the domain it adds actions of its own that do nothing but let time pass:

- a *short wait*, which lets 0.1 seconds pass;
- a *long wait*, which lets time pass, in steps of 0.1 seconds, until some numeric condition is about to change: it stops at the last step before the change.

When time passes, the processes that are active are integrated numerically with a Runge-Kutta method, so that a quantity whose rate is changing (the height of the ball) is computed accurately and not as a staircase. Events fire between steps: after each step and after each action, if the condition of an event holds, the event is applied, and nothing else can happen while some event can fire. With these additions the problem becomes a search like those of Chapter 3, over states that now carry numbers and are reached by actions and waits. The waits do not appear in the plan: they are the gaps between its lines.

This explains the look of the plans we have seen. Every time in them is a multiple of 0.1. The valve of the tank was closed at 4.0 seconds, the first step in which the goal held, and not at 4.5. The robot that coasted switched off at 15.4 seconds, the last step before its battery ran out. And the first plan for the car, with its 33 seconds at 0.3 m/s, is two short waits, a long wait and a few actions: few search steps, a lot of time.

Validation in continuous time

A plan found in steps is only a candidate, because the world of the model does not move in steps. Suppose the level of a tank crosses its capacity between two steps and comes back: the stepped search never sees the flood. For this reason Kairos validates every candidate **in continuous time** before printing it. The validator integrates the processes exactly and fires each event at the instant its condition is crossed, wherever that instant falls. A candidate that the steps reach but continuous time rejects is discarded, and the search goes on. Every PDDL+ plan that Kairos prints has passed this check, and `kairos validate` applies the same check to any plan, including the ones we write by hand, whose times need not be multiples of anything.

Kairos searches in steps of 0.1 s and validates in continuous time. What it prints is correct; what it cannot print is a plan whose timing is finer than the steps.

What it can miss

The price is paid on the other side. A plan that needs a decision at an instant that is not on the grid of steps is not found. Problem `tank-p04` is `p01` with an inflow of 3 liters per second. The level reaches 8 liters at 2.667 s, between the steps at 2.6 and 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
```

No harm here: the goal asks for *at least* 8 liters, and at 2.7 s the tank holds 8.1. The goal is a wide target and some step lands on it (Figure 17.7). Had the goal been a level between 7.95 and 8.05 liters, no step would land on it, and Kairos would answer `no plan found` although closing at 2.667 s solves the problem. This is why that answer is not called `unsolvable`, and why the notes that come with it say "evidence but not a proof". Chapter 18 comes back to this limit and to how to model around it.

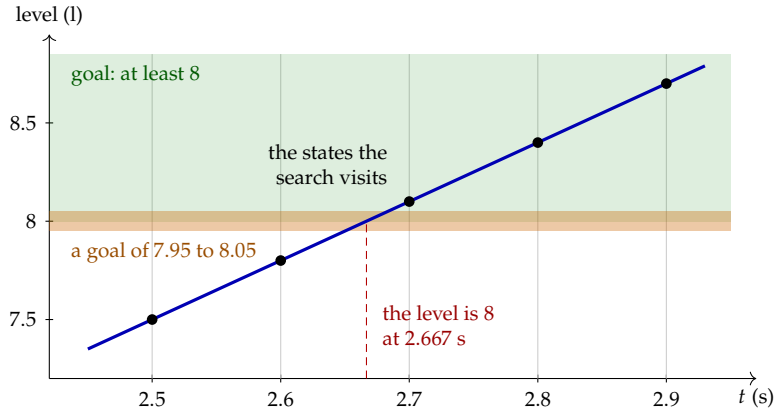


Figure 17.7.: The tank with an inflow of 3 l/s, close to the instant in which it holds 8 liters. The line is the level in continuous time; the dots are the states that the stepped search visits. A goal of "at least 8" contains dots. The narrow orange band contains the line for a moment and no dot.

Other ways to plan PDDL+

Planners for PDDL+ differ in how they treat time. Some fix a step, as Kairos does, and search over the stepped model; ENHSP Scala et al., 2016 is the best known of them. Others translate the problem into equations over real numbers and give them to a constraint solver, which finds exact instants but restricts the kind of change that can be written. All share the idea of checking the result against the continuous semantics, which is also what the validator VAL does for plans of any planner Howey et al., 2004. Few planners accept PDDL+ at all: of the eight planners in the support table of Appendix B, only Kairos planned the test domain with processes. A model that needs processes and events is therefore the least portable kind of model in this book.

17.7. The MODEL method: who causes each change

Until now, step D asked what can be done. From now on it asks what can be done *or can happen*, and the first job of the modeler is to sort every change of the story by its cause. Table 17.2 lists the questions that this chapter adds.

Table 17.2.: What processes and events add to the MODEL method.

Step	New questions
M	Does the world change when the agent does nothing? Does waiting cost something, or achieve something?
O	For every quantity that changes by itself: which fact switches the change on and off, and which number measures how far it has gone?
D	For every change: who causes it? The agent, by a decision (action); the world, while a condition holds (process); the world, at the instant a condition becomes true (event).
D	For every event: what makes its condition false again once it has fired?
E	Is there a stretch of time in which a good process runs for free, or a bad one is switched off? Write the problem that needs it.

A quick test for the third question: imagine that the agent falls asleep. Whatever still happens is a process or an event. Whatever stops happening was an action. The water keeps flowing (process), the tank overflows (event), the valve stays as it was (so opening and closing it are actions). And if something keeps happening only for a fixed time after the agent started it, such as a move that ends by itself, both views are possible, which is the subject of the comparison that follows.

Alternatives compared

How should the end of a move be modeled? We wrote it as an event. Two other encodings are reasonable, and all three are in the examples of this chapter, each with the problem `p01`:

- **Event** (`courier-battery`): `start-move`, process `travel`, event `arrive`. The model of Section 17.5.
- **Action** (`courier-battery-stop`): the event is replaced by an action `stop-move` with the same precondition and effects. The agent decides when the robot "is" in the next room, any time after covering the distance.
- **Durative action** (`courier-battery-durative`): the three pieces are folded back into the durative `move` of Chapter 10, with `(over all (powered ?r))`. The battery is still a process and `shutdown` still an event: durative actions and processes can share a model, as Chapter 18 explains.

With the action, Kairos returns a plan of 44 seconds in which the robot reaches the pharmacy at 10 s and "stops" at 16: for six seconds it is nowhere, with the battery draining. With the other two the plan takes 38 seconds. The numbers of the Size and Guidance rows are those that `kairos -v` reports for `p01`.

	Arrival as an event	Arrival as an action	Durative move
Fidelity	Good: the robot arrives when it arrives. A robot that dies on the way stays there.	Allows plans in which the robot lingers between rooms; no real robot does that. No solution is lost.	Good. A shutdown during a move breaks the `over all` condition, so such plans are invalid rather than stuck.
Size	32 ground actions, 15 processes, 15 events	46 ground actions, 15 processes, 1 event	18 actions and 14 durative actions, 1 process, 1 event
Guidance	76 139 states, 0.4 s	290 833 states, 1.4 s: every arrival is one more decision with many possible times	279 states, 0.01 s: a move is one decision and its end is a known time
Clarity	Three pieces per activity; the plan shows only the starts	The plan shows starts and stops, like a log	One piece per activity; the plan shows durations
Portability	PDDL+ only	PDDL+ only	PDDL+ planners that also accept durative actions

Table 17.3.: Three encodings of the end of a move, on `courier-battery-p01`.

Verdict. When an activity ends by itself, do not make its end a decision: the action `stop-move` loses on Fidelity and on Guidance and wins nothing. Use an action to end an activity only when stopping really is a choice (closing a valve, releasing a pedal). Between the event and the durative action, prefer the durative action whenever the activity has a duration known at its start and nothing can interrupt it half way and let it resume: the search is two orders of magnitude smaller. Keep the event when the end depends on how the world evolves: a robot that slows down when loaded, a tank that fills at a rate that other processes change, a move that a shutdown should freeze and not cancel.

17.8. Common mistakes

An event that does not switch itself off

The event `overflow` written with the precondition `(>= (level) (capacity))` alone looks harmless (`tank-wrong-loop-domain.pddl`). `kairos check` accepts the domain: the mistake is not in the syntax. The symptom appears when a plan reaches the event. The planner finds no plan for the problem whose goal is the flood, and the validator says why for a plan written by hand:

```
tank-wrong-loop-p02.plan: error: events keep firing at
5.000000 (overflow): an event must make its own condition
false [validate.event_loop]
```

Fix: every event needs, in its precondition, something that its own effect makes false. A fact that it adds and negates (`(not (flooded))`), a fact that it deletes (`(powered ?r)` in `shutdown`), or a number that it moves back across the threshold (the velocity in `bounce`).

A process without #t, or #t outside a process

A process changes numbers at a rate, so its effect must multiply by `#t`. Writing `(increase (level) (inflow))` in the process `fill` is rejected:

```
tank-wrong-jump-domain.pddl:29:13: error: in process 'fill':
processes can only have continuous effects (with #t)
[semantic.discrete_effect_in_process]
```

The opposite mistake is to give the continuous effect to the action that starts the change, "because opening the valve fills the tank":

```
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]
```

Fix: the action adds a fact; a process with that fact as its condition does the filling. A jump in a number at an instant belongs in an action or an event.

A process that runs when it should not

The robot that coasts while switched off (Section 17.5) is the typical bug of a hybrid model: the condition of a process is too weak. Nothing rejects the domain, and ordinary problems give ordinary plans. The symptom is a plan for a problem that should have none, or a plan that switches things on and off at odd moments. Fix: for each process, list everything that must be true for the change to go on, and write all of it in the precondition. Then run the cheat test.

Reading "no plan found" as "unsolvable"

On a PDDL+ problem without a plan Kairos ends with `kairos: no plan found (exhausted)` and exit code 14, never with `unsolvable` and 10. It has tried all the timings on its grid of steps, not all the timings. Before concluding that a model is too strict, check whether the plan you expected needs an instant between two steps: write it by hand and give it to `kairos validate`, which has no grid.

Forgetting `:time`, or using the keywords of a durative action

A domain with processes or events and without `:time` in its requirements is accepted with a warning for each feature:

```
tank-no-time-domain.pddl:3:18: warning: uses processes but
:requirements does not include :time
[semantic.undeclared_requirement]
```

Other planners may refuse the file. And processes and events take `:precondition` and `:effect`, like actions; writing `:condition`, as in a durative action, is an error (`unknown key :condition in process or event`).

17.9. Summary

PDDL+ separates what the agent decides from what the world does. The agent has instantaneous actions. The world has processes, which change numbers continuously while their condition holds, and events, which change the state at the instant their condition becomes true. A plan lists the decisions with their times and where it ends; the rest follows from the model and must be replayed to be read.

- `(:process name :parameters (...) :precondition C :effect E)`: runs while `C` holds; `E` has only `increase` and `decrease` with `#t`. Requirement `:time`.
- `(:event name :parameters (...) :precondition C :effect E)`: fires when `C` becomes true; `E` is like the effect of an action and must make `C` false.

- Actions control processes indirectly, through the facts in their conditions. The start-process-stop pattern gives the agent an activity it can start and end.
- Bad outcomes are events that leave a mark the goal refuses; the planner avoids them by keeping their condition from becoming true.
- In the courier world, a move is an action, a process and an event; the battery drains while the robot is on and an empty battery switches it off; a sample warms up out of the fridge and spoils.
- Kairos searches in steps of 0.1 s, integrates processes with Runge-Kutta, fires events between steps and validates every plan in continuous time. Its plans are valid; a plan that needs finer timing is not found, and the answer is then `no plan found` (exit code 14), not `unsolvable`.
- The MODEL method asks of every change who causes it, and tests hybrid models for processes that run for free.

17.10. Further reading

The definition of PDDL+ is the article of Fox and Long Fox and Long, 2006: its first sections motivate processes and events with the examples used here (the car is theirs) and can be read without the formal part; the semantics, by translation to hybrid automata, is in the second half. The continuous effects of durative actions, of which processes are the generalization, are defined in the PDDL 2.1 article Fox and Long, 2003. The book of Haslum and others Haslum et al., 2019 has a short chapter on PDDL+ in the same notation. For how a planner searches a discretized PDDL+ problem, see the paper on ENHSP Scala et al., 2016; for how plans with continuous change are validated, the paper on VAL Howey et al., 2004. What Kairos does is described in its documentation and in Martín Rico et al., 2026.

17.11. Exercises

Exercises for Section 17.2 — Processes: change while a condition holds

Exercise 17.2.1. The tank has an inflow of 4 liters per second and a capacity of 10, and the goal asks for at least 9.5 liters, the valve closed and no flood. Before running anything: between which two instants must the valve be closed? Which time will the plan of Kairos show? Write the problem and check.

Exercise 17.2.2. The tank has a leak: it loses half a liter per second whenever there is water in it. Add the leak to the domain as a second process. How long does it take now to reach 8 liters with an inflow of 2? Run it and explain the time in the plan. What happens to the level after the valve is closed?

Exercises for Section 17.3 — Events: change at an instant

Exercise 17.3.1. In `tank-domain.pddl`, a student removes the conjunct `(not (flooded))` from the event and the goal `(not (flooded))` from the problem `p01`, "because the event is not needed for that problem".

Does Kairos still find the plan of Section 17.2? What happens with the plan that closes the valve at 6 seconds? Explain both answers.

Exercise 17.3.2. Add a float switch to the tank: an event that closes the valve by itself when the level reaches `(switch-level)`. With the switch at 9 liters, write a problem whose plan has a single action, and a problem that shows that the tank can no longer flood.

Exercises for Section 17.4 — Three kinds of change

Exercise 17.4.1. Take the plan for `ball-p01` and move the `catch` to 3.0 seconds. Compute by hand the height and the velocity of the ball at that instant, predict the message of the validator, and check.

Exercise 17.4.2. Change the goal of `ball-p01` to catching the ball after exactly one bounce: `(= (bounces) 1)`. Run it. Is the answer a limit of the planner or a fact about the ball? Justify it with numbers.

Exercise 17.4.3. Classify each of these as an action, a process or an event, for a model of an elevator used by one person: the cabin goes up; the door opens when the cabin reaches a floor; the person presses a button; the person walks in; the door closes five seconds after opening; the cabin is overloaded.

Exercise 17.4.4. Problem `car-p01` with a speed limit of 1 m/s. Design by hand the fastest plan, compute its duration, and validate it. Then run Kairos and compare. What does the comparison say about the note that Kairos prints with every PDDL+ plan?

Exercise 17.4.5. The car burns fuel: 1 ml per second while the engine runs and 2 ml per second more while the acceleration is positive. When the fuel runs out the engine dies, and a car with a dead engine cannot be parked. Extend the domain. Find a plan with 15 ml, and a quantity of fuel for which there is no plan.

Exercises for Section 17.5 — The courier world in PDDL+

Exercise 17.5.1. In `courier-battery-p01`, what is the highest drain rate for which the delivery is still possible? What does the battery show when the robot picks the medicines? Answer from the plan, without running anything.

Exercise 17.5.2. Write a problem for `courier-battery` in which the robot starts switched off in the pharmacy, next to the medicines, with the battery at 15. Predict the plan and its first action.

Exercise 17.5.3. Suppose that `pick` did not have `(powered ?r)` in its precondition. Describe a plan that the model would then accept and the world would not. Is it a missing precondition or a process that runs for free?

Exercise 17.5.4. For `courier-cold-p01` there are three plans in the text. Order them by number of actions and by duration. Which one would a planner that minimizes actions prefer, and why did Kairos not return it? Which one would you want the robot to execute, and what would have to be in the model for the planner to prefer it?

Exercises for Section 17.6 — How Kairos plans PDDL+

Exercise 17.6.1. Kairos answers `unsolvable` (exit code 10) for a classical problem without a plan and `no plan found` (exit code 14) for `tank-p03`. Explain the difference to a classmate who thinks the second answer is a bug.

Exercise 17.6.2. Write by hand a plan for `tank-p01` that closes the valve at 4.05 seconds. Is it valid? Could Kairos have returned it? Does it matter?

Exercises for Section 17.7 — The MODEL method: who causes each change

Exercise 17.7.1. A coffee machine: when it is switched on, a heater warms the water at 5 degrees per second; a thermostat cuts the heater at 95 degrees; coffee can be brewed from 90 degrees. Write the action table, marking the cause of each change, and then the domain. How long does a coffee take from water at 20 degrees?

Exercise 17.7.2. Apply the five criteria to this choice in the cold-chain model: the warming of the sample as a process (as in the text) or as an effect of the moves (each `start-move` with a sample in the hand adds the degrees of the trip). Give one story in which the second encoding accepts an impossible plan.

18. Modeling hybrid systems

Chapter 17 gave us the two constructs of PDDL+ and three small worlds in which each law could be read on its own: one process fills the tank, one event floods it. Real systems are not so tidy. In a greenhouse the heater warms the air while the cold outside cools it, and the plants grow only when the result of that fight stays in a narrow band. A thermostat reacts to the temperature, and its reaction changes the temperature that it reacts to. A robot that waits for the pharmacy to open is spending battery, and whether it survives the wait depends on a clock that no action controls. And some rules are not about any moment in particular: the car must *never* exceed the limit, the plants must *never* freeze. With the constructs of the previous chapter each of these can be written, but writing them well takes more than knowing the syntax.

This chapter is about modeling with processes and events. It adds one construct, the global constraint, for rules that must hold at all times. It looks at change that is not linear, where the rate of a quantity depends on other quantities or on itself. It studies what happens when several processes touch the same number and when events trigger other events, including the ways in which a set of events can fire for ever, and how to prevent it. It puts processes together with the durative actions and timed literals of Part IV. Then it takes one system, the courier that must recharge, and models it at three levels: numbers only, durative actions with continuous effects, and PDDL+. The comparison is the main result of the chapter: most problems do not need PDDL+, and knowing when they do is a modeling skill.

The chapter follows that order. After the three models come the limits of what a planner in steps of time can find, and what the modeler can do about them. The section on the MODEL method collects a procedure for testing hybrid models and a decision guide among the three levels. As always, mistakes and exercises close the chapter.

18.1 Rules that hold at all times: global constraints 386
18.2 Change that is not linear 388
A car with air drag . . . 388
18.3 Processes that interact . 390
A drone that lands . . . 392
18.4 Events that cascade, and events that never stop . 394
Events as devices: the thermostat 395
Events that never stop . 396
18.5 Processes with durative actions and timed literals 398
Waiting is not free . . . 399
18.6 One system, three models 401
What each level cannot say 402
18.7 Limits: steps and tolerances 403
An event does not rescue the search 404
Exact values 405
18.8 The MODEL method: testing and choosing hybrid models 406
Alternatives compared . 407
18.9 Common mistakes . . . 407
18.10 Summary 408
18.11 Further reading 409
18.12 Exercises 409

18.1. Rules that hold at all times: global constraints

In the car of Chapter 17 the speed limit was enforced in two steps. An event, `radar`, added the fact `(fined)` when the velocity passed the limit, and the goal of the problem asked for `(not (fined))`. It works, and it has two weaknesses. The rule lives half in the domain and half in each problem: a problem whose author forgets the goal has no speed limit. And the search can wander through states in which the car has already been fined, from which no plan can be completed, without knowing that they are dead ends.

A **global constraint** states the rule directly: a condition that must be true at every instant of every plan. Listing 18.1 is the car again, with the event replaced by a constraint.

```
;; 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)))
```

Listing 18.1: The car with the speed limit as a global constraint (`examples/ch18/car-limit-domain.pddl`).

A constraint has a name, parameters and a `:condition`. It has no effect: it does nothing, it only forbids. With parameters, there is one constraint for each way of instantiating them (one per robot, one per tank). A plan is valid only if every constraint holds in the initial state, after every action and event, and all the time that processes run in between.

A global constraint is a condition that every instant of every plan must satisfy. It replaces the pair "event that marks the failure, goal that refuses the mark".

The construct `:constraint` is not in PDDL+ as Fox and Long defined it. It comes from the ENHSP planner Scala et al., 2016 and is common in the PDDL+ benchmarks written since. Kairos reads it as ENHSP does and asks for no requirement of its own. [1] `kairos check` lists it among the features of the file:

```
$ kairos check car-limit-domain.pddl car-limit-p01.pddl
valid, plannable
  numeric-fluents: full
  continuous-effects: full (linear ones as discrete changes in
     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
```

1: Do not confuse it with the `(:constraints ...)` section of PDDL 3 (Chapter 15). That one holds formulas like `(always ...)` over the states that a plan visits; this one is a named declaration, in the singular, checked in continuous time.

The problem `car-limit-p01` is `car-p01` without the conjunct about the fine: the goal is only to be parked at ten meters or more. Kairos returns the same two plans as in Chapter 17, the one of 33.8 seconds and then the one of 7.5. The difference shows with a plan that breaks the rule. The driver in a hurry keeps the pedal down for 3.2 seconds:

```
$ 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
```

With the event, the validator said that the goal did not hold at the end. Now it says which rule was broken and when: just after 2 seconds, the instant at which the velocity passes 2 m/s (the values are printed rounded, hence the odd-looking `2 <= 2 is false`). The diagnosis is better, and it does not depend on the goal.

When is the event still the right tool? When the bad thing is not forbidden but has consequences. A fine that costs money and can be paid, a sample that spoils and must be replaced, a battery that runs out and leaves the robot to be rescued by another: these are states of the world that plans may pass through, and they need an event that changes the state. A constraint is for what must not happen at all. The comparison table of this section's two encodings is Exercise 18.1.1.

18.2. Change that is not linear

The effect of a process has the form `(* #t` *rate*`)`. What the rate is decides the shape of the change.

- The rate is a **constant**, or a fluent that nothing changes while the process runs: `(inflow)`, `(drain-rate ?r)`. The quantity is a straight line in time. This is **linear change**, the only kind in Chapter 12.
- The rate is **another quantity that is itself changing**: the distance of the car grows at the rate `(velocity)` while the velocity grows at the rate `(acceleration)`. The distance is a parabola. Each link in such a chain raises the degree by one.
- The rate depends on **the quantity itself**: a hot body cools faster the hotter it is; air drag grows with the speed that it reduces. The quantity follows a curve that is not a polynomial at all: an exponential that approaches a limit without reaching it.

PDDL+ writes the three in the same way. The second and third are **non-linear change**, and they matter for two reasons: many planners that accept continuous effects accept only the first kind, and the instant at which a non-linear quantity crosses a threshold cannot be found by dividing a distance by a speed.

A car with air drag

Listing 18.2 adds drag to the car. A second process slows the car down in proportion to the square of its velocity, the usual model for air resistance.

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

Listing 18.2: The processes and the event of the car with air drag (part of the domain `car-drag`).

Two processes change the velocity, and their rates add up: with the pedal down the velocity grows at $a - kv^2$, where a is the acceleration and k the drag coefficient. As the car speeds up the second term grows, until it cancels the first. At that *terminal velocity*, $\sqrt{a/k}$, the car stops gaining speed. With $a = 1$ and $k = 0.25$ it is 2 m/s: this car needs no speed limit, because the physics gives it one.

The event `halt` repairs a weakness of the first car. There, `park` needed the velocity within 5 cm/s of zero, a window that the plan had to hit while braking. Here, when the braking car reaches zero, the event stops it: velocity and acceleration become exactly zero, and `park` can ask for exactly zero and be applied at any later time. An event that fires at a crossing turns a narrow window into a stable state. We will need this idea again in Section 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
```

The driver accelerates for 6.4 seconds, then brakes twice in the same instant (the first `brake` releases the pedal, the second one brakes), and parks at 8 seconds. Why 6.4? Figure 18.1 replays the plan. The velocity rises quickly at first and then bends towards 2 m/s without touching it. At 6.4 seconds the car has covered 10.03 m: that is the first step at which the goal distance is reached. Braking from 1.99 m/s takes 1.57 s, helped by the drag, and the event `halt` fires at 7.97 s, not at a multiple of anything. The plan parks at the next step.

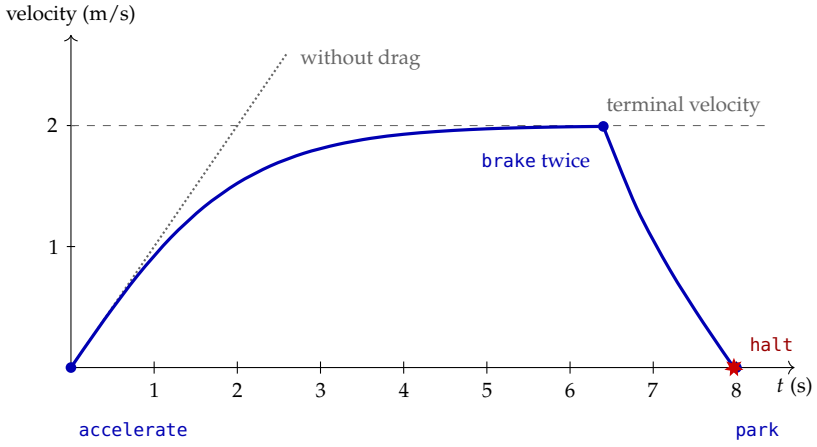


Figure 18.1.: The velocity of the car with drag in the plan of Kairos. With a constant rate the velocity would follow the dotted line; with drag it bends towards the terminal velocity. The star is the event `halt`, at 7.97 s.

Why a staircase is not enough

The simplest way to compute a changing quantity in steps is to assume that its rate stays fixed during each step. For linear change this is exact. For anything else it is wrong by an amount that accumulates. Take the ball of Chapter 17, dropped from 5 m: computing the height in steps of 0.1 s, each with the velocity at the start of the step, gives 0.5 m after one second, when the ball is in fact on the floor. A planner that worked that way would place the bounce a step late, and every later instant would inherit the error. Kairos integrates the active processes with a Runge-Kutta method, which evaluates the rates at several points inside each step. Its features table summarizes it: non-linear numeric change is "planned and validated exactly; the heuristics linearize non-linear conditions at each state". The last part means that the estimates that *guide* the search are rougher than the computation that *checks* each state, which costs search effort but never validity.

For Portability, non-linear change narrows the field again. Planners that accept the continuous effects of Chapter 12 usually expect constant rates. A model whose rates depend on changing quantities is, in practice, a model for a PDDL+ planner.

18.3. Processes that interact

Two processes interact when one changes a number that the other reads or changes. The drag and the engine of the car were a first case. The greenhouse of Listing 18.3 has the three ways in which processes meet.

```
;; 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))))
```

Listing 18.3: A greenhouse in winter (domain `greenhouse`).

- **Two processes change the same quantity.** `heating` raises the temperature by a constant 3 degrees per second while the heater is on. `losing-heat` lowers it all the time, by a tenth of the difference with the outside per second. Their rates add up. Neither process knows about the other.
- **A process changes what its own rate depends on.** The loss is proportional to the temperature it reduces: the third kind of change of the previous section.
- **A process is switched by a quantity that others change.** `growing` has no fact in its condition, only numbers: it runs while the temperature is between 22 and 28 degrees. The agent has no action that starts the growth. It can only steer the temperature into the band and keep it there.

A global constraint keeps the plants alive: never below 10 degrees, never above 30. Before running the planner, step E for a hybrid model begins with a pencil. Where does the temperature go if the heater stays on? It rises until loss equals heating: $0.1\,(T - 5) = 3$, that is, $T = 35$ degrees, above what the plants stand. And if it stays off? It falls towards the 5 degrees of the outside, below what they stand. Neither position of the switch can be left alone. The plan must switch back and forth, and the planner must find when.

```
$ 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
```

Figure 18.2 replays it. The heater is switched on at the start (one step late, which costs a tenth of a degree and nothing else). The temperature enters the growing band 4.5 seconds later and climbs. At 10.7 seconds it is at the top of the band, and the heater goes off; the temperature falls, faster than it rose, and at 13.8 seconds, at the bottom of the band, the heater goes on again. The cycle repeats until the plants have had their 20 seconds of growth. Nobody told the planner to keep the temperature between 22 and 28: the goal asks for growth, growth happens only in the band, and the search found the switching times that keep it there. Control engineers call this kind of control, all or nothing, *bang-bang*.

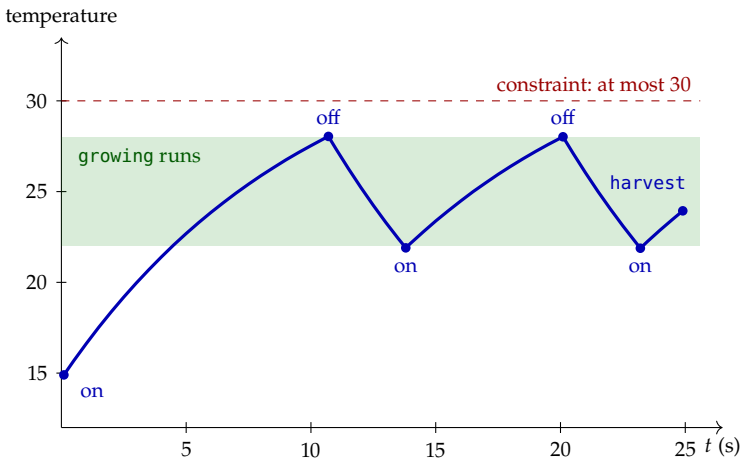


Figure 18.2.: The temperature of the greenhouse in the plan of Kairos. Each dot is an action. With the heater on the curve bends towards 35 degrees, with the heater off towards 5: the rises are slower than the falls. The plants grow while the curve is in the band.

A drone that lands

The second example of interacting processes has no band to stay in but a target to hit: the ground, slowly. A drone hovers at 20 meters and must land. Listing 18.4 shows its laws.

```
;; Flight rules, at every instant
(:constraint below-ceiling
  :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))))
```

Listing 18.4: The laws of the drone (part of the domain `drone`).

Gravity acts all the time on the velocity; the rotors, while they are on, push the other way and drain the battery. The same process, `pushing`, changes two quantities of different kinds: it is how the model says that thrust costs energy. A constraint keeps the drone under a ceiling, and three events watch for thresholds.

The last two events deserve attention. Reaching the ground ends the flight in one of two ways, and the model has one event for each. Their conditions share `(<= (altitude) 0)` and differ in the last conjunct: at most `(max-impact)` for `touch-down`, more for `crash`. The two conditions *exclude each other*. This is a rule worth keeping: when two events can become true at the same instant and both change the same facts, make their conditions exclusive, so that the model never has to say which goes first.

```
$ 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
```

The drone falls freely for 1.1 seconds and reaches 11 m/s. The rotors, which give 15 m/s^2 against the 10 of gravity, then slow it down by 5 m/s every second. After a pause of a tenth of a second they stay on until the ground, which the drone touches at 3.28 seconds at 1.58 m/s, under the 2 m/s allowed (Figure 18.3). The event `touch-down` fires there, and the plan ends at the next step. The rotors ran for 2.08 seconds and used 10.4 of the 30 units of battery. Notice the pause at 2.0: the search moves in steps of 0.1 s, and one single burn started at a step would touch the ground too fast or stop the drone in the air. A tenth of a second of free fall is how the search adjusted the landing speed with the only resolution it has.

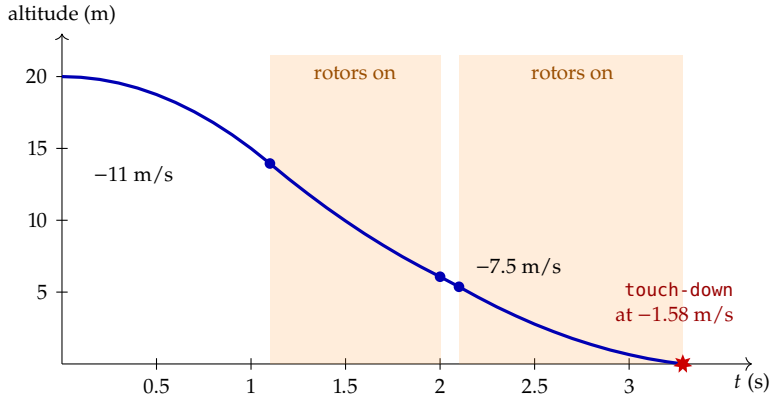


Figure 18.3.: The descent of the drone in the plan of Kairos. In the white stretches only gravity acts and the curve bends down; in the shaded ones the rotors are on and it bends up. The dots are the actions.

18.4. Events that cascade, and events that never stop

When an event fires, its effects may make the condition of another event true. That one fires at once, in the same instant, and may trigger a third. A chain of events at one instant is a **cascade**. The cleanest example has no numbers in it (Listing 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)))
```

Listing 18.5: Dominoes: one event per pair of neighbors (`examples/ch18/dominoes-domain.pddl`).

```
$ kairos dominoes-domain.pddl dominoes-p01.pddl
; cost = 0
; makespan = 0.000
0.000: (push d1) [0.000]
0.000: @PlanEND
```

The agent pushes the first domino, the only one within reach. The goal is the fifth, and the plan ends at time zero: four events fired one after the other without any time passing. The cascade stops because each event switches itself off (its domino is now fallen) and the row has an end.

A cascade is also something that the agent cannot interrupt. Problem `dominoes-p02` asks for the second domino down and the third up, and

has no plan: between the event that knocks down `d2` and the one that knocks down `d3` there is no instant in which an action could be placed. If the story needs the agent to be able to step in, time must pass between the events: a process and a threshold, as in Exercise 18.4.2.

Events at the same instant fire in a cascade that no action can interrupt. A cascade must end: each event must switch itself off, and no event may switch another back on.

Events as devices: the thermostat

In Chapter 17 events were accidents: a flood, a shutdown, a spoiled sample. Events are also how PDDL+ models the automatic devices of a system. The bang-bang plan for the greenhouse had five switch actions that a real greenhouse would not leave to a planner: a thermostat does that. The domain `greenhouse-auto` removes the two switch actions and adds one action, `enable-thermostat`, and two events (Listing 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)))
```

Listing 18.6: A thermostat is two events (part of the domain `greenhouse-auto`).

```
$ 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
```

Two actions. Everything else is the world: the instant the thermostat is enabled, at 15 degrees, `too-cold` fires and the heater starts (an action that triggers an event in the same instant). Replaying the model in continuous time, `too-warm` fires at 9.16 s, `too-cold` at 11.17 s, and so on, seven events in all, at instants that are not on any grid (Figure 18.4). The temperature stays between 23 and 27, and the plants are ready half a second earlier than with the switching of the planner.

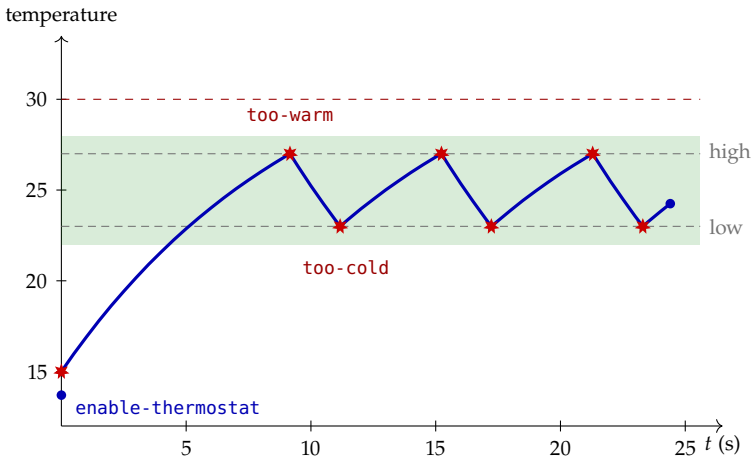


Figure 18.4.: The greenhouse under the thermostat. The stars are events: no action of the agent appears between the first instant and the harvest. Compare with Figure 18.2: the same sawtooth, drawn by the world and not by the plan.

For the modeler, moving the switching from the plan to the events is a choice of *where the intelligence is*. If the system has a thermostat, model it: the plans are shorter, the search is smaller (8 030 states instead of 432 899 in these two runs) and the plan no longer depends on the planner hitting the right tenths of a second. If the switching is what we want the planner to work out, leave it as actions.

Events that never stop

Each event of the thermostat switches itself off: `too-warm` needs the heater on and turns it off. But the rule of Chapter 17, that an event must make its own condition false, is not enough when there are several events. Two events can each switch *the other* on. The blinker of Listing 18.7 is the smallest case.

```
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))))
```

Listing 18.7: Wrong on purpose: two events that undo each other (part of the domain `blinker-wrong`).

Each event, taken alone, is correct. Together, once the blinker is on, the lamp lights up, which enables `go-dark`, which enables `light-up`, and so on, infinitely many times at time zero. `kairos check` accepts the domain, since nothing is wrong with its syntax. The planner finds no plan for the problem "let three seconds pass", and the validator explains why when given the obvious plan:

```
$ 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
```

The repair is to put time between the events. In the domain `blinker` a timer, driven by a process, must reach half a second before either event can fire, and each event resets it:

```
  ;; 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))))
```

Listing 18.8: The blinker repaired: each event resets a timer that the other must wait for (part of the domain `blinker`).

```
$ kairos blinker-domain.pddl blinker-p01.pddl
; cost = 2.5
; makespan = 2.500
0.000: (switch-on) [0.000]
2.500: @PlanEND
```

The lamp lights at 0.5, 1.5 and 2.5 seconds; the third flash is the goal. The pair "process that advances a timer, event at a threshold that resets it" is the PDDL+ idiom for anything periodic and for anything delayed.

The thermostat hides a subtler version of the same problem. It has two thresholds, 23 and 27, and after each event the temperature needs seconds to travel to the other one. That gap between the thresholds is called **hysteresis**. Problem `greenhouse-auto-p02` sets both thresholds to 25. Now, the moment the temperature reaches 25, `too-warm` turns the heater off, and at that same temperature `too-cold` turns it on again. No time passes between them: the model asks for infinitely many events around one instant, which is known as *Zeno behavior*, after the paradox. In practice:

```
$ 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)
```

The planner does not return, with 3 seconds or with 20. The validator, given the plan "enable the thermostat and harvest at 40 seconds", accepts it: computing with numbers of finite precision, it never sees the temperature at exactly 25 with both conditions true, and it follows the heater as it switches on and off many times per second. Neither behavior is a verdict on the model, and that is the lesson: a model with Zeno behavior does not fail with a clear message. It fails by making the tools slow, or by making them disagree. Real thermostats have hysteresis for a similar reason, to protect the heater from switching without rest. Models need it too.

Step D: the questions for a set of events

For each event alone: what makes its condition false once it has fired? For the events together: draw an arrow from event A to event B if an effect of A can make the condition of B true. Follow the arrows. A chain that ends is a cascade, and is fine. A cycle is fine only if somewhere along it *time must pass*: a quantity has to travel from one threshold to another (hysteresis), or a timer has to run. A cycle that can close in zero time is a bug. For two events that can become true at the same instant and touch the same facts: make their conditions exclusive, as in the two ways of reaching the ground.

18.5. Processes with durative actions and timed literals

Nothing forces a hybrid model to be pure. Processes and events live in the same domain as the durative actions of Chapter 10 and the timed initial literals of Chapter 13, and the comparison at the end of Chapter 17 showed why we want them to: a move whose duration is known when it starts is better written as one durative action than as an action, a process and an event. The rule of thumb is to use each construct for the kind of change it owns:

- a **durative action** for an activity of the agent that, once started, runs its course;
- a **process** for change that depends on the state and not on what the agent is doing;
- an **event** for a consequence that the world draws by itself;
- a **timed initial literal** for a change that comes at a time on the clock, whatever the state.

The domain `courier-plus` is the courier world written that way. Moves are durative actions. The battery is the business of three processes and two events (Listing 18.9), and charging is no longer an action: the agent plugs and unplugs the robot, and the charger does the rest.

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

Listing 18.9: The battery of `courier-plus`: three processes and two events (part of the domain).

The durative `move` marks the robot as `(driving ?r)` from its start to its end. That fact is the link between the action and the process `motor-drain`: the action does not touch the battery, it only tells the world that the motors are running. The rates add up as before. A robot that is on loses 0.2 per second; driving, $0.2 + 0.8 = 1$; plugged and on, $2 - 0.2 = 1.8$ in its favor. The condition `(over all (powered ?r))` ties the other way: if the event `shutdown` fires during a move, the move is broken and the plan is invalid. The event `battery-full` is a device, like the thermostat: the charger disconnects itself.

Three problems exercise the model. In `p01` the battery is at 40 and the delivery, 34 meters from the lobby, needs 34: the plan is the classical one, four moves, a pick and a drop. In `p02` the battery is at 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
```

The robot fetches the medicines, goes to the dock with 4 units left, and plugs in. It stays plugged for 53.3 seconds and unplugs with the battery at 99.9, one step before the event `battery-full` would have done it. For the 26 meters that remain it needed 26. This is the long wait of Chapter 17 at work: once plugged, the search lets time run to the last step before some numeric condition changes, and the nearest such change was the battery filling up. The plan is valid and wasteful, and it is the same waste that a planner makes when it tries the durations of a durative `charge` only at their bounds.

Waiting is not free

Problem `courier-plus-p03` adds a clock. The battery is at 10, and the pharmacy is closed: a timed initial literal opens it at 60 seconds.

```
    ; the pharmacy opens one minute from now
    (at 60 (is-open pharmacy))
```

In the temporal models of Part IV the robot would simply wait in the lobby. Here a minute of waiting costs 12 units of battery, and the robot has 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
```

Figure 18.5 follows the battery. The robot goes to the dock at once and arrives with 6 units. Then it does something no human would: it stands next to the charger for half a minute, and plugs in at 33.9 seconds, with 0.02 units left, one step before the shutdown. It charges until the pharmacy opens, at 60 seconds exactly (timed literals keep their exact times), and does the delivery with 47 units, of which it needs 38.

The plan is correct, and it tells us how the search reasons. From the dock, a long wait runs until the last step before something changes; the first thing that would change is the battery reaching zero. There the search must act, and plugging in is the action that helps. A plan that plugs in on arrival and leaves the dock in time to be at the pharmacy when it opens is ten seconds shorter (Exercise 18.5.1), and the planner had no reason to look for it. What matters for the model is that both plans respect the same fact: the wait had to be spent at the charger.

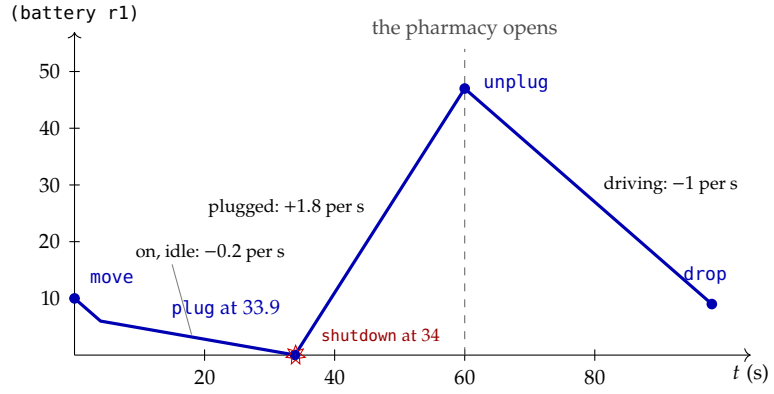


Figure 18.5.: The battery in the plan for `courier-plus-p03`. Each slope is a sum of the rates of the processes active in that stretch. The dashed line is the timed initial literal. The hollow star is the event that the plan avoids by one step.

How does Kairos mix the constructs? Its documentation says that, in a model with processes, a durative action runs as a start, a clock and an end, with its end and each timed literal at their exact times, and that there is a third kind of wait, of length epsilon, to separate happenings that must not coincide. The durations in the plan are exact for that reason, and the search is small: 736 states for `p02`, when the pure PDDL+ courier of Chapter 17 needed tens of thousands for a simpler problem.

18.6. One system, three models

The courier with a battery has now been modeled in three chapters with three sets of tools. This section puts the three side by side on the same task: the robot is in the lobby with the battery at 20, the medicines are in the pharmacy, ward A is 34 meters of travel away, and there is a charger at the dock.

Level 1: numbers. The domain `courier-discrete` uses only the numeric fluents of Chapter 8. A move subtracts the distance times the consumption; `charge` sets the battery to its capacity.

```
  (: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)
```

Listing 18.10: Level 1: the battery changes in jumps (part of the domain `courier-discrete`).

Nine actions, found in a millisecond and proved optimal. The plan says what to do and in which order. It does not say how long anything takes, least of all the charge.

Level 2: durative actions with continuous effects. The second domain, `courier-durative`, uses the tools of Chapter 12. The battery falls continuously during a move and rises continuously during a `charge` whose duration the planner chooses, up to what fills the battery.

```
  (: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]
```

Listing 18.11: Level 2: the battery changes while an action runs (part of the domain `courier-durative`).

Now the plan has times. Both plans charge for the longest duration allowed, 48 and then 42 seconds, a full battery each time: the bound of the inequality is where the search looks first. The model knows that charging takes time and that a move needs battery all the way. It does not know that a robot standing still spends anything.

Level 3: PDDL+. The domain `courier-plus` of the previous section, with its problem `p02`: the plan of 95.3 seconds that we have read. The battery drains whenever the robot is on, charging is a state of the world, and an empty battery is an event.

What each level cannot say

The three plans are the same route with the same stop at the dock. The levels differ in the problems they get *wrong*. Give each the scenario of `p03`, a pharmacy that opens in a minute and a battery at 10. Level 1 has no time, so it cannot even state the opening hour. Level 2 can state it, with a timed literal, and will answer with a robot that waits in the lobby for a minute and then delivers: in that model nothing happens to a robot that is not running an action. Only level 3 knows that the wait empties the battery. On the other side, give each a day with forty deliveries: level 1 will plan it and the other two, on most planners, will not.

	1. Numbers	2. Durative, continuous	3. PDDL+
Fidelity	Order and energy of the moves. No time: charging is instantaneous, waiting does not exist.	Time, overlap, energy during actions. Change stops when actions stop: waiting is free.	Change whenever the state says so: idle drain, shutdown, the charger that cuts off.
Size	31 ground actions	31 durative actions	11 actions, 14 durative actions, 3 processes, 2 events
Guidance	220 states, proved optimal	9 629 states; durations tried at their bounds	736 states here; in general the hardest, since every instant is a decision
Clarity	The simplest to read and to explain	One action per activity; plans are timelines	The laws are separate from the decisions; plans must be replayed to be read
Portability	Any numeric planner: five of the eight of Appendix B	Continuous effects: two of the eight	Processes: one of the eight

Table 18.1.: Alternatives compared: the courier that recharges, at three levels. The figures of Size and Guidance are those of `kairos -v` on the three problems of this section.

Verdict. Choose the lowest level that does not lie about what the problem turns on. Level 1 when the question is which actions and in which order, and quantities are budgets that actions spend: most logistics and most robot missions seen from far enough. Level 2 when time is part of the question (deadlines, opening hours, robots working in parallel) and every change can be blamed on an activity of the agent. Level 3 when something changes while the agent does nothing, or when the world interrupts: draining, warming, filling, falling, and the events that follow. Within level 3, keep as durative actions everything that would have been one at level 2. The levels are also a path: Chapter 19 shows how to grow a model from one to the next, keeping the tests of the lower level as tests of the higher one.

18.7. Limits: steps and tolerances

Chapter 17 ended its account of how Kairos plans with a warning: a plan that needs finer timing than the steps of the search is not found. A modeler should know what that looks like, because the symptom, `no plan found`, is the same as that of a model that is wrong.

The domain `doser` is a tank that must be filled to a dose of 8 liters, give or take 0.1 (Listing 18.12). At 2 liters per second (`p01`) the plan closes the valve at 4 seconds. Problem `p02` has a faster pump, 7 liters per second.

```
;; 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))))
```

Listing 18.12: The doser, with a float switch that the problem may arm (`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
```

A plan exists: the one written by hand, which closes the valve at 1.143 seconds, is valid. The level is inside the dose from 1.129 to 1.157 seconds, a window 29 milliseconds wide. The search looks at 1.1 s, where the level is 7.7, and at 1.2 s, where it is 8.4 (Figure 18.6), and at nothing in between.

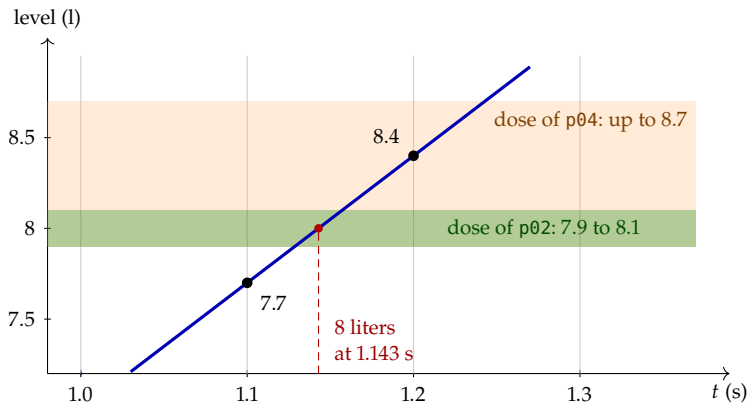


Figure 18.6.: The doser at 7 liters per second. The level (line) is inside the narrow dose (dark band) for 29 ms, between two states of the search (dots), which are 0.7 liters apart. The wider dose (light band) contains a dot.

An event does not rescue the search

The previous sections suggest a remedy: let an event do the precise part. Problem `p03` arms the float switch, which closes the valve when the level reaches 8 liters. In continuous time the plan is trivial, open the valve and wait, and the validator agrees (`doser-p03-hand.plan` is valid). Yet:

```
$ kairos doser-domain.pddl doser-p03.pddl
...
kairos: no plan found (exhausted)
```

The reason is in how the search treats events: it fires them *between steps*. In the stepped world the switch is checked at 1.1 s (7.7 liters, not yet) and at 1.2 s (8.4 liters, now), and closes the valve with 8.4 liters in the vessel. The search never visits a state with the right dose, so it has no candidate to hand to the validator, which would have fired the event at 1.143 s. The event makes the *world* precise, not the search.

What helps is a model in which the states of the search are also acceptable. Problem `p04` keeps the switch and accepts doses up to 8.7 liters, the 0.7 liters that enter in one step:

```
$ kairos doser-domain.pddl doser-p04.pddl
; cost = 1.2
; makespan = 1.200
0.000: (open-valve) [0.000]
1.200: @PlanEND
```

The search reaches 1.2 s with the valve closed by the switch and, in its own arithmetic, 8.4 liters. The validator then replays the plan in continuous time: the switch closes at 1.143 s with 8 liters, the goal holds, the plan is printed. The plan is better in the world than in the search that found it.

A window that a plan must hit should be at least as wide as what changes in one step: the rate times 0.1 s. Narrower windows give "no plan found" for problems that have plans.

From this follows a rule for the modeler. **Every window that a plan must hit should be at least as wide as the change of one step**: the rate of the quantity, times 0.1 s. A velocity that changes at 10 m/s^2 moves 1 m/s per step, so a window on the velocity should be 1 m/s wide or more. If the story needs a narrower window, ask whether the story really needs it (few valves close to the millisecond), slow the process down by modeling a finer actuator, or accept that this problem is outside what a stepped search finds and validate plans obtained otherwise.

Exact values

The extreme case of a narrow window is equality. Problem `doser-p05` asks for `(= (level) 8)`. No step lands on it, and neither does a plan written by hand to the millisecond:

```
$ 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
```

Closing at 1.143 s leaves 8.001 liters, and 8.001 is not 8. The validator compares the numbers that processes produce exactly, as it must: a tolerance generous enough to accept this plan would also hide real violations of thresholds. No decimal time gives exactly 8 liters at 7 liters per second. A quantity that a process changes should be compared with `>=` and `<=`, never with `=`. The only safe equalities are on values that an action or an event has *assigned*: the velocity that `halt` set to zero in the car with drag is exactly zero, and `park` could ask for it.

18.8. The MODEL method: testing and choosing hybrid models

The chapter adds to two steps of the method. To step E, a procedure for testing a model in which most of what happens is not in the plan. To step L, a decision that comes before any comparison of encodings: at which of the three levels to model.

Step E for a hybrid model

1. **Pencil first.** For each quantity, work out where it goes if nothing is done and if each switch is left in each position (the 35 and the 5 degrees of the greenhouse). Those numbers tell which problems should have plans.
2. **Check every event.** For each event, write a problem whose only plan passes through it, or whose goal is its effect (the flood as a goal). An event that never fired in a test has not been tested. Check that it fires once, and draw the arrows between events to look for cycles in zero time.
3. **Bound the horizon.** A problem without a plan is searched to exhaustion, and the search grows with the time during which things can keep changing. Give the "no plan" problems small margins (a battery for 16 seconds, not for 300) so that they end in seconds.
4. **Validate in continuous time.** Write by hand the plan you expect, with times that are *not* multiples of 0.1, and the plan you fear, and give both to `kairos validate`. It is the judge of the model; the planner is only one source of plans.
5. **Replay and plot.** Compute the quantities along the plan of the planner and draw them, as the figures of these two chapters do. A plan that is valid for a wrong reason (a process that ran for free) is seen at once in a plot and almost never in the list of actions.
6. **Measure the windows.** For every condition that the plan must meet while a quantity moves, compare the width of the window with the rate times 0.1 s.

Table 18.2 lists the new questions, and Figure 18.7 is the decision guide for step L.

Step	New questions
M	Is there something that must never happen, in any plan? (A global constraint.) Is a bad outcome forbidden, or does it only have consequences?
O	For each rate: is it a constant, another changing quantity, or the quantity itself?
D	Which processes touch the same quantity, and what is the sum of their rates in each situation? Which events can trigger which, and where must time pass in each cycle? Which devices of the system are events, and which decisions are left to the agent?
E	The six points of the box above.
L	Which is the lowest level (numbers, durative actions, PDDL+) that does not lie about what the problem turns on?

Table 18.2.: What the modeling of hybrid systems adds to the MODEL method.

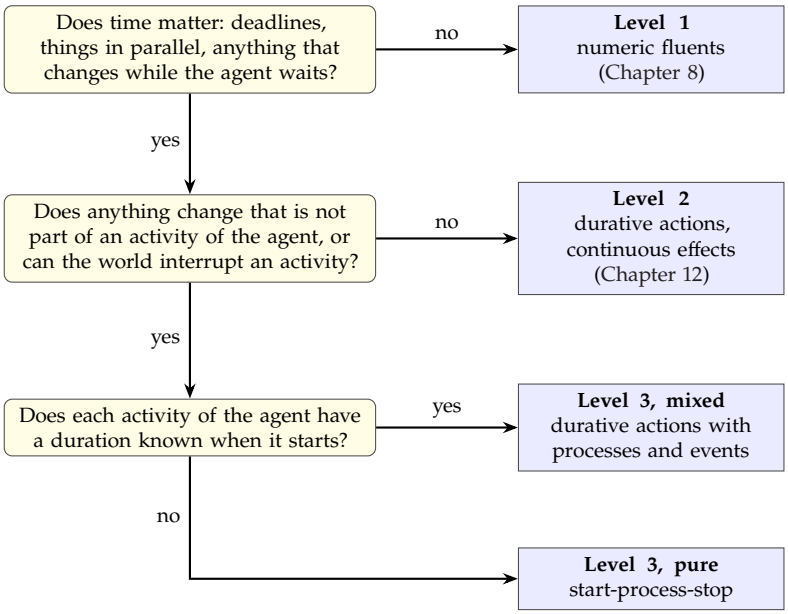


Figure 18.7.: Choosing the level of a model. Each "yes" buys fidelity and pays for it in search effort and in the number of planners that can read the model.

Alternatives compared

The alternatives of this chapter are the three levels, written and run in Section 18.6 and judged on the five criteria in Table 18.1, with their verdict. The guide of Figure 18.7 condenses that table into three questions. Inside level 3 a second choice remains, between an event that marks a failure and a global constraint that forbids it; it is compared in Exercise 18.1.1.

18.9. Common mistakes

Two events that switch each other on

Each of `light-up` and `go-dark` makes its own condition false, and together they never stop (`blinker-wrong-domain.pddl`). `kairos check` says `valid, plannable`; the planner answers `no plan found`; only the validator, with a plan that reaches the loop, names the problem:

```
blinker-wrong-p01.plan: error: events keep firing at 0.000000
(go-dark): an event must make its own condition false
[validate.event_loop]
```

The message speaks of one event because that is where the validator gave up; the cause is the pair. Fix: make time pass inside every cycle of events, with a timer or with two different thresholds.

A thermostat with one threshold

"Heater off above 25, on below 25" reads naturally and is a Zeno model. Nothing reports it. With `greenhouse-auto-p02` the planner ran out of time (`kairos: timeout (timeout)`) and the validator accepted a plan written by hand, because in finite precision the temperature is never exactly on the threshold. A slow planner and tools that disagree are the symptoms. Fix: hysteresis, two thresholds far enough apart that the quantity needs several steps to go from one to the other.

A window narrower than a step

The dose of 7.9 to 8.1 liters at 7 liters per second gives `no plan found (exhausted)` after 100 states, although a plan exists. The tell-tale sign is the size of the search: a model that is too strict usually takes many states to exhaust, and a window that is missed takes few, because the search walks past it. Confirm with a plan written by hand. Fix: widen the window to the rate times 0.1 s, or slow the process.

Equality on a quantity that flows

`(= (level) 8)` in a goal or in a precondition, when a process changes the level, is a window of width zero. The validator is exact:

```
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]
```

Fix: two inequalities with a tolerance, or an event that assigns the exact value when the threshold is crossed.

Expecting the shortest plan

The plans for `courier-plus` charge to 99.9% when 30% was enough, and wait next to the charger before plugging in. They are valid. In PDDL+ Kairos minimizes the number of actions, and its long waits stop where something is about to change, not where it would be convenient. Fix, when the duration matters: put the pressure in the model (a deadline as a timed literal, a smaller battery), or improve the plan by hand and validate it.

A constraint where an event was needed, and the reverse

A global constraint "the battery is never empty" forbids every plan in which a robot runs out, including the plans in which a second robot rescues the first. An event that marks the robot as `stranded`, with a goal that says which robots must not be, allows them. The reverse mistake is the speed limit enforced by an event and a goal that one problem forgets. Ask of each rule: is breaking it impossible, forbidden, or merely costly? Impossible is a process or event of the world; forbidden is a constraint; costly is an event with consequences.

18.10. Summary

Hybrid models are built from few constructs, and their difficulty is in the interplay.

- A **global constraint**, written with `:constraint`, a name, `:parameters` and a `:condition`, must hold at every instant of every plan. It comes from ENHSP, not from the original PDDL+. Use it for what is forbidden; use an event for what has consequences.
- Change is **linear** when the rate is constant, and **non-linear** when the rate is another changing quantity (polynomial) or the quantity itself (exponential). PDDL+ writes all three alike; Kairos integrates them with Runge-Kutta and validates exactly.
- Processes that change the same quantity add their rates. A process may be switched by a number that other processes move, and then the agent can only steer.

- Events **cascade** within an instant. A set of events must not contain a cycle that closes in zero time: put a timer or **hysteresis** in every cycle. Events with a common trigger should have exclusive conditions. Events are also the devices of a system.
- Processes and events combine with durative actions and timed initial literals. Keep as a durative action every activity whose duration is known when it starts.
- The same system can be modeled with numbers, with durative actions and continuous effects, or with PDDL+. Choose the lowest level that does not lie about what the problem turns on.
- A search in steps of time misses windows narrower than a step, even when an event would hit them. Make windows at least as wide as the rate times 0.1 s, and never test a flowing quantity for equality.
- Step E for a hybrid model: pencil first, check every event, bound the horizon, validate in continuous time, replay and plot, measure the windows.

18.11. Further reading

The second half of the article of Fox and Long Fox and Long, 2006 gives the semantics of interacting processes and of cascading events, and discusses which models are well formed; their planetary lander is a larger example in the spirit of this chapter. The treatment of non-linear numeric conditions by intervals is in the paper on ENHSP Scala et al., 2016, the planner from which the global constraints of this chapter come. For linear continuous change inside durative actions, level 2 of this chapter, see the PDDL 2.1 article Fox and Long, 2003 and the planner that made it practical Coles et al., 2010. The validation of plans with continuous effects, including how invariants are checked between happenings, is the subject of the paper on VAL Howey et al., 2004. The book of Haslum and others Haslum et al., 2019 places PDDL+ among the other extensions of the language, and Martín Rico et al., 2026 describes how Kairos chooses among its engines for each of the three levels.

18.12. Exercises

Exercises for Section 18.1 — Rules that hold at all times: global constraints

Exercise 18.1.1. Compare the two encodings of the speed limit, the event `radar` with the goal `(not (fined))` (Chapter 17) and the constraint `keep-the-limit`, on the five criteria of step L. Include in Fidelity a problem that one encoding gets wrong when its author is careless.

Exercise 18.1.2. Rewrite the tank of Chapter 17 so that the flood is ruled out by a global constraint and not by an event. What does the validator say about the plan that closes the valve at 6 seconds? What can the old model express that the new one cannot?

Exercises for Section 18.2 — Change that is not linear

Exercise 18.2.1. For each quantity say whether its change is linear, polynomial or exponential, and why: (a) the battery of a robot that

drains at 0.5 per second; (b) the height of the ball of Chapter 17; (c) the temperature of the greenhouse with the heater off; (d) the level of a tank that empties through a hole, faster the fuller it is; (e) the odometer of a robot that moves at constant speed.

Exercise 18.2.2. A cup of coffee is poured at 90 degrees in a room at 20, and cools at a rate of a tenth of the difference per second. Drinking it above 65 degrees burns the tongue; below 55 it is not worth drinking. Model it and find when to drink. Compute the earliest instant by hand and compare it with the plan.

Exercises for Section 18.3 — Processes that interact

Exercise 18.3.1. In the greenhouse, the heater has a power of 2 degrees per second instead of 3. Where does the temperature go with the heater always on? What is then the shortest plan, and how many actions does it have? Answer with a pencil, then change the problem and run it.

Exercise 18.3.2. In the plan for `drone-p01`, compute the altitude and the velocity at each of the three actions, and the speed at which the drone would hit the ground with the rotors never switched on. Which event would fire in that case, and what would the planner do about it?

Exercises for Section 18.4 — Events that cascade, and events that never stop

Exercise 18.4.1. Draw the arrows "an effect of A can make the condition of B true" between the events of four models: `blinker-wrong`, `blinker`, `greenhouse-auto` with thresholds 23 and 27, and with both thresholds at 25. For each cycle, say what must happen between one event and the next, and how long it takes.

Exercise 18.4.2. Dominoes that take time: each domino needs half a second to fall, and only when it is down does it hit the next. Model it with a process and events, so that a row of five takes 2.5 seconds. Could the agent now save the third domino, if it had an action for it?

Exercises for Section 18.5 — Processes with durative actions and timed literals

Exercise 18.5.1. For `courier-plus-p03`, write by hand a plan in which the robot plugs in as soon as it reaches the dock and is at the pharmacy when it opens. Validate it. How much shorter is it than the plan of Kairos? Why must the `pick` come a little after 60 seconds and not at 60?

Exercise 18.5.2. The pharmacy is open now and closes 8 seconds from now; the battery is at 40. Write the problem with a timed initial literal, predict whether there is a plan, and run it. Read the times of the plan: is anything in it odd?

Exercises for Section 18.6 — One system, three models

Exercise 18.6.1. Choose a level for each story with the guide of Figure 18.7, and say which question decided: (a) a truck with a tank of fuel delivers to ten shops; (b) two robots paint a room, and a wall can

be given a second coat only when the first has dried for an hour; (c) an oven that keeps heating while the cook prepares the dough; (d) a satellite that can transmit only while a ground station is visible, at times known in advance.

Exercise 18.6.2. In the level 2 model of Section 18.6 the charge lasted 42 seconds, and in the level 3 model 53.3. Explain each number. Then explain why neither planner charged for the dozen seconds that the task needs.

Exercises for Section 18.7 — Limits: steps and tolerances

Exercise 18.7.1. The doser of `p02` with a pump of 5 liters per second. Is the window wider than a step? Does Kairos find a plan? Reconcile the two answers.

Exercise 18.7.2. A classmate proposes to solve `doser-p03` by setting the target of the float switch to 7.6 liters. Predict, with the stepped view of the search, the level at which the search sees the valve close, and the level at which it closes in continuous time. Is the proposal a fix?

Exercises for Section 18.8 — The MODEL method: testing and choosing hybrid models

Exercise 18.8.1. Complete the test set of the drone. Besides the typical problem (`p01`), write a trivial one, one that must have no plan, and one that tempts the planner to break a rule. For each, say what you expect before running it.

Exercise 18.8.2. Apply the six points of step E to the domain `courier-plus`: for each point, name the problem or the plan of this chapter that covers it, or say that it is missing and write what should be added.

Part VII. The method at work

19. The MODEL method, complete

A student who has read the eighteen chapters before this one knows every construct of PDDL and has seen each of them at work. Now give that student a blank file and a paragraph: *a laundromat, three loads of clothes, one washer, one dryer, a woolen sweater that must not be tumble-dried, and a morning that is not endless*. Where does the first line come from? The answers are in the book, but they are spread over it. The question about absence is in Chapter 5, the one about quantities in Chapter 8, the rule that tells an action from a process in Chapter 17, the pattern for a shared machine in Chapter 7 and again, with a clock, in Chapter 10. Nobody models with seventeen chapters open at once.

This chapter puts the MODEL method in one place. It teaches no new PDDL. It collects what each chapter added to the five steps, orders it, and connects it: the questions as five checklists; a guide from what a problem needs to the feature that provides it, with what the feature costs; one index of the patterns and one of the anti-patterns of the whole book; and the debugging procedure of Chapter 7 extended to numbers, time and hybrid systems. It also adds two things that no single feature could teach: how to grow a model in versions, from classical to PDDL+, without losing what was already right, and how to document a model so that somebody else can use it.

The chapter starts with the method on one page and the five checklists. Then come the decision guide, the two catalogs and the debugging procedure. The second half is a worked example on a domain that is new to the book, a laundry, taken through five versions, each one a tested file. The chapter ends with the templates for documenting a model, a last comparison of alternatives, and exercises that are no longer exercises but projects.

19.1 The method on one page 413
19.2 The five checklists . . . 415
19.3 Which feature for which need 417
What each feature costs 419
19.4 The catalog of patterns 420
19.5 The catalog of anti-patterns 423
19.6 Debugging, beyond classical 425
Read the verdict first . . 425
No plan, with a proof . 426
No plan, without a proof 426
A wrong plan, and a poor one 427
19.7 Growing a model in versions 428
Version 1: classical . . . 429
Version 2: numbers . . . 431
Version 3: time 432
Version 4: wishes 434
Version 5: processes . . 436
Five versions, side by side 438
19.8 Documenting a model . 438
19.9 The MODEL method: the whole method . . . 441
Alternatives compared . 441
19.10 Common mistakes . . . 442
19.11 Summary 444
19.12 Further reading 444
19.13 Exercises 445

19.1. The method on one page

The five steps have not changed since Chapter 1. What has grown is the list of things to ask at each of them. Table 19.1 states the steps once more, with the artifact that each one leaves, and Figure 19.1 shows which chapters added questions to which step.

Step	Question	Artifact
M Map the situation	What is the problem, who acts, what counts as solved, what is left out?	The *model card*: Story, Agents, Success, Scope, Assumptions; later also the metric, the wishes, the rules and the schedule of the world
O Objects and state	Which things exist, and what must the planner know about them to decide?	The *state table*: each property, what it is about, static or dynamic, how many values, how it is encoded
D Dynamics	What can be done or can happen, when, and what does it change?	The *action table*: each action, process and event; its parameters, when it applies, what it changes, how long it lasts
E Exercise the model	Does the planner do with it what the world would allow, and nothing else?	The *test set*: a trivial problem, a typical one, one with no plan, one that tempts the planner to cheat; every plan read and validated
L Look again	Which other encoding would do, and which is better here?	The *comparison*: two or three alternatives on the five criteria, and a decision with its reason

Table 19.1.: The MODEL method: five steps, five questions, five artifacts.

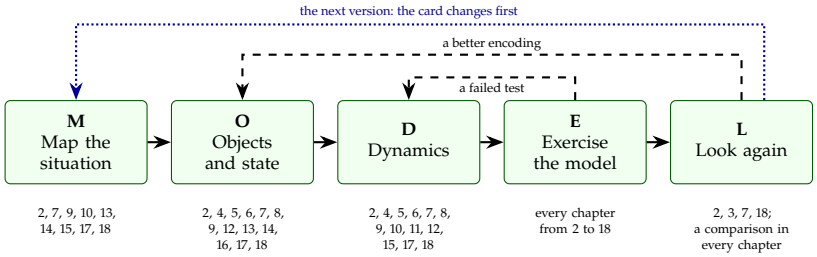


Figure 19.1.: The five steps, the two loops of Chapter 1 (dashed) and the chapters that added questions to each step. The dotted arrow is the loop that this chapter adds: a model grows in versions, and each version starts again at step M.

Three things are worth reading in the figure. Step D and step O carry most of the language: almost every feature of PDDL is a new way to describe a property or a change. Step E grew in every chapter, because every feature brought its own way of being wrong. And step M, which writes no PDDL, was touched by nine chapters: metrics, time, calendars, wishes, rules and a world that moves by itself are all decided on the model card, before they are features.

The criteria of step L are the same five, in the same order (Chapter 1). What the book added is how to put numbers on them, and Table 19.2 collects it.

Criterion	How it is judged
Fidelity	No impossible plans: the cheat tests, and wrong plans written by hand that `kairos validate` rejects (Chapter 2). No lost solutions: reasonable plans written by hand that it accepts (Chapter 11)
Size	The atoms and ground actions of the `[ground]` line of `kairos -v`, and how they grow with the problem (Chapter 3)
Guidance	States expanded until the first plan and until the last; the first plan against the best; the state variables and mutex groups of the `[sas]` line (Chapter 3, Chapter 7); the engine that the planner had to use (Chapter 11)
Clarity	Whether the file reads like the story, and whether the plan does; how many places change when a rule changes
Portability	The requirements of the file against the table of planners of Appendix B

Table 19.2.: The five criteria of step L and how the book measures them.

The method is five steps and two loops. This chapter adds a third loop: a model grows in versions, and each version starts again at the model card.

19.2. The five checklists

A **checklist** is a list of questions to be answered in writing, in order, every time. The five that follow hold every question that Chapters 2 to 18 added to the method, with the chapter where the question was first asked and where its answer is explained. They are meant to lie next to the keyboard. Nobody needs every question for every model: a classical model skips most of them with a quick "no", and each "no" is a decision that the model card should record under *Scope*.

Checklist M: Map the situation (the model card)

- **Story.** Can I write the five lines of the card before any PDDL? Which rules of the world does the card state (Chapter 2)?
- **Agents.** Who acts, and who is only part of the scene (Chapter 7)? Who can act at the same time as whom (Chapter 10)?
- **Success.** What counts as solved? Which goals are required and which are wishes, what is each wish worth, and in which unit (Chapter 14)? What is the plan when nothing is granted (Chapter 14)?
- **Quality.** What makes one plan better than another, in which unit? Is it one number, and if not, who sets the weights (Chapter 9)? Does it matter *when* the goal is reached (Chapter 10)?
- **Scope.** What is left out, and who takes care of it (Chapter 7)?
- **Assumptions.** Which of the classical assumptions hold (Chapter 1, Chapter 7)? Does the world change at known times, whatever the agent does (Chapter 13)? Does it change when the agent does nothing; does waiting cost or achieve something (Chapter 17)?
- **Rules.** Listed apart from the goals. For each: is it about one action or about the whole plan; does it look at one state or at history; whose rule is it, and can it bend (Chapter 15)? Is there something that must never happen, at any instant, and is it forbidden or does it only have consequences (Chapter 18)?

Checklist O: Objects and state (the state table)

- **Kinds.** Which kinds of things are there? Is there a hierarchy? Does an object belong to every problem, and so is a constant (Chapter 4)?
- For each property:
  - Is it static or dynamic (Chapter 2)?
  - Is it yes or no, or does it have several values (Chapter 5)? How many values per object: exactly one, at most one, many (Chapter 16)? Every "one" written as a predicate is an invariant to test (Chapter 7).
  - Is it about absence, and does it need a fact of its own? Do two predicates overlap, and who keeps them in step (Chapter 2, Chapter 5)?
  - Can it be deduced from another property, and then need not be stored (Chapter 6)? Is it a consequence of others that a rule can state, by looking at a fixed number of facts or by following a chain (Chapter 13)?
  - Is it a quantity? Does only a threshold matter? Is it bounded, and what keeps it inside its bounds (Chapter 8)? Is it a measure of the metric, a counter that starts at zero (Chapter 9)?

  - Does it change while an action runs, and at what rate (Chapter 12)? Does it change by itself: which fact switches the change on and off, and which number measures how far it has gone (Chapter 17)? Is its rate a constant, another changing quantity, or the quantity itself (Chapter 18)?
- Is a wish or a rule about the way, with nothing in the state to record it? Then a fact must record it (Chapter 14).

Checklist D: Dynamics (the action table)

- **Who causes each change?** The agent, by a decision: an action. The world, while a condition holds: a process. The world, at the instant a condition becomes true: an event (Chapter 17). The test: imagine that the agent falls asleep.
- For each action:
  - When does it apply? Is a condition about absence, about any object, about every object; are there several ways to satisfy it (Chapter 5)? May two parameters be the same object (Chapter 4)? Is every parameter needed (Chapter 7)?
  - Does a precondition say what the action *needs*, or what somebody *wants* of the plan? The second is a rule (Chapter 15).
  - What does it change? For each dynamic property: made true, made false, or left alone; everything that changes and nothing that does not (Chapter 2). Does what it changes depend on the situation; does it change many objects at once (Chapter 6)?
  - For each quantity: does the action need some of it, does it change it, and what does it add to each counter, zeros included (Chapter 8, Chapter 9)?
  - Can it be undone? Is it one decision or several (Chapter 7)?
  - Does its duration matter, and what does it depend on? What must hold at the start, throughout and at the end? When does each effect happen? What may run in parallel with it (Chapter 10)?
  - Must it overlap with another action, and with how much slack? Is its duration a decision, and does anything depend on it (Chapter 11)?
  - Does anybody read a quantity while the action changes it; can another action change it at the same time (Chapter 12)?
- For each process: is everything that must be true for the change to go on in its condition? Which processes touch the same quantity, and what is the sum of their rates (Chapter 17, Chapter 18)?
- For each event: what makes its condition false once it has fired? Which events can trigger which, and where does time pass in each cycle (Chapter 17, Chapter 18)?

Checklist E: Exercise the model (the test set)

- **Before running.** Estimate the ground actions and the reachable states, and compare with the `[ground]` line (Chapter 3). For a hybrid model, work out with a pencil where each quantity

goes if nothing is done (Chapter 18).
- **The routine.** `kairos check` with the domain and the problem, no warning unread; solve; read the plan on the state; `kairos validate` (Chapter 2).
- **The four tests.** A trivial problem, a typical one, one that must have no plan, one that tempts the planner to cheat. Know the expected length, cost or makespan before running (Chapter 2, Chapter 10).
- **The cheats, by hand.** Each shortcut as a plan file that the validator must reject, for the reason expected (Chapter 7).
- **One test for each thing the model promises:**
  - each pair of parameters that must differ (Chapter 4); each state defined by absence; each goal that could be true for a silly reason (Chapter 5);
  - each conditional effect, with its condition true and false (Chapter 6);
  - each quantity: a problem that needs more than there is, a goal on the far side of a bound (Chapter 8); two plans of known cost (Chapter 9);
  - each agent in two places at once; each overlap in a plan read as a question (Chapter 10); each envelope with a content that does not fit (Chapter 11); each limit, with a short plan that breaks it half way (Chapter 12); each window, with an action longer than the window (Chapter 13);
  - each wish: the plan with no weights, and the price test (Chapter 14); each rule: a plan that breaks it (Chapter 15);
  - each event fired at least once; each process: is there a stretch of time in which it runs for free, or fails to run (Chapter 17, Chapter 18)?
- **When the planner finds nothing.** Is the verdict a proof? If not, write the plan by hand and validate it; give the "no plan" tests a short horizon (Chapter 11, Chapter 18).
- **On failure,** the procedure of Section 19.6.

Checklist L: Look again (the comparison)

- Which other encodings exist? Every pattern has a rival (Section 19.4), and every feature has a cheaper neighbor (Section 19.3).
- Which is the lowest level (classical, numbers, durative actions, PDDL+) that does not lie about what the problem turns on (Chapter 18)?
- Compare two or three, written and run on the same problem, on Fidelity, Size, Guidance, Clarity and Portability, with numbers (Chapter 3).
- If the decision depends on a question that nobody asked, it belongs to step M: go back and ask it (Chapter 7).
- Write the decision and its reason where the next reader will find it: in the model card.

Five checklists hold every question of the book. A quick "no" to a question is also an answer, and it goes into the Scope line of the card.

19.3. Which feature for which need

A feature of PDDL is the answer to a need of the problem, and the checklists find the needs. This section goes the other way: from the

need to the feature, the requirement that declares it and the chapter that teaches it, and then to what the feature costs.

Figure 19.2 is the guide as a flowchart. It is read from the top, and it has no exit on the right: a "yes" adds a feature to the model and the reading goes on with the next question. The order is the order of the book and also the order of cost. The advice of Chapter 18 for hybrid systems holds for the whole language: *stop at the lowest level that does not lie about what the problem turns on*.

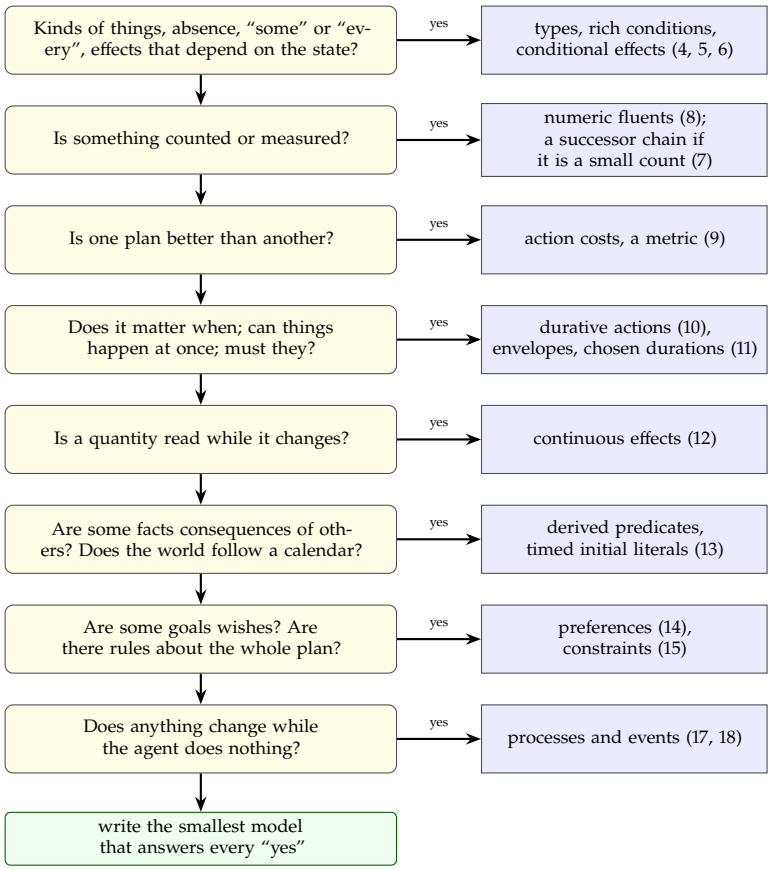


Figure 19.2.: The decision guide. Each question is asked of the model card; each "yes" adds the feature on its right (chapter numbers in parentheses) and the reading continues downwards. The lower a feature is, the fewer planners accept it and the harder the search.

Tables 19.3 and 19.4 are the same guide in detail: one row for each need that the book has met. The requirement `:adl` declares at once all those of Table 19.3 above the line (Chapter 6).

The problem needs	Feature	Requirement	Ch.
kinds of things, some treated alike	types, a hierarchy, `either`	`:typing`	4
an object that the rules name	a constant	none	4
two parameters that must differ	`(not (= ?a ?b))`	`:equality`	4
a condition about absence	`not`	`:negative-preconditions`	5
several ways to satisfy a condition	`or`, `imply`	`:disjunctive-preconditions`	5
a condition about some object	`exists`	`:existential-preconditions`	5
a condition or a goal about every object	`forall`	`:universal-preconditions`	5
an outcome that depends on the state; a change to many objects	`when`, `forall` in effects	`:conditional-effects`	6
a quantity that is added, spent, compared	functions, `increase`, `decrease`, `assign`	`:numeric-fluents`	8
a better plan, as a sum of fixed costs	`(total-cost)`, `:metric`	`:action-costs`	9
a better plan, by a measure that depends on the state	`:metric` over fluents	`:numeric-fluents`	9
a property with exactly one value	an object fluent	`:object-fluents`	16

Table 19.3.: From need to feature: the state, the conditions and the effects.

The problem needs	Feature	Requirement	Ch.
actions that last and may overlap; the finishing time	durative actions, `(total-time)`	`:durative-actions`	10
something that holds only while an action runs	an envelope (required concurrency)	no flag of its own	11
a duration that is a decision	bounds on `?duration`	`:duration-inequalities`	11
a quantity that somebody reads while it changes	`#t` in a durative action	`:continuous-effects`	12
a fact that follows from others, at any depth	`:derived`	`:derived-predicates`	13
a world with a calendar: hours, windows, deadlines	`(at t (fact))` in `:init`	`:timed-initial-literals`	13
numbers that change at known times	`(at t (= (f) v))`	`:timed-initial-fluents`	13
goals that are wishes, with a price	`preference`, `is-violated`	`:preferences`	14
a rule about the whole plan	`always`, `sometime`, `within` and the other operators	`:constraints`	15
change while a condition holds, whoever acts	`:process`	`:time`	17
change at the instant a condition becomes true	`:event`	`:time`	17
a condition that no instant of any plan may break	`:constraint` in the domain	`:time`	18

Table 19.4.: From need to feature: time, preferences and constraints, and the world that changes by itself.

What each feature costs

No feature is free. Table 19.5 gives, for each family, what it buys in Fidelity and what it charges on the other four criteria. The column Portability counts, among the eight planners that the documentation of Kairos compares (Martín Rico et al., 2026) (the table of Appendix B), those that returned a valid plan for one small problem with the feature, with ten

seconds each. A count says that a planner reads the construct and plans with it, not how well it does on large problems.

Table 19.5.: What each feature buys and what it costs. Portability: planners, out of eight, that returned a valid plan for a small problem with the feature.

Feature	Buys (Fidelity)	Charges (Size, Guidance, Clarity)	Portability
Types, constants, equality	mixed kinds become errors of `check`	nothing: the same ground problem	7 of 8; `either` 5
Negation, `or`	absence and alternatives said directly	`or` may duplicate ground actions	5
`forall`, `exists`	rules that follow the objects	expanded over the objects of the type	5
Conditional effects	one action for what is one thing in the world	compiled into more actions or longer plans by planners that lack them	3
Derived predicates	consequences cannot go out of step	rules are grounded and evaluated in every state	3
Numeric fluents	amounts as data	the state space is no longer finite	5
Action costs; fluent metrics	which plan is better	the best plan is found late and proved later	4; 4
Durative actions	overlap, makespan	search over timetables; plans are valid, not the shortest	5
Required concurrency	what exists only while an action runs	search over starts and ends; "no plan found" without proof	4
Chosen durations	"for as long as needed"	durations come at their bounds	3
Continuous effects	a number read while it moves	instants become decisions	2
Timed literals	the calendar of the world	actions must fit in windows	3
Preferences	over-subscription, wishes with a price	an optimization problem: the first plan may be empty	4
Constraints	rules stated once, about every state	a monitor in every state of the search	2
Object fluents	single-valued properties cannot break	nothing: planners build these variables anyway	1
Processes, events	a world that does not wait	search in steps of time; plans replayed to be read	1

Two notes on the last column. The planners that are missing are not always the same: POPF and OPTIC, two temporal planners, have no negative preconditions, disjunctions or conditional effects (Chapter 5, Chapter 6), and they need numeric conditions and effects inside `(and ...)` (Chapter 8); LAMA, a classical planner, has no numeric fluents. So the portable version of a temporal model is not the portable version of a classical one. And a feature of a low row is often worth trading for a pattern of a high row: complementary predicates for negation, a successor chain for a small number, blocks for a chosen duration, an automaton for a constraint. The next section is the index of those trades.

A feature answers a need and charges for it. Add features in the order of the flowchart, and stop when the model no longer lies about what the problem turns on.

19.4. The catalog of patterns

Chapter 7 defined a **pattern** as a named way of encoding something that many problems have in common, and gave eleven of them for classical planning. The chapters before and after it named others as they met them. This section is the index of all of them, under the names that their chapters use, each with the need that it answers and its rival: the

other encoding that step L should consider. The definitions, the tested examples and the costs are in the chapters; here is where to look.

Pattern	Use it when	Rival	Ch.
Static table	a rule may differ between problems	the rule written in an action	7
Mirrored facts	a relation holds both ways	one fact and an `or` in the action	7
At-most-one invariant	a property has one value	an object fluent	7, 16
Successor chain	a small quantity; only the amount matters	slots; a numeric fluent	7, 8
Slots	units of capacity are distinct places	a successor chain	7
Phase token	stages must not mix	a precondition on what is already done	7
Turn token	agents, or the agent and the world, alternate	actions that take one step of each	7
Resource lock	one user at a time	an envelope, when time is modeled	7, 11
Inverse pair	the world can undo it	a one-way action	7
One-way action	the world cannot undo it	an inverse pair	7
Merged action	no decision between the steps	separate actions	7
Type predicate	a kind that changes, or that crosses the hierarchy	a type	2, 4
Complementary predicate	absence without `not`; three or more named values	a negative precondition	5
One action per case	few fixed cases; the plan should name the case	a conditional effect	6
Bookkeeping action	many cases and a planner without conditional effects	a conditional effect	6
Deduced, not stored	a fact can be deduced from another	a stored fact kept by effects, or a derived predicate	6, 13

Table 19.6.: The patterns of classical planning: the catalog of Chapter 7 (above the line) and the encodings that Chapters 2 to 6 compared (below it).

Table 19.7.: The patterns of numbers, time, preferences and constraints, and hybrid systems.

Pattern	Use it when	Rival	Ch.
Bounded quantity	an amount is spent and restored: a precondition before every decrease, a cap on every increase	a threshold fact; a successor chain	8
Counter fluent	a measure of the metric: starts at zero, read by no precondition	action costs	9
Hard limit	something must not be exceeded	a term of the metric	9
Weighted sum	several measures, one metric	a limit on all but one	9
Token	an agent or a tool does one thing at a time: deleted at start, added at end	none: without it actions overlap	10
Invariant	something must hold all the way: `over all`	a condition at start, when an instant is enough	10
Take at start, give at end	every effect of a durative action	the exception is the envelope	10
Envelope and contents	a thing exists only while an action runs, and ends by itself	a lock, closed by the goal	11
Slack	two actions must overlap: one contains the other with room to spare	exact simultaneity, which planners do not search for	11
Chosen duration	"for how long?" is a decision	a fixed duration; blocks	11
Blocks	a chosen duration on a planner without one	a chosen duration	11
Careful discrete model	a quantity changes during an action and nobody reads it meanwhile: consume at start, produce at end	a continuous effect	12
Derived predicate for a twin	a redundant fact used in many actions	a twin kept by effects	13
Window	a fact true between two timed literals, needed `over all`	a `hold-during` constraint	13, 15
Deadline	a helper fact that a timed literal deletes, needed `at end`	a `within` constraint	13, 15
Soft goal	more goals than can be met	a hard goal	14
Preference in a precondition	a wish about the way, paid each time	a hard precondition; an action cost	14
Constraint first	any rule about the whole plan: the constraint is the specification	preconditions on every action; a monitor	15
Monitor	a rule about history on a planner without constraints: an automaton in extra predicates	a constraint	15
Object fluent	exactly one value, or at most one	a predicate with an invariant	16
Start-process-stop	an activity whose end depends on how the world evolves	a durative action	17
Event as device	a part of the system reacts by itself: a thermostat, a float switch	an action of the agent	18
Hysteresis	an event switches something on and off around a threshold	one threshold, which never settles	18
Global constraint	a failure is forbidden in every plan	an event that marks it and a goal that refuses the mark	18

The chapters after Chapter 7 did not always present their encodings as patterns; the second table uses the words of each chapter. The two tables are the vocabulary of steps O and D. When a row of the state table is being written, the question is which pattern it is: a static table, an at-most-one invariant, a bounded quantity. When an action is being written: a token, an inverse pair, an envelope. And at step L the third column of the tables is the list of alternatives that deserve a column of the comparison.

Forty named encodings, each with a rival. The name says what a row of the state table or of the action table is; the rival is what step L compares it with.

19.5. The catalog of anti-patterns

An **anti-pattern** is a mistake common enough to deserve a name (Chapter 7). The ones that matter most have one thing in common: the file is valid PDDL and `kairos check` says `valid, plannable`. They are recognized by what the planner does, so the tables that follow are ordered by symptom. Mistakes of form (a misspelled name, a missing parenthesis, a requirement left out) are not here: `check` reports them, and the rule for them is the one of Chapter 2, no warning unread.

Table 19.8.: Anti-patterns of classical models, by symptom.

Symptom	Anti-pattern	Ch.
A plan that is too short; something is in two places; fewer mutex groups than properties with one value	Forgotten delete	2, 7
A plan that does what the card forbids	A missing precondition	2
A plan of cost 0; a plan that skips part of the work	Goal in the initial state	2, 7
The trivial test passes, the typical one has no plan	Out-of-step twins	5, 7
Far more ground actions than estimated; arguments that make no sense	Idle parameter	3, 7
An object acts on itself, or plays two roles	Trusting that different variables are different objects	4
A goal with “every” or “not” holds for a silly reason: no objects, an item in no room	A `forall` that is true of nothing	5
A plan that does harm on the way and still reaches the goal	A harm that the goal does not watch	6
A plan without the actions that pass the token	Two tokens	7
Plans longer than expected; a count that never passes some value	A chain with a broken link	7
A robot that enters a room and cannot leave	Half of the mirrored facts	7

Symptom	Anti-pattern	Ch.
Plans that are too good; a quantity below zero in the trace	The consumption without the precondition	8
No plan after a handful of states; a plan that avoids something for no visible reason	A forgotten initial value	8, 16
No plan for a problem that a child solves, with a goal `(= ...)` on a computed number	Equality on fractions	8
One action used as much as possible; a cost lower than computed by hand	An action that forgot its cost	9
An agent starts something and leaves; a makespan shorter than the sum expected	Checking at the start what must hold all the way	10
One action overlaps all the others, or the agent never acts again	A token forgotten in one action	10
A model that was solved at once stops being solved; `-v` reports engine M3 for an action nobody thought of as an envelope	An envelope by accident	11
One object holds the door and walks through it	The agent of the envelope is also its content	11
A charge of the shortest duration allowed, or past the capacity	A chosen duration that nothing depends on	11
A plan that the validator accepts while a number crosses its limit half way	A limit checked only at the start	12
Works on short maps, fails on the first long corridor	A rule that only looks one step ahead	13
An action before the opening time	The fact is already true	13
An action that ends inside a closed window	A window checked only at the start	13

Table 19.9.: Anti-patterns of models with numbers and time, by symptom.

Symptom	Anti-pattern	Ch.
An empty plan of cost 0 for a problem full of wishes	Preferences without a metric	14
A plan that does less than expected and is `proven optimal`	Weights in the wrong unit	14
A rule of the story broken whenever that is cheaper	A preference in a precondition is not a precondition	14
A constraint broken before the first action	Forgetting that the initial state counts	15
A plan that ends before the window of a constraint opens	A window that the plan never reaches	15
A deadline that restricts nothing	Steps taken for seconds	15
`unsolvable` on a problem that looked easy	A rule stronger than the world	15
The robot starts nowhere; `unsolvable` after one state	Functions are not predicates	16
A plan for a problem that should have none; things switched on and off at odd moments	A process that runs when it should not	17
No plan when the plan must pass through an event; the validator says `events keep firing`	An event that does not switch itself off; two events that switch each other on	17, 18
A slow planner, and tools that disagree	A thermostat with one threshold	18
`no plan found` after few states, and a plan by hand that is valid	A window narrower than a step	18
A plan to the millisecond that the validator rejects by 0.001	Equality on a quantity that flows	18
A valid plan that waits and charges more than needed	Expecting the shortest plan	18
A rescue that the model forbids, or a limit that one problem forgets	A constraint where an event was needed, and the reverse	18

Table 19.10.: Anti-patterns of models with preferences, constraints, object fluents, processes and events, by symptom.

Read down the first column, the symptoms fall into four families, and the family already says where to look.

- **Too easy**: a plan that is too short, too cheap or too early. The model allows too much: a missing precondition, delete, token, invariant or cost.

- **Too hard**: no plan, or a plan with strange detours. The model allows too little: a fact that nobody produces, a value that nobody gave, a rule that is too strong.
- **Too big**: more ground actions or states than estimated, a search that does not end. A parameter, a symmetry or a horizon is out of control.
- **Not what was asked**: the plan is right for the model, and the model answers another question. The metric, the weights or the goal say something that the card did not.

Anti-patterns pass the check and are caught by their symptom: a plan too easy, too hard, too big, or right for the wrong question.

19.6. Debugging, beyond classical

The procedure of Chapter 7 had two branches. No plan: read the note of the planner, relax the goal one conjunct at a time, remove preconditions until a plan appears, and ask which action should have made the missing fact true. Wrong plan: find the first step that the world refuses, then the precondition or the delete that is missing. Both branches stay as they are. Numbers, time and processes add three things: more verdicts, more ways for a plan to be wrong, and a planner that can miss plans that exist.

Read the verdict first

In classical planning a run ended with a plan or with `unsolvable`. Table 19.11 lists the verdicts that the book has met since, each with a file of this chapter that produces it. Only one of them is a proof that no plan exists.

Table 19.11.: The verdicts of Kairos and their exit codes. The scripts that run a test set (`tests.txt`) state the code that each test expects.

Verdict	Code	What it says, and an example
a plan	0	The plan is valid for the model. `solved` is not `proven optimal` (Chapter 9)
`unsolvable`	10	A proof: no plan exists (Chapter 3). `laundry-v1-p03`
`no plan found`	14	The search ended without a plan and without a proof (Chapter 11, Chapter 12, Chapter 17). `laundry-v5-p03`
`timeout`	11	Nothing: the search was still running (Chapter 3). `causeway-p04`, among the solutions
`invalid`	2	The files do not say what was meant. Read the first error (Chapter 1)

Figure 19.3 is the whole procedure. The gray boxes are those of Chapter 7; the others are new.

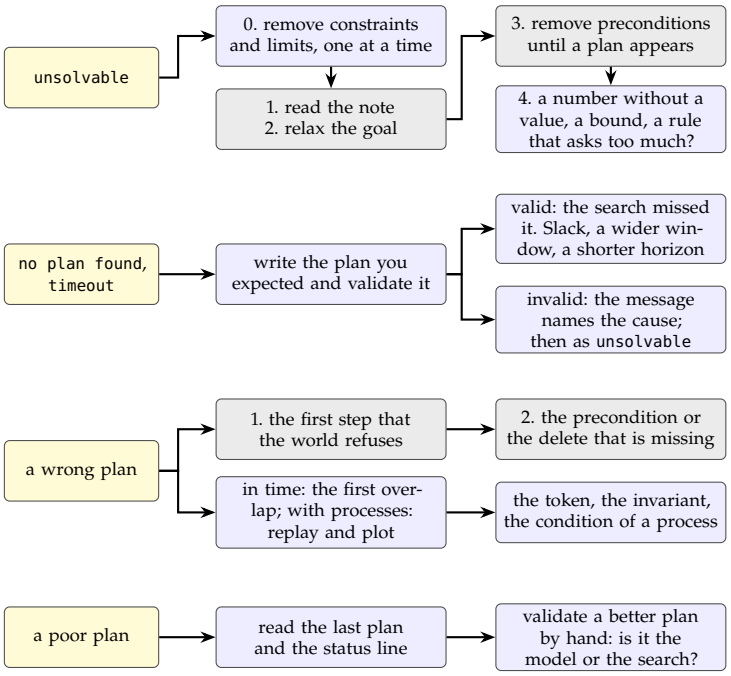


Figure 19.3.: The debugging procedure of step E, extended. Gray: the steps of Chapter 7. Blue: what numbers, time, constraints and processes add.

No plan, with a proof

The verdict `unsolvable` is handled as in Chapter 7, with one step before and one after.

0. Remove what cuts plans away. A constraint, a hard limit on a budget, a deadline and a global constraint do not take part in any action: they remove whole plans. Take them out one at a time, in a copy of the problem, until a plan appears. Then validate that plan against each of them alone, to see which one it breaks and where (Chapter 15). Steps 1 to 3 follow, on the model without them.

4. Ask the numbers. When the precondition whose removal brings a plan back is numeric, the fact "that nobody makes true" is a value. There are three usual causes. A fluent that an action reads was never given a value, so the action never applies: the sign is a search of a handful of states, and the fix is a line in `:init` (Chapter 8; the same for an object fluent, Chapter 16). A bound is tighter than the task: add up by hand what the plan needs and compare it with what there is. Or a test asks for exact equality on a number that is computed (Chapter 8) or that flows (Chapter 18).

No plan, without a proof

`no plan found` and `timeout` do not say that the model has no plan. Before changing the model, ask the validator, which has no search to get lost in: write in a file the plan that you expected and run `kairos validate`.

If the plan is *invalid*, the message names the condition that fails and the instant. That is step 3 of the classical procedure done for you: the model

does have no plan of that shape, and the question is again which action should have made the condition true.

If the plan is *valid*, the model is right and the search could not find it. The chapters of the book report four situations in which that happens with Kairos 1.0.6, and for each of them a way to model around it. The box collects them.

What the planner of this book does not find

- **A duration between its bounds.** A duration that the planner chooses comes out at one of its bounds, or at the instant where ending the action makes a condition true. A duration in between that no condition fixes is not tried: the courier of Chapter 12 charges for 8 seconds where 6 were enough. *Model around it*: make something need the duration, or use blocks (Chapter 11).
- **The same envelope used twice.** Plans that repeat one envelope action, with a content inside each repetition, are among those that the search over starts and ends misses (Chapter 11, in the solutions). *Model around it*: read the plan, write the better one by hand and validate it.
- **Two starts at exactly the same instant.** Happenings are searched one after another, so a model whose only plans start two actions in the same instant gets `no plan found` (Chapter 11). *Model around it*: slack. One action contains the other.
- **What happens between two steps.** With processes, time advances in steps of 0.1, and events fire between steps. A window narrower than what changes in one step is walked past, and an event does not rescue the search (Chapter 17, Chapter 18). *Model around it*: a window at least as wide as the rate times 0.1; inequalities, not equality.

And two that concern good plans rather than plans. A temporal plan is a valid timetable, not the shortest one (Chapter 11). In PDDL+ the search minimizes the number of actions, whatever the metric says, so plans wait and charge more than needed (Chapter 18). In every case the plan that Kairos prints is valid. What may be missing is a plan, or a better one.

A search that runs out of time on a problem that should have no plan is usually a horizon that nobody bounded (Chapter 18): things can keep changing for a long time, and every instant is a state. Give the "no plan" tests a small margin. Section 19.10 shows one that takes 0.1 seconds and its careless version, which uses all the time it is given.

A wrong plan, and a poor one

A plan is wrong when the world would not allow it, and the first branch of Chapter 7 holds: read it against the story and stop at the first thing that the world refuses. What changes is what a "thing" is.

- **In a sequential plan**, a step. Its cause is a missing precondition or an earlier action that left a wrong fact (Chapter 7); with numbers, a decrease without its precondition (Chapter 8).
- **In a temporal plan**, an overlap. Draw the timeline, one row per agent and per resource, and ask of every pair of bars that overlap whether the world allows it (Chapter 10). The cause is a token that

one action does not take, a condition `at start` that should be `over all`, or an effect at the wrong end.

- **In a plan with processes**, a stretch of time. Replay the plan, compute the quantities and plot them (Chapter 18). The cause is a process whose condition is too weak, so that something changes for free, or an event that should have fired.

A plan is poor when it is valid and a better one exists. First make sure that it is the last plan of the run and read why the search stopped (Chapter 9). Then write the better plan by hand and validate it against the same problem: the validator prints its makespan and the value of the metric. If the better plan is valid and cheaper, the model is right and the search stopped short; say so in the documentation of the model, and if it matters, put the pressure in the model (a deadline, a tighter bound). If the better plan is invalid, the model does not allow what you thought it allowed, and that is the finding. The laundry of the next section meets both cases.

Three verdicts are not proofs. When the planner finds nothing, or nothing good, write the plan you expected and ask the validator: it tells whether the model or the search is to blame.

19.7. Growing a model in versions

Nobody should write a PDDL+ model with preferences in one sitting. The reason is the debugging procedure that we have just read: it works by changing one thing and looking at what happens. A model written all at once has a hundred things that changed at once. Chapter 10 advised to get the logic right with instantaneous actions before adding time, and Chapter 18 called its three levels "also a path". This section makes the path explicit (Figure 19.4).

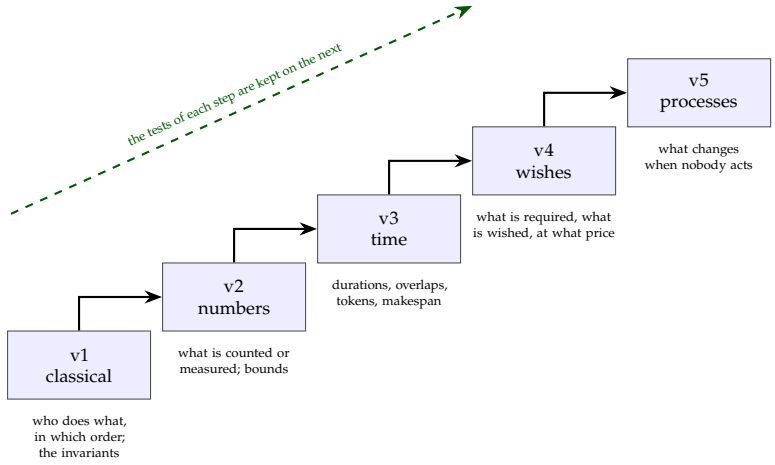


Figure 19.4.: A model grows one dimension at a time. Each step keeps the tests of the steps below it, translated, and adds its own.

Rules for growing a model

1. **One dimension per version.** Classical first; then numbers; then time; then preferences and constraints; then processes. A version that adds two is two versions.
2. **Green before the jump.** A version is finished when its whole test set gives the expected verdicts. Only then does the next one start.
3. **The card changes first.** Every jump moves something from "out" to "in" on the Scope line, or adds an entry (a metric, a wish, a schedule). Write that, then the tables, then the PDDL.
4. **Old tests are translated and kept.** The problems of the previous version, in the new syntax, with the same verdicts. A verdict

that changes must be explained by a line of the card that changed.
5. **New tests for the new dimension.** At least one "no plan" and one cheat that the previous version could not even state.
6. **Every version is a file.** Old versions are not deleted: they are the cheap models that answer the simple questions, and the reference when a new version misbehaves.
7. **Stop** when the model no longer lies about what the problem turns on.

The domain of the example is new to the book. The story:

> *Saturday morning at the laundromat. There are loads of clothes to wash: whites, colors, towels, some days a woolen sweater. Each load is washed, then dried, then folded. The label of each garment says which machines it may go in; wool must not be tumble-dried. There is one washer and one dryer.*

Version 1: classical

Step M: the model card of the laundry, version 1

Story	One person washes, dries and folds loads of clothes.
Agents	The person. The machines are tools: they decide nothing.
Success	Every load of the goal is folded.
Scope	In: whether a load is dirty, wet, folded; where it is (the basket or a machine); which machines its label allows. Out: time, kilograms, coins, detergent and programs, the clothesline.
Assumptions	The classical ones. A machine holds one load. A cycle never fails.

Steps O and D take a few minutes with the checklists and the catalog. Table 19.12 has both tables, with the pattern behind each row.

Property	About	Kind, pattern
`(dirty ?l) (wet ?l)`	the state of a load: dirty; clean and wet; clean and dry when neither holds	dynamic; two facts, three states
`(in-basket ?l) (in ?l ?m)`	where a load is	dynamic; at-most-one invariant
`(empty ?m)`	the machine holds no load	dynamic; twin of `in`, resource lock
`(allowed ?l ?m)`	the label of the load admits the machine	static table
`(folded ?l)`	the load is done	dynamic
Action	**Applies when**	**Changes**
`put ?l ?m`	the load is in the basket, the machine empty and allowed	the load is in the machine; inverse pair with `take-out`
`wash ?l ?w`	the load is in the washer and dirty	clean and wet; one-way
`dry ?l ?d`	the load is in the dryer and wet	dry; one-way
`fold ?l`	the load is in the basket, neither dirty nor wet	folded; one-way

Table 19.12.: The state table and the action table of the laundry, version 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)))
```

Listing 19.1: The laundry, version 1 (`laundry-v1-domain.pddl`).

The test set has the four problems of the method. The trivial one is one load of towels, and its plan can be written before running: seven steps.

```
$ 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)
```

The typical problem, three loads, gives 21 steps, proved optimal. The problem that must have no plan is the woolen sweater: the label does not allow the dryer, and this version has no other way to dry.

```
$ 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)
```

Two shortcuts tempt the planner, and both are written by hand as plan files: washing two loads at once, and folding the towels as they leave the washer.

```
$ 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
```

Version 1 is green. It already answers a real question, in which order to do things, and it has fixed what every later version will rely on: a load is in one place, a clean load is never dirty again, only clean and dry clothes are folded.

Version 2: numbers

The first cheat test bothers anyone who has used a washing machine: of course two loads go in one wash, if they fit. The card changes. *Scope*: kilograms are in. *Assumptions*: a machine holds loads up to its capacity, and one cycle treats everything that is inside.

Checklist O asks its three questions of the new property (Chapter 8). Is it a quantity? Yes: weights add up. Does only a threshold matter? No: different loads take different amounts. Is it bounded? Between zero and the capacity of the drum, and the precondition of `put` keeps it there. The twin `empty` disappears: the room left in the drum, a number, takes its place. Checklist D asks whether an action changes many objects at once: a cycle now washes whatever is in the drum, which is a universal conditional effect (Chapter 6). One jump in the model pulled a feature of another chapter with it, and that is 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)))))
```

Listing 19.2: The laundry, version 2: capacity in kilograms and cycles that treat the whole drum (part of `laundry-v2-domain.pddl`).

The old tests are translated: the same loads, now with weights, in drums of 8 kg. The trivial one still has seven steps. The typical one has loads of 3, 4 and 5 kg, so two of them fit in one cycle and the three do not:

```
$ 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)
...
```

Nineteen steps: two washes and two drying cycles where version 1 needed three of each. The new tests are the two that Chapter 8 asks for each quantity. A problem that needs more than there is: a duvet of 9 kg, which is `unsolvable`. And the cheat on the far side of the bound, the three loads in one wash:

```
$ 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
```

One verdict changed between the versions: the cheat of version 1, two loads in one drum, is now a legal plan. Rule 4 asks for the line of the card that explains it, and it is there. What was a rule of the model ("a machine holds one load") was replaced by the rule of the world.

Version 3: time

A plan of 19 steps says nothing about the morning. The washer takes half an hour and works alone; a person who waits in front of it wastes that half hour, and a person who reads the plan of version 2 does exactly that. The card changes. *Success*: every load folded, *as early as possible*. *Agents*: the person does one thing at a time; the machines run by themselves once started. *Scope*: durations are in, in minutes.

The four columns that Chapter 10 adds to the action table are filled in Table 19.13.

Action	Duration	Must hold	May overlap with
`put, take-out`	1	the machine is stopped, throughout	anything that does not need the hands
`wash, dry`	the cycle of the machine	stopped at the start; something in the drum	everything, except opening its own door
`fold`	5	clean and dry at the start	the cycles

Table 19.13.: The laundry, version 3: the columns of time in the action table.

The table names two tokens (Chapter 10): `(hands-free)` for the person and `(stopped ?m)` for each machine, both taken at the start and given back at the end. Loading and unloading need the machine stopped *throughout*, an invariant, so that nobody opens the door of a running machine.

```
(: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))
                   (at end (and (not (dirty ?l)) (wet ?l)))))))
```

Listing 19.3: The laundry, version 3: two of its durative actions (part of `laundry-v3-domain.pddl`).

With the condition written `at start` and the effect `at end`, Kairos 1.0.6 plans the domain with its engine M2 only, which does not overlap actions (Chapter 11); forced to use M3 it answers that it "cannot check them yet". Written as in the listing, both engines apply.

One line of the listing needs a comment. The conditional effect of `wash` reads what is in the drum *at the end*. Since the door is shut during the cycle, that is what was in the drum at the start, and the condition could be written either way.

The old tests are translated again, and the first result of the typical one is a lesson. The file `laundry-v3-idle-domain.pddl` is the first version 3 that we wrote. Its plan for the three loads has the right makespan and one line too many:

```
$ 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]
```

At minute 90 the washer runs a cycle with nothing in it. In version 2 the planner never did that, because every action made the plan longer. Now the measure is the makespan, the empty cycle ends before the plan

does, and it costs nothing. The model had always allowed it; the jump changed what the planner is indifferent to. This is the branch "a wrong plan" of the debugging procedure: no rule of the card is broken, but nobody pays for a cycle of an empty machine, and the first step that the world refuses is that wash. The cause is a missing precondition: a cycle needs something in the drum. Version 3 as printed in Listing 19.3 has it, and gives:

```
$ 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]
```

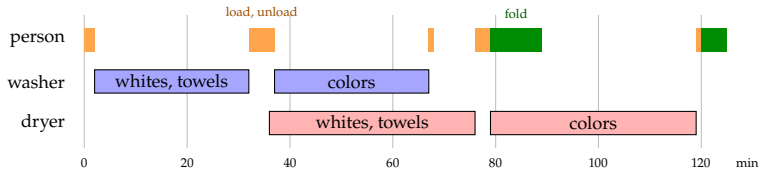


Figure 19.5.: The plan of version 3 for the three loads, as a timeline. The second wash runs while the dryer works on the first batch, and the person folds during the second drying cycle.

The timeline of Figure 19.5 is how a temporal plan is read (Chapter 10). The dryer is the bottleneck: two cycles of 40 minutes that cannot overlap, after a wash of 30, with a few minutes of handling: no timetable can be much shorter than 125. Knowing the honest makespan before trusting the planner is the first cheat test of Chapter 10. The other two are written by hand, each as a plan that breaks one token: opening the washer ten minutes into its cycle, and loading two things with the same hands.

```
$ 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
```

Version 4: wishes

The plan of 125 minutes is the best one for a person who has all morning. Most people do not, and would rather leave the towels for another day

than stay for them. The card changes where Chapter 14 said it would: *Success* splits into what is required and what is wished, and each wish gets a price in the unit of the metric, minutes.

Step M, version 4: the wishes

Name	Condition	Weight	Reason
(required)	the whites are folded		the shirt for Monday
`colors-done`	the colors are folded	60	worth an hour of the morning
`towels-done`	the towels are folded	30	there are clean towels at home

Quality: the minutes until the last action ends, plus the weight of every wish given up. When nothing is granted, the plan washes the whites alone.

Steps O and D do not change: a preference uses the predicates and the actions that are there (Chapter 14). The domain gains one word, `:preferences`, in its requirements, and the whole jump is in the problem file:

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

Listing 19.4: The laundry, version 4: the goal and the metric of `laundry-v4-p02.pddl`.

```
$ 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]
```

The first plan washes the whites alone: 79 minutes, and 90 for the two wishes given up. The second takes the colors along in the same drum, 9 minutes more for a wish worth 60, and leaves the towels: they would need a second batch, 37 minutes more (Figure 19.5) for a wish worth 30. The plan is the one a sensible person would choose.

The tests of Chapter 14 are two. With nothing that can be granted (a duvet of 9 kg as a wish, `p01`), the plan is empty and costs the weight. And the price test: change one weight until the plan changes, and check that it changes where it should. In `p04` the colors are worth only 5 minutes, less than the 9 that they add, and Kairos drops them; the towels, 5 kg, take their place next to the whites in the drum, for a cost of 93.010. The weight at which the colors enter the plan is between 5 and 60, and by the arithmetic it is 9.

The price test of the towels is the second case that Section 19.6 announced. Their break-even point is 37 minutes. In `p03` they are worth 40, so the best plan should wash everything in 125.011. Kairos returns another:

```
$ 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
```

The file `laundry-v4-p03-all.plan` is the timetable of version 3, typed in. The validator accepts it for this problem with a metric of 125.011, better than the 128.010 of the plan returned. So the model is right, and the search stopped short: with `-v`, Kairos 1.0.6 shows that it planned this problem with its engine M2 alone, the one that puts whole actions one after another, and a batch of towels that does not overlap the first one costs 79 minutes, not 37. A poor plan, a better one by hand, the validator as judge: the last row of Figure 19.3. We write it down in the documentation of the model and go on.

Version 5: processes

One test has given the same verdict since version 1: the woolen sweater has no plan. The story has the answer hanging in the yard. *Scope*: the clothesline is in.

The first question of checklist D decides how. Imagine that the person falls asleep after hanging the sweater. It dries all the same, so drying on the line is not an action. It dries while it hangs and the sun shines: a process. It is dry at one instant, whether anybody looks or not: an event. And the sun follows a calendar that the person cannot change: timed initial literals (Chapter 13). Checklist O asks, for a quantity that changes by itself, which fact switches the change on and off and which number measures it (Chapter 17): the facts are `(hanging ?l)` and `(sunny)`, the number is the moisture of the load, 100 when it leaves the washer.

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

Listing 19.5: The laundry, version 5: an action
of the person, a process and an event (part of `laundry-v5-domain.pddl`).

Everything else stays as it was. Washing and drying in the dryer are still durative actions: their length is known when they start and nothing interrupts them, which is the advice of Chapter 17 and Chapter 18 for mixing the two. The wash now also sets the moisture to 100.

The old "no plan" test changes its verdict, and the card says why. With the sun out all morning (`p01`) the sweater is folded at minute 90: washed by 31, hung by 34, fifty minutes on the line at 2 units per minute. The typical test (`p02`) adds the whites and a cloud from minute 60 to minute 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
```

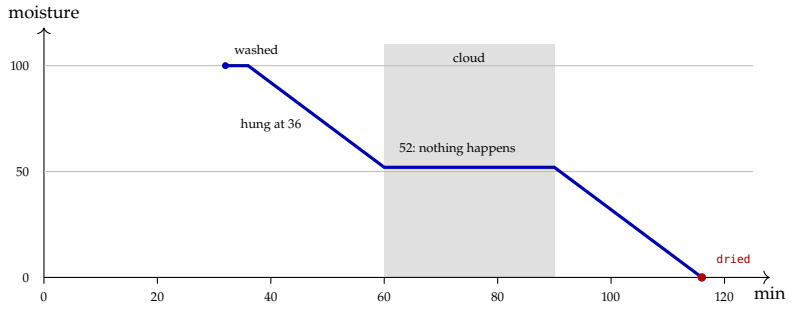


Figure 19.6.: The moisture of the wool along the plan for `laundry-v5-p02`. The process `sun-dry` runs from minute 36, stops under the cloud and resumes at 90; the event fires at 116, and only then can the load be collected and folded.

Replaying the plan (Figure 19.6) confirms the part that matters: the wool is collected at 116, when the event has fired, after 24 minutes of sun, 30 of cloud and 26 more of sun. It also shows the weakness that Chapter 18 warned about. The whites lie wet in the basket from minute 33 to 122 and go to the dryer only when the wool is folded: in PDDL+ the search minimizes the number of actions, and nothing rewards doing two things at once. The better plan is easy to write, and the validator accepts it:

```
$ kairos validate laundry-v5-domain.pddl laundry-v5-p02.pddl \
    laundry-v5-p02-hand.plan
plan valid, makespan 122, metric 122
```

The new tests are those of a hybrid model. The cheat that this version makes possible is a free ride (Chapter 17): clothes that dry in the shade. The plan `laundry-v5-p02-shade.plan` collects the wool after fifty minutes on the line, cloud or no cloud, and is rejected at the fold: `(wet wool) is true`. The "no plan" test is an overcast morning, in which `(sunny)` never holds:

```
$ 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)
```

The verdict is `no plan found`, exit code 14, as for every PDDL+ problem without a plan (Chapter 17). Here the pencil of Chapter 18 gives the proof that the planner cannot: without sun the moisture stays at 100. The wishes of version 4 still work on top of the processes: in `p04`, an overcast morning with the wool as a wish worth 30, the plan washes and dries the whites in 79 minutes and pays the 30.

Five versions, side by side

	Adds to the card	Typical test	Search
v1	order of the work; one load per machine	21 steps, optimal	808 states, 3 ms
v2	kilograms; a cycle treats the whole drum	19 steps, optimal	2 106 states, 12 ms
v3	durations; what may overlap; the makespan	125.011 min	101 696 states, 2.1 s
v4	required and wished; prices in minutes	cost 118.010	5 358 states, 20 ms
v5	the line, the sun, the cloud	169 min (122 by hand)	168 296 states, 2.0 s

Table 19.14.: The five versions of the laundry on their typical tests (`p02` of each), with the figures of `kairos -v` on one thread. The problems are not the same, so the last column compares orders of magnitude, not versions.

Table 19.14 is the trade of the whole book in five lines. Each version says something true that the previous one could not say, and each costs more to search and runs on fewer planners: by the counts of Table 19.5, the conditional effects of version 2 already leave three planners of eight, and the processes of version 5 leave one. The versions are not drafts of the last one. Which loads fit together is answered best by version 2, in milliseconds and with a proof; the timetable of an ordinary morning by version 3; version 5 is needed only on the days with wool.

Grow a model one dimension at a time: card first, old tests translated and kept, new tests for what is new. Each version stays as a file, and often as the best answer to a simpler question.

19.8. Documenting a model

A model is read more often than it is written: by the person who will write the problems, by the one who will connect it to a robot, by its author six months later. PDDL has one tool for them, the comment, and the method has five artifacts. Documenting a model means putting each artifact where its reader will look for it.

The header of the domain. What a reader of the file needs before the first parenthesis: what the model is, who decides what, what is left out, the units, the rules with the place where each is enforced, the invariants, what a problem must provide, where the tests are and how the model got here. Listing 19.6 is the header of the last version of the laundry, and a template for any domain.

```
;; 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.
```

Listing 19.6: A header comment as documentation: the first lines of `laundry-v5-domain.pddl`.

Each heading has a reason. *Decides* separates the actions of the agent from the processes and events of the world, which is the first thing that the user of a hybrid model must know. *Units* prevents the commonest silent mistake of numeric models, a weight in grams in one problem and in kilograms in the next. *Rules* names, for every rule of the card, the action or the predicate that enforces it, so that a rule without a place is seen at once. *Keeps* states the invariants that the tests check. *Problem* is the contract with whoever writes problem files: every function that needs a value, every fact that must be there (Chapter 8).

Inside the file, a comment belongs where a reader would stop: next to each predicate whose name does not say everything, next to each static function with its unit, above each action whose precondition encodes a rule. Comments that repeat the PDDL in words are noise.

The model card. The header is the short form. The card is the full one, and it is the document to show to the people who own the problem, who do not read PDDL. After eighteen chapters it has more than its five original lines.

The model card, complete: a template

Story	The situation in two or three sentences, in the words of its owner.
Agents	Who decides. Who and what only happens. Who can act at the same time.
Success	The required goals.
Quality	What makes a plan better, with its unit; the weights and their reason.
Wishes	A table: name, condition, weight, reason. The plan when none is granted.
Rules	A table: the sentence as it was said, its scope (one action or the whole plan), its strength (hard or soft), where it is encoded.
Scope	In, and out. For each thing left out, who takes care of it.
Assumptions	What is taken for granted about the world: what cannot fail, what changes at known times (the schedule), what changes by itself.
Level	Classical, numeric, temporal or PDDL+, the requirements, and the planners that the model was run on.
Decisions	The comparisons of step L: what was considered, what was chosen, why.
Known limits	What the model gets wrong on purpose, and what the planner does not find in it.
Versions	One line per version: what it added, and the question it answers best.

For the laundry, the entry *Known limits* has two lines that this section has earned: the plans of version 4 can be improved when a second batch could overlap the first, and the plans of version 5 do not hurry.

The tests. The state table and the action table live well as the comments of the domain. The test set needs a form that a machine can run. In this book it is the file `tests.txt` of each folder of examples, one line per test: what to do, with which files, and the exit code expected (Table 19.11). These are the lines for version 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
```

A test file of this kind is documentation that cannot go stale without somebody noticing: it says which problems the model is for, which it must refuse, and which plans it must reject. Every problem file starts with a comment that says which of the four tests it is and what is expected, and every plan written by hand says what it tries.

The header of the domain for who reads the file, the model card for who owns the problem, a runnable test file for whoever changes the model next.

19.9. The MODEL method: the whole method

This chapter collected the questions of the others. It adds a few of its own, which are not about any feature of the language: they are about the model as something that lives longer than an afternoon.

Questions that this chapter adds to the method

M	Which version is this, and what moved from "out" to "in" since the last one? Which question does this version answer that the previous one could not?
O, D	Which pattern is each row of the tables, by its name in the catalog?
E	Are the tests of the previous version translated, and do they keep their verdicts? Is each verdict a proof? Does every cheat fail for the reason expected?
L	Is there a feature in the model that a pattern of a higher row of the guide could replace? Is a lower version enough for the question that is being asked today?
All	Is each artifact written where its reader will look: the header, the card, the test file?

Alternatives compared

The jump to version 5 took a decision that step L must review: is the clothesline a process? Chapter 17 advises a durative action whenever an activity has a duration known at its start and nothing can interrupt it. So we wrote the line three ways, on top of version 3, and ran the three on the same morning: the wool, the whites, and the cloud from minute 60 to 90.

A. A durative action of fixed length (`laundry-line-fixed`). An action `line-dry` lasts the 50 minutes of sun that a load needs and does not look at the sky.

B. A durative action inside a window (`laundry-line-window`). The same action with `(over all (sunny))`: fifty minutes of sun without a break.

C. A process and an event (`laundry-v5`). The model of Listing 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]
```

Encoding A folds the wool at minute 86, after 25 minutes of sun and 25 of cloud. Encoding B waits for the cloud to pass and starts the fifty minutes at 90: the sun before the cloud is of no use to it, because a durative action cannot pause.

	A: fixed length	B: inside a window	C: process and event
Fidelity	an impossible plan: the wool is folded wet. Right only on a day without clouds	no impossible plans; lost solutions: drying cannot pause, and a day whose sunny spells are all shorter than 50 minutes has no plan	exact: drying stops under the cloud and resumes
Size	17 durative actions	17 durative actions	15 durative actions, 2 processes, 2 events
Guidance	10 407 states; makespan 91.005	9 535 states; makespan 146.002	168 296 states; makespan 169, and 122 by hand
Clarity	one more action like the others	the same, with one condition	the law of the line is apart from the chores; plans must be replayed
Portability	durative actions, timed literals	the same	processes: Kairos alone among the planners compared

Table 19.15.: Three encodings of the clothesline on the same morning (the problems `p02` of each domain; figures of `kairos -v`, one thread).

Verdict. A is the model for a place where the sun can be trusted, and for a first version: it is version 3 with one action more. It must not be used when the sky is part of the problem, since its plans are then wrong and nothing says so. B is the honest model inside the temporal level. It never returns a wrong plan and it runs on the planners that read timed literals; its price is the solutions that it loses, and on this morning it ends later than the best plan of C by hand (146 against 122) though earlier than the plan that Kairos finds for C. Choose it when sunny spells are long compared with the drying time. C is the only one that tells the truth about a broken morning, and it is needed exactly when drying is interrupted and resumes, which is the case that Chapter 17 reserved for processes. The decision depends on a question of step M that the story had not answered: *how reliable is the sun?* As with the waiter of Chapter 7, step L ended by sending us back to the card.

19.10. Common mistakes

The mistakes of this chapter are not in a construct. They are mistakes in the way of working, and each of them was made while the laundry was written.

A cheat test that passes for the wrong reason

The plan files of version 2 were carried to version 3 as they were. The test still says what it should, `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
```

The plan is rejected at its first line because it has no durations, not because 12 kg do not fit in the drum. The test is green and tests nothing; if the capacity rule were lost in the jump, it would stay green. *Fix*: translate the plan files with the model, and read the message of

every rejection, not only its verdict.

A header that tells the story of an older version

`laundry-v2-stale-domain.pddl` is version 2 with the comment and the requirements of the file it was copied from. Kairos plans it and gives a warning:

```
$ 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
```

The requirements are the part of the documentation that a machine reads: another planner decides from them whether it can plan the file. The comment above them, which still says that a machine holds one load, is read by people, and no tool checks it. *Fix*: the header is part of the jump. The list that `kairos check` prints under its verdict is what the file really uses: copy the requirements from it.

A "no plan" test with an open horizon

The first "no plan" test of version 5 let the sun shine until minute 50, sixteen minutes of drying for a sweater that needs fifty (`laundry-v5-p05.pddl`). With a time limit of 3 seconds:

```
$ 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
...
```

The search used the three seconds, and 371 765 states. With 20 seconds it uses the 20. Until minute 50 every instant in which the sweater can be hung leaves a different moisture, and each is a state to explore. The overcast morning of `p03` asks the same question of the model, does the line dry without sun, and is answered after 8 093 states in less than 0.1 seconds. *Fix*: the advice of Chapter 18. Bound the horizon of the problems that must have no plan.

A verdict kept when the card has changed

The sweater was the "no plan" test of four versions, and a test file copied forward would still expect exit code 10 for it in version 5, where the right answer is a plan of 90 minutes. The opposite slip is more dangerous: to see a test change its verdict after a jump and to update the expected value without asking why. *Fix*: rule 4 of Section 19.7. A verdict changes only together with a line of the model card, and the comment at the top of the problem file says which.

19.11. Summary

The MODEL method is now complete. Nothing in it is new since Chapter 1 except its content: five steps, each with a question, an artifact and, after eighteen chapters, a checklist.

- **Five checklists** (Section 19.2) hold every question that the chapters added. Step M decides more than any other: metrics, time, wishes, rules and a world that moves are entries of the model card before they are PDDL.
- **The decision guide** (Figure 19.2, Tables 19.3 to 19.5) goes from a need to a feature, its requirement and its cost. Features are added in the order of the book, and the model stops at the lowest level that does not lie about what the problem turns on.
- **Forty patterns** (Tables 19.6 and 19.7) are the vocabulary of steps O and D. Each has a rival, and the rivals are the columns of step L.
- **Anti-patterns** (Tables 19.8 to 19.10) pass the check and show in the plans: too easy, too hard, too big, or right for another question.
- **Debugging** starts by reading the verdict. `unsolvable` is a proof; `no plan found` and `timeout` are not. When the planner finds nothing, or nothing good, the plan written by hand and the validator tell the model from the search.
- **A model grows in versions**, one dimension at a time: the card first, the old tests translated and kept, new tests for what is new. The laundry went from 44 lines of classical PDDL to processes and events in five steps, and each version remains the best answer to some question.
- **Documentation** is the five artifacts in three places: the header of the domain, the model card, and a test file that a machine runs.

The next chapter uses all of it on a problem of the size that a real robot has: a day of a fleet of couriers in a hospital, from the story to the execution of the plan.

19.12. Further reading

- The book of Haslum and others Haslum et al., 2019 is the reference for the language of the whole book in one volume, with the semantics of each level. It is the place to check what a construct means when a planner and a validator disagree.
- The textbooks of Ghallab, Nau and Traverso Ghallab et al., 2004 and of Geffner and Bonet Geffner and Bonet, 2013 explain what the planners do with a model, which is the ground of the criteria Size and Guidance. The later book of the first three authors Ghallab et al., 2016 treats models that are used for acting as well as for planning, the subject of the next chapter.
- The papers that define each level discuss modeling choices of their own: Fox and Long for numbers and time Fox and Long, 2003, Gerevini and others for preferences and constraints A. E. Gerevini et al., 2009, Fox and Long again for processes and events Fox and Long, 2006.
- The paper of Cushing and others Cushing et al., 2007 is the analysis behind the distinction between models that need actions to overlap and models that do not, which this chapter met as a cost in Guidance and Portability.

- The paper of Howey, Long and Fox Howey et al., 2004 describes VAL, the validator of the planning competitions. The habit that this chapter relies on most, a plan written by hand and checked by a program that does not search, comes from there.
- The table of features and planners used in Section 19.3 is kept with the documentation of Kairos (Martín Rico et al., 2026).

19.13. Exercises

The exercises of this chapter are modeling projects. Each one is a story, and for each one the deliverables are the same five artifacts: (1) the **model card**; (2) the **state table**, with the pattern of each row; (3) the **action table**; (4) the **test set**, as PDDL files with a `tests.txt`, holding at least a trivial problem, a typical one, one that must have no plan and one cheat written by hand as a plan file; (5) the **comparison** of the encoding chosen with at least one alternative, written and run, on the five criteria. The stories leave things unsaid on purpose: deciding them, and writing the decision in the card, is part of the work. The solutions give one reference model for each project; a different model that passes its own honest test set is as good.

Exercises for Section 19.2 — The five checklists

Exercise 19.2.1. *The library.* Books returned by readers pile up at the desk of a library. A robot takes them back to the shelves. The shelves are in aisles, one per subject, along a single corridor that starts at the desk: to reach the third aisle the robot passes the first two. Every book belongs to one aisle. The robot has a basket that holds three books. The evening is over when every book is on its shelf and the robot is back at the desk.
Model it as a classical problem. Your typical test should have more books than the basket holds; your cheat tests, a fourth book in the basket and a book left in the wrong aisle.

Exercise 19.2.2. *The hotel floor.* Two cleaners make up the rooms of a hotel floor. Every room must be vacuumed, and the floor has one vacuum cleaner, kept in the store room, where it must be at the end. In the rooms whose guest has left today the bed linen is changed too; clean linen is in the store room, and a cleaner carries one set at a time. A room with a "do not disturb" sign is not entered.
Model it as a classical problem with the conditions of Chapter 5; state the goal as a rule about every room, not as a list. Look for a situation that the story does not settle, and turn it into your "no plan" test. When your model is done, read the best plan for two cleaners and explain what the second one does.

Exercises for Section 19.3 — Which feature for which need

Exercise 19.3.1. *The water truck.* In summer a tank truck waters the flower beds of a town. The beds are in a few places joined by roads of known length. Each bed needs a known amount of water, the tank holds 1000 liters, and it can be refilled at the hydrants, which are in some of the places. The truck starts full at the depot and returns there. The town wants every bed watered with as few meters driven as possible.

Use the decision guide to choose the features, and write down which questions of Figure 19.2 you answered with "no". Decide whether a bed can be watered in two visits.

Exercise 19.3.2. *The water network.* A plumber must repair leaking joints in the water network of a village. Water flows from the reservoir through pipes, each with a valve, and reaches every joint that is connected to the reservoir by open valves. A joint can be repaired only when it is dry. The network has rings, so a joint may get water from two sides. The clinic must have water at every moment of the work. At the end the valves that were open are open again.

The interest of this project is in what is *not* an effect of any action. Include a network where the obvious three-step repair breaks the rule of the clinic.

Exercise 19.3.3. *The turnaround.* An aircraft that arrives at its gate must leave again. The passengers get off (10 minutes); then the cabin is cleaned (15) and the catering is loaded (12); the aircraft is refueled (20), never with passengers on board; and when the cabin is clean and catered the new passengers board (15). The airport has one cleaning team and one fuel truck for all the gates, and a catering truck for each aircraft. The measure is the time at which the last aircraft is ready.

Compute by hand the shortest turnaround of one aircraft, and of two that arrive together, before running your model.

Exercise 19.3.4. *The causeway.* A van delivers parcels on an island that is joined to the mainland by a causeway. The causeway is dry at low tide only: today from the start of the day until minute 60, and from minute 300 to minute 360. Crossing takes 20 minutes and delivering a parcel 15. The driver's shift ends at minute 480, and the van must be back on the mainland with every parcel delivered.

Decide what must hold during a crossing and during a delivery. Among your tests, include one whose answer is "no plan" and that your planner proves, and try one that it cannot settle.

Exercise 19.3.5. *The robot vacuum.* A robot vacuum cleans a flat: a hall that gives onto a kitchen, a living room and a nursery, with the charging base next to the hall. Cleaning a room takes a part of the battery that depends on the room, moving between rooms a little more, and the battery is not enough for the whole flat. The family cares most about the kitchen, then the living room, then the nursery, then the hall. Two rules of the house: the hall is never cleaned before the kitchen, because the crumbs end up there; and the robot enters the nursery once at most, because the baby is asleep. The robot must end at its base.

Separate what is required, what is wished and what is a rule, and give each wish a weight with its reason.

Exercise 19.3.6. *The canal lock.* A boat on the lower reach of a canal must get to the upper reach, three meters higher, through a lock: a chamber with a gate at each end. The keeper can open the paddles of the upper gate, and the chamber fills at half a meter per minute until its level is that of the upper reach; or the paddles of the lower gate, and it empties in the same way. A gate can be opened only with the same water on both of its sides, and the two gates are never open together. At the end the boat is above and both gates are closed.

Decide who causes each change. Include a test in which the chamber starts full, with the upper gate open, and one that tries to open a gate too soon.

Exercises for Section 19.7 — Growing a model in versions

Exercise 19.7.1. *The laundry, with coins.* The laundromat of this chapter charges for every cycle: 3 coins for a wash and 2 for a drying cycle. A second washer, of 12 kg, costs 5 coins a wash. Grow version 2 of the laundry into a version that finds the cheapest way to do the three loads of its typical test, with and without the second washer. Follow the rules of Section 19.7: say what changes in the card, translate the tests of version 2 and check their verdicts, and add the tests that Chapter 9 asks for. Then say which later version of the chapter you would add the coins to next, and what you expect to happen to the metric.

Case study: a robot mission from story to execution

20.

Every chapter so far started from a feature of the language and looked for the situations that need it. Real work goes the other way. Somebody hands us a page written by a person who has never heard of PDDL, with robots, rooms, opening hours, urgent boxes and a few "if possible", and asks for a system that tells the robots what to do tomorrow morning. Nothing on the page says which requirement to declare. Chapter 19 put the whole MODEL method in one place; what is still missing is to see it carry a problem of that size from the first reading to a plan that a robot executes.

This chapter does that with the courier world at full size: a day of a hospital fleet, with two couriers, batteries, the pharmacy's opening hours, an urgent delivery, routine ones, the sterile lab and some wishes. We take the page through the five steps and through four versions of the model, each one a tested domain with its test set: a classical one, a temporal and numeric one, one with a calendar and wishes, and a PDDL+ variant of the battery. Each jump is made because the previous version cannot say something that the page says, and at each one we show the alternatives that we wrote, ran and discarded. Twice the planner cannot do what the model asks of it within our time limit; we say so, and show what a modeler does then.

The chapter has two halves. Sections 20.1 to 20.6 build the four versions. Sections 20.7 to 20.9 leave the planner and follow the plan to the robot: what a plan is for an executor, what can go wrong, when to plan again, how PlanSys2 does all this in ROS 2, and what PDDL leaves out. The exercises are projects.

20.1. The story

This is the page. It was written by the logistics manager of the hospital floor that we know from Chapter 1; we have only added numbers where she said "a while".

The morning round, as the logistics manager describes it

We have two courier robots, `r1` and `r2`. They sleep in the dock, where there is one charger. In the morning `r1` is fully charged; `r2` did the night errands and starts with about a third of its battery. A robot uses about one unit of battery per meter and holds a hundred. On the charger it gains five units per second, and a robot that plugs in stays until it is full: the contacts do not like being plugged and unplugged all the time.

Four requests are waiting. A box of medicines for the ICU, which is *urgent*: it must be there 80 seconds after the round starts. A routine box of medicines for ward A. A sample that the nurses of ward B took and that the lab must analyze. And a meal that the kitchen left in the lobby for ward B.

The pharmacy is not open yet. It opens 30 seconds into the round and closes at 90, and nobody hands a box to a robot through a closed shutter. The lab is sterile: a sample goes in only in a sealed bag, the robots carry bags and can seal one in a few seconds. Things are handed over only where they are going; I do not want boxes left in corridors.

20.1 The story 448
20.2 Step M: map the situation 449
A plan for the modeling 450
20.3 Version 1: who carries what 451
Steps O and D 451
Step E: the test set 453
Step L: three ways to write the rule of the lab 454
What version 1 cannot say 455
20.4 Version 2: time and battery 456
Steps O and D: what changes 456
Step E: the round in time 457
Step L: how long is a charge 459
What version 2 cannot say 460
20.5 Version 3: a calendar and wishes 460
Steps O and D: what changes 461
Step E, first part: the tests that pass 462
Step E, second part: the round, and a limit . . . 463
Step L: three ways to plan the wishes 464
What version 3 cannot say 466
20.6 Version 4: the battery belongs to the world . . 467
Steps O and D: what changes 467
Step E: one robot and a shutter 468
Step L: does the round need version 4? 470
20.7 From plan to execution 471
What a plan is for a robot 471
What can go wrong . . 471
Planning again 472
20.8 PlanSys2 473
Four nodes 473
A plan becomes a behavior tree 474
Action performers . . . 474
The application 475
What this asks of the model 476
20.9 What PDDL leaves out 476
20.10 The MODEL method: one story, four models 477
Alternatives compared 478
20.11 Common mistakes . . . 479
20.12 Summary 480
20.13 Further reading 481

> The round ends at 100 seconds, when the day shift takes the corridors. Whatever is not delivered by then waits for the next round. The urgent box is not negotiable. After it, I care most about the other medicines, then the sample, then the meal. And I like to find the robots back in the dock, if possible.

Figure 20.1 draws the situation at the start of the round. The times are short (a real round lasts hours, not seconds) because we keep the distances and the speed of the earlier chapters, 1 meter per second; multiplying every number by sixty changes nothing in what follows.

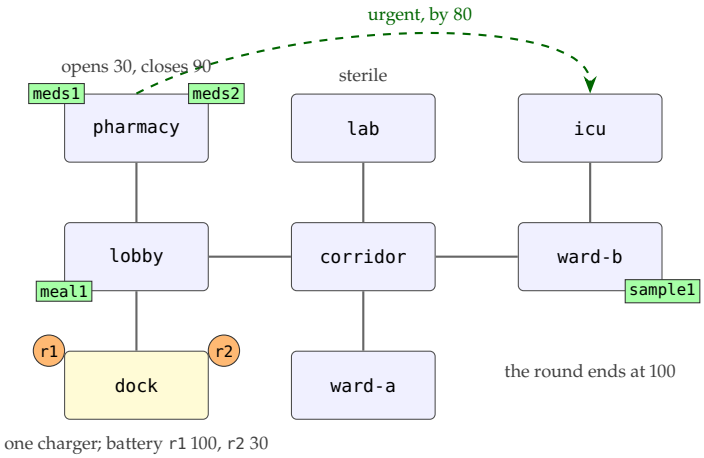


Figure 20.1.: The morning round. Two robots in the dock; `meds1` (urgent, for the ICU) and `meds2` (for ward A) in the pharmacy, `sample1` in ward B for the lab, `meal1` in the lobby for ward B. The notes are what the classical courier world never had to deal with at once.

20.2. Step M: map the situation

Step M is done with a pencil and, when possible, with the author of the page in front of us. A first reading leaves questions that only she can answer, and every answer is a line of the model card. These are the ones we asked.

- *Can a robot carry two things?* No: one compartment. (An assumption, as in Chapter 2.)
- *Can a robot leave the charger before it is full?* No, she had said so. This sentence will decide an encoding in Section 20.4.
- *May a robot hand a box to the other one, or leave it somewhere on the way?* No: "no boxes in corridors". The relay that the planner invented in Chapter 10 is forbidden here.
- *What happens if a sample enters the lab unsealed?* "It must not happen." She did not say that the sample is lost; she said that it is not done.
- *"If possible": compared to what?* A robot in the dock is worth less than any delivery. A delivery after the end of the round is worth nothing.
- *Does a robot spend battery while it waits?* "A little." We will leave it out at first and come back to it in Section 20.6.

Step M on a real page is a conversation: every question to the author of the story becomes a line of the model card.

Step M: the model card of the morning round

Story	Two courier robots serve the requests of one round on a hospital floor: an urgent box, routine deliveries, a sample for a sterile lab.
Agents	The robots `r1` and `r2`. They act at the same time; each does one thing at a time. The pharmacy, the charger and the clock are not agents: they follow a schedule or a law.
Success	Required: `meds1` delivered in the ICU by 80. Wished (table below): the other deliveries, before the round ends at 100; the robots in the dock.
Scope	In: rooms, connections and distances; where robots and items are; what a robot carries; sealing; durations; battery and charging; opening hours and deadlines. Out: the route inside a room, people in the corridors, how the arm grasps, failures, anything uncertain.
Assumptions	One item per robot. Items are handed over only at their destination. One robot at the charger, and it stays until full. Battery: 1 per meter, nothing while standing (revised in version 4). The schedule is known in advance: pharmacy open from 30 to 90, urgent deadline 80, end of the round 100.

Wish	**Condition**	**Weight**	**Why**
`meds2-done`	`meds2` delivered in ward A	5	medicines first
`sample-done`	`sample1` delivered in the lab	4	more than the meal; less than medicines
`meal-done`	`meal1` delivered in ward B	2	can wait a round
`r1-home, r2-home`	the robot ends in the dock	1 each	less than any delivery

The card has the five headings of Chapter 1 and the *Wishes* table that Chapter 14 added. Its line of Assumptions already contains a schedule, as Chapter 13 asks. The weights are the manager's order turned into numbers ($5 > 4 > 2 > 1$), with one property that we checked with her: no delivery is ever given up to bring the robots home, since even the meal is worth as much as both robots in the dock.

A plan for the modeling

The card needs time, numbers, a calendar and preferences. Writing all of it at once is the surest way to end with a model that does not run and no idea why. Chapter 19 gave the alternative: grow the model in versions, each one complete, tested and a little more faithful than the one before, and move on only when the current version cannot say something that matters (Figure 20.2).

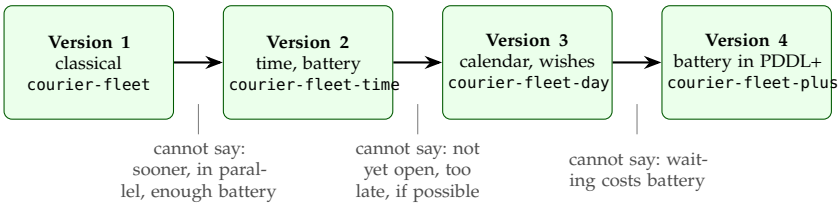


Figure 20.2.: The four versions of the case and, under each jump, what the version on its left cannot say. Every version keeps its test set, and every plan of a later version is compared with the plan of the one before.

20.3. Version 1: who carries what

The first version is classical. Its Scope is the first line of the card's Scope and no more: rooms, where things are, what is carried, sealing, and the two rules about handing over and about the lab. No time, no battery, no wishes: every delivery is a goal.

Steps O and D

We start from the typed domain of Chapter 4 and the sample of Chapter 6. Table 20.1 is the state table; the last four rows are new with respect to the courier world of Chapter 4.

Property	About	Kind	Encoded as
One can go from a room to another	two locations	static	`(connected ?from ?to)`
Where a robot is	robot, location	dynamic	`(robot-at ?r ?l)`
Where an item lies	item, location	dynamic	`(item-at ?i ?l)`
Which item a robot carries	robot, item	dynamic	`(carrying ?r ?i)`
The robot carries nothing	robot	dynamic	`(free ?r)`
Where an item must go	item, location	static	`(destination ?i ?l)`
The item was handed over	item	dynamic	`(delivered ?i)`
The room has an airlock	location	static	`(sterile ?l)`
The sample is in a sealed bag	sample	dynamic	`(sealed ?s)`

Table 20.1.: Step O for version 1. Types: `robot`, `location`, `item`, and `medicine`, `sample`, `meal` under `item`, as in Chapter 4.

Two decisions of the card are visible in the table. `destination` is a static table in the sense of the pattern catalog of Chapter 7: the requests differ between problems, so they are data and not part of the domain. And there is no fact for "contaminated": the manager's sentence was a prohibition, so the bad state will be unreachable and needs no name.

Action	Parameters	Applies when	Changes
`move`	robot, from, to	the robot is in *from*; the rooms are connected; if *to* is sterile, every sample the robot carries is sealed	the robot is in *to*, no longer in *from*
`pick`	robot, item, location	both are in the location; the robot is free; the item was not delivered yet	the robot carries the item; the item no longer lies there; the robot is not free
`seal`	robot, sample	the robot carries the sample; it is not sealed	the sample is sealed
`deliver`	robot, item, location	the robot is there and carries the item; the location is the item's destination	the item lies there and is delivered; the robot is free

Table 20.2.: Step D for version 1. There is no `drop`: `deliver` is a drop that only applies at the destination.

Three rows of Table 20.2 come from the conversation of step M. "No boxes in corridors" removed an action: with `deliver` in place of `drop`, an item that is picked can only be put down where it is going. It is the

one-way action of the catalog, chosen on purpose, and Section 20.7 will show its price. `(not (delivered ?i))` in `pick` keeps a robot from taking a delivered item back. And the rule of the lab is a precondition of `move`. Listing 20.1 is the domain.

```
1  ;; The hospital fleet, version 1 (classical): who carries what,
2  ;; where to, and the rule of the sterile lab.
3  (define (domain courier-fleet)
4    (:requirements :strips :typing :negative-preconditions
5                   :disjunctive-preconditions
6                   :universal-preconditions)
7
8    (:types robot location item - object
9            medicine sample meal - item)
10
11   (:predicates
12     (connected ?from ?to - location)
13     (robot-at ?r - robot ?l - location)
14     (item-at ?i - item ?l - location)
15     (carrying ?r - robot ?i - item)
16     (free ?r - robot)
17     (destination ?i - item ?l - location) ; static: the request
18     (delivered ?i - item)
19     (sterile ?l - location)       ; static: a room with airlock
20     (sealed ?i - sample))         ; the sample is in a sealed bag
21
22   ; Nobody enters a sterile room with a sample that is not
23   ; sealed: the rule of the lab, asked of every move.
24   (:action move
25     :parameters (?r - robot ?from ?to - location)
26     :precondition (and
27       (robot-at ?r ?from) (connected ?from ?to)
28       (or (not (sterile ?to))
29           (forall (?s - sample)
30             (or (not (carrying ?r ?s)) (sealed ?s)))))
31     :effect (and (not (robot-at ?r ?from))
32                  (robot-at ?r ?to)))
33
34   (:action pick
35     :parameters (?r - robot ?i - item ?l - location)
36     :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r)
37                        (not (delivered ?i)))
38     :effect (and (not (item-at ?i ?l)) (not (free ?r))
39                  (carrying ?r ?i)))
40
41   ; Put the carried sample in a sealed bag.
42   (:action seal
43     :parameters (?r - robot ?s - sample)
44     :precondition (and (carrying ?r ?s) (not (sealed ?s)))
45     :effect (sealed ?s))
46
47   ; Items are handed over at their destination and nowhere else:
48   ; there is no drop.
49   (:action deliver
50     :parameters (?r - robot ?i - item ?l - location)
51     :precondition (and (robot-at ?r ?l) (carrying ?r ?i)
52                        (destination ?i ?l))
53     :effect (and (not (carrying ?r ?i)) (free ?r)
54                  (item-at ?i ?l) (delivered ?i))))
```

Listing 20.1: Version 1: the classical domain of the fleet.

The condition of lines 28 to 30 reads: either the room entered is not sterile, or every sample is either not carried by this robot or sealed. It needs `:disjunctive-preconditions` and `:universal-preconditions` (Chapter 5). The `forall` is there although a robot carries one item: the rule should not depend on the assumption about the compartment.

Step E: the test set

The typical test is the round itself, `courier-fleet-p02.pddl`: the four deliveries and both robots back in the dock.

```
    (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)
```

Read the plan as the manager would. The sample is sealed in the corridor, one move before the lab: correct. Every item ends where it was going. And then two things that she would not accept. The urgent box is the third delivery of four. And every one of the 31 actions is of `r2`: `r1` never leaves the dock. With `-v`, Kairos reports `solved (proven optimal), cost 31`: no plan has fewer actions, and using the second robot would only add its trip out and back. The model counts actions, so one robot doing everything is the best plan there is.

The other three tests:

```
$ 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` is the round with the door between ward B and the ICU closed for cleaning. `p04` is the cheat test, in the form that Chapter 2 called "ask for the forbidden state": the goal `(and (delivered sample1) (not (sealed sample1)))` can only be reached by walking into the lab with an open sample. The second form of the cheat test is a plan written by hand, the one a lazy robot would follow, given to the validator:

```
$ 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
```

Step E: the test set of version 1

Test	File	Expected, and obtained
Trivial	`courier-fleet-p01`	5 steps: pick, three moves, deliver
Typical	`courier-fleet-p02`	31 steps, proven optimal; sample sealed before the lab
No plan	`courier-fleet-p03`	unsolvable: the ICU cannot be reached
Cheat	`courier-fleet-p04, p05`	unsolvable; the plan without `seal` is rejected

Step L: three ways to write the rule of the lab

The book has taught three encodings of "no unsealed sample in the lab", and we wrote all three as domains with the same name, so that the same problems run on each.

A, physics (`courier-fleet-when-domain.pddl`). The encoding of Chapter 6: `move` is always possible, and a conditional effect marks an unsealed sample that enters a sterile room as `contaminated`; `deliver` refuses such items.

B, a precondition (Listing 20.1). The move into the lab is not applicable.

C, a constraint (`courier-fleet-rule-domain.pddl`). `move` has no condition about samples, and the domain states the rule once, over every state of the plan, as Chapter 15 taught:

```
22  ; Never in a sterile room with a sample that is not sealed.
23  (:constraints
24    (always (forall (?r - robot ?s - sample ?l - location)
25      (imply (and (carrying ?r ?s) (robot-at ?r ?l) (sterile ?l))
```

```
            (sealed ?s)))))
```

All three give a 31-step plan for the round, proven optimal, answer `unsolvable` to `p04` and reject the plan without `seal`, each in its own words: a false precondition of `move` (B), a false precondition of `deliver`, because `(contaminated sample1)` is true (A), and `the plan violates the domain :constraints` (C). Table 20.3 compares them; the numbers are those of `kairos -v` on the round.

	A: conditional effect	B: precondition	C: constraint
Fidelity	Models what the airlock does. Allows plans that ruin a sample nobody asked to deliver	Says what the manager said: it is not done. Guards the door only (see below)	Says it for every state, whatever action leads there
Size	54 ground actions, 40 facts	80 ground actions (the `or` splits `move`), 41 facts	54 ground actions, 3 axioms added by the planner
Guidance	10 749 states; a contaminated sample is a dead end found late	10 432 states; no dead end	10 749 states
Clarity	The rule is spread over an effect, a fact and a precondition	One condition, in the action that could break the rule	One line, in the manager's words
Portability	`:conditional-effects`	`or`, `forall`: ADL preconditions	`:constraints`: few planners

Table 20.3.: Step L for version 1: three encodings of the rule of the lab, run on `courier-fleet-p02`.

Verdict. On Size and Guidance the three are within a few percent: at this scale they do not decide. We discard A on Fidelity: it describes a world in which samples get ruined, and the card describes a hospital in which that is not done; its extra fact and its dead end buy nothing. Between B and C the advice of Chapter 15 applies literally: the constraint is the specification, and the precondition is the encoding to use when the rule is about one state, few actions can break it and the model must travel. Ours must travel: the next versions add time, and Section 20.8 takes the domain to an executor that checks the conditions of each action while it runs. We keep B, and we keep C as a test: any plan of B can be given to `kairos validate` with the domain of C.

That test is worth running with imagination. B guards the door of the lab; C guards the room. They differ for a sample that is already inside, unsealed, and that a robot picks up there. The card says that samples go *to* the lab, so the round never contains that state and B is right for our Scope; Exercise 20.3.2 builds the problem that tells them apart.

A rule of the story has three encodings: physics with a bad outcome, a precondition, a constraint. Write the constraint as the specification even when the model uses the precondition.

What version 1 cannot say

Version 1 answers one question well: which robot must be where with what, and when to seal. It is silent on three things that the card says. It cannot say *sooner*: its best plan leaves a robot in the dock, and nothing in a count of actions rewards working in parallel. It cannot say *enough battery*: `r2` drives 190 meters in that plan, with a battery for 30. And it cannot say *urgent*. The first two need durations and numbers, and they are the next version.

20.4. Version 2: time and battery

Version 2 adds to the Scope: durations, robots working at the same time, battery and charging. It is the courier of Chapter 10 and Chapter 11 with the rules of version 1.

Steps O and D: what changes

The state table gains one fact and one lock, and six numbers (Table 20.4). Every action becomes durative, and each takes the token `(idle ?r)` at its start and returns it at its end, the way Chapter 10 keeps a robot from doing two things at once.

Property	Kind	Encoded as
The robot is doing nothing	dynamic	`(idle ?r)`
The room has a charger; nobody is plugged in there	static; dynamic	`(has-charger ?l)`, `(charger-free ?l)`
Meters between connected rooms; speed	static	`(distance ?from ?to)`, `(speed ?r)`
Charge, from 0 to the capacity	dynamic; static	`(battery ?r)`, `(battery-capacity ?r)`
Battery per meter; battery per second on the charger	static	`(consumption ?r)`, `(charge-rate ?r)`

Table 20.4.: Step O for version 2: what is added to Table 20.1. The names are those of Chapter 8, Chapter 10 and Chapter 11.

For each action we answer the four questions that Chapter 10 added to step D: how long, what must hold when, when each effect happens, what may run in parallel (Table 20.5).

Action	Duration	New in time
`move`	distance / speed	Battery for the whole stretch required and taken at the start. The robot is nowhere while it moves
`pick`	2 s	The robot stays in the room (`over all`). The item leaves the shelf at the start and is carried at the end
`seal`	3 s	The sample is sealed at the end
`deliver`	2 s	The robot stays in the room. Delivered, and the robot free, at the end
`charge`	missing battery / charge rate	Takes the charger at the start, returns it at the end. The battery is full at the end. The robot stays in the room

Table 20.5.: Step D for version 2. All actions take `(idle ?r)` at start and return it at end.

Earlier chapters used different handling times: a `handling-time` function with the value 2 in Chapter 10, a constant 2 in Chapter 12 and Chapter 13, 1 in Chapter 18. We take 2 seconds for `pick` and `deliver`, as constants: nothing in the story makes one robot faster at it than another. The moves and the charge are in Listing 20.2.

```
34  ; The battery for the whole stretch is taken at the start.
35  (:durative-action move
36    :parameters (?r - robot ?from ?to - location)
37    :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
38    :condition (and
39      (at start (idle ?r))
40      (at start (robot-at ?r ?from))
41      (at start (connected ?from ?to))
42      (at start (>= (battery ?r)
43                    (* (distance ?from ?to) (consumption ?r))))
44      (at start (or (not (sterile ?to))
45                    (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)))))
```

Listing 20.2: Version 2: `move` and `charge` (part of `courier-fleet-time-domain.pddl`).

The charger is the resource lock of the catalog: `(charger-free ?l)` is deleted at the start of a charge and added at its end. The battery of a move is taken whole at its start, as in Chapter 11: a discrete change, which Chapter 12 showed to be enough when nobody reads the quantity while the action runs.

Step E: the round in time

The problem of the round, `courier-fleet-time-p02.pddl`, has the numbers of the card and the metric `(:metric minimize (total-time))`. The robots need not return to the dock: that is a wish, and this version has no wishes. A temporal search is anytime and would use its whole time limit, so from here on we give it three seconds.

```
$ 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]
```

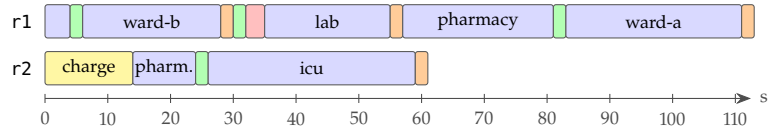


Figure 20.3.: The round in version 2 (makespan 113.017). Blue: moves, joined and labeled with where they end; green: pick; orange: deliver; red: seal; yellow: charge. `r2` charges for 14 seconds and makes one delivery; `r1` makes three.

Figure 20.3 draws it whole (the file `courier-fleet-time-p02.plan` has its 27 lines). Both robots work from the first second. `r2` charges from 30 to 100 in $70/5 = 14$ seconds and takes the urgent box to the ICU, 43 meters from the dock: it arrives with 57. `r1` takes the meal, picks the sample where it leaves the meal, seals it while standing in ward B, and ends with the routine box: 98 meters, on a battery of 100. The pauses of the plan are worth a look too: no robot waits between two of its actions. The makespan fell from "31 actions" to 113 seconds because the model can now say *sooner*.

The rest of the test set follows Chapter 10 and Chapter 11: the cheat test becomes a family, one member for each thing that must not overlap.

```
$ 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
```

In `p03` a robot stands in ward B with 20 of battery; the dock is 26 meters away and the delivery needs 34. The answer is `no plan found`, with exit code 14, and not `unsolvable`: as Chapter 11 explained, the temporal engines do not prove it. Twenty states were enough to run out, and our own arithmetic is the proof. The two hand-written plans are the tempting ones: a robot that starts to move while it is still picking, and two robots on the one charger. In `p04` both robots have 10 of battery and a delivery each, and the planner's plan takes turns:

```
$ 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]
```

While `r1` charges, `r2` spends 8 of its 10 units to fetch the meal from the lobby and is back at the charger, loaded, the instant it is free. We had not thought of that plan, and it is right.

Step E: the test set of version 2

Test	File	Expected, and obtained
Trivial	`...-time-p01`	32.004: pick, 28 m, deliver
Typical	`...-time-p02`	113.017; one charge of 14 s; batteries end at 2 and 57
No plan	`...-time-p03`	no plan found (exit 14); by hand, 20 < 26
Cheat	`...-time-p04 and two plans`	charges take turns (65.605); a robot doing two things and two robots on the charger are rejected

Step L: how long is a charge

Chapter 11 compared three encodings of a charge and Chapter 12 added a fourth. We wrote three for the fleet, again as domains with one name. **Full** (Listing 20.2): the charge lasts what is missing over the rate and fills the battery. **Chosen** (`courier-fleet-time-chosen-domain.pddl`): the duration is between 1 second and the time to full, and `?duration` times the rate is added at the end. **Continuous** (`courier-fleet-time-cont-domain.pddl`): the battery falls with `(* #t (drain-rate ?r))` during a move and rises with `#t` during a charge of chosen length. On the round, the last two return the same 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]
```

Four charges of one second each. Chapter 12 told us to expect this: Kairos tries a chosen duration at its bounds, and a robot that needs 46 units and has 30 is served by four times the lower bound. The plan is valid and six seconds shorter than ours. It also plugs the robot in four times in four seconds.

	Full charge	Chosen duration	Continuous
Fidelity	What the card says: a robot that plugs in stays until full	Has plans that the manager forbade (partial charges); would be right for another charger	As chosen; also right if somebody read the battery during a move
Size	82 ground actions	84	84
Guidance	One charge decides everything. Round: 113.017	The planner must choose a number and takes the bounds. Round: 107.015, with four charges of 1 s	The same plan, 107.015
Clarity	A computed duration and an `assign`	Says "for as long as you want"	The most natural reading of a battery
Portability	Durative actions with numbers	`:duration-inequalities`: Kairos, OPTIC, POPF	`:continuous-effects`: Kairos and POPF only

Table 20.6.: Step L for version 2: three encodings of the charge, run on the round. The lists of planners are those of Chapter 11 and Chapter 12.

Verdict. The full charge, and for a reason that is in the story: "a robot that plugs in stays until it is full" makes the shorter plan an impossible one, so the encoding that looked like a loss of solutions in Chapter 11 is here the faithful one. Had the manager answered "it can leave when it wants", the choice would be harder and the *planner* would enter it: a model whose good plans need a duration strictly between the bounds asks Kairos for something that its fast engines do not try, and the practical answer would be the blocks of Exercise 20.4.2. That is the Guidance criterion at work on a real decision. The continuous battery adds nothing while nobody reads the battery during a move; it will, in Section 20.6.

What version 2 cannot say

Look again at Figure 20.3 with the card in hand. `r2` picks the urgent box at 24, from a pharmacy that opens at 30. `r1` picks the routine box at 81, fine, and delivers it at 113, after the end of the round. In the other plan, the urgent box reached the ICU last of all, at 107. Version 2 has durations and no calendar: it cannot say *not yet open* or *too late*. And it has only goals: give it a round in which not everything fits before 100 and its only possible answer is "no plan", when the manager's answer is "then leave the meal".

20.5. Version 3: a calendar and wishes

Version 3 adds the last two lines of the card: the schedule under Assumptions and the table of wishes under Success. They are the subjects of

Chapter 13 and Chapter 14.

Steps O and D: what changes

The domain changes little. Two facts enter the state table, with the names of Chapter 13: `(is-open ?l)`, the room is open now, and `(in-time ?i)`, the deadline of the item has not passed. No action changes them. They are dynamic all the same, because the world changes them at known times, and each problem says when with timed initial literals. Two actions gain one condition each (Listing 20.3): the room must be open during the whole `pick`, and a delivery must end in time.

```
  ; Somebody hands the item to the robot: the room must be open
  ; during the whole pick.
  (: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))
      (at start (free ?r))
      (at start (not (delivered ?i)))
      (over all (robot-at ?r ?l))
      (over all (is-open ?l)))
  ; A delivery counts only if it ends before the deadline.
  (:durative-action deliver
    :parameters (?r - robot ?i - item ?l - location)
    :duration (= ?duration 2)
    :condition (and
      (at start (idle ?r))
      (at start (carrying ?r ?i))
      (at start (destination ?i ?l))
      (over all (robot-at ?r ?l))
      (at end (in-time ?i)))
```

Listing 20.3: Version 3: `pick` and `deliver` read the calendar (part of `courier-fleet-day-domain.pddl`).

Everything else is in the problem. The calendar and the wishes of the card are the end of `courier-fleet-day-p02.pddl`:

```
    ; the calendar of the day
    (is-open lobby) (is-open ward-b)
    (at 30 (is-open pharmacy))
    (at 90 (not (is-open pharmacy)))
    (in-time meds1) (in-time meds2)
    (in-time sample1) (in-time meal1)
    (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)))))
```

The end of the round is the deadline pattern used three times: after 100, a delivery of a routine item is not possible any more, which is how the model says that it is worth nothing. The urgent box is the only hard goal. The requirements are `:timed-initial-literals` and `:preferences`.

Step E, first part: the tests that pass

The first test costs nothing: the plan of version 2 is a file, and the validator can read it against version 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
```

The new version rejects the old plan for the reason we gave when we read it. This is a habit worth keeping at every jump: the plans of version n are test cases of version $n + 1$, and the message should name what version n could not say.

The trivial test has one robot, the urgent box, the pharmacy that opens at 30 and the deadline at 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]
```

The robot is at the shutter at 10 and stands there for twenty seconds. The box is in the ICU at 67.005, thirteen seconds before its deadline. The cheat is to pick at 10; the plan that does it is rejected with the message we have just seen. For the test without a plan we move the deadline to 30, with the pharmacy open from the start (`p03`), and Kairos proves it:

```
$ 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)
```

A subtler problem has no plan either: the pharmacy opens at 30, the deadline is at 60, and only a pick before the opening would be in time (`p04`). Here the planner does not conclude:

```
$ 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)
```

The proof of `p03` reasons about facts that timed literals take away; in `p04` the obstacle is a fact that a timed literal brings late, and the robot can also recharge for ever, so the search has no end. The answer is a timeout, exit code 11, and it is ours to complete: $30 + 2 + 33 + 2 = 67 > 60$. Chapter 11 gave the rule for these tests, and it holds here: prefer the ones whose answer you can compute, and give the tempting plan to the validator.

Step E, second part: the round, and a limit

Now the round with both 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]
```

The plan is valid and it is the plan of the trivial test. Its cost is 12 out of a possible 13: the urgent box is delivered and the only wish granted is `r2-home`, by a robot that never moved. Chapter 14 warned that a search with soft goals starts from the plan that is cheapest to find and improves from there. Here it does not improve within our three seconds. With 25 seconds the same command reached cost 7; we know, from the next pages, that 1 is possible.

Before changing anything we follow the debugging procedure of Chapter 7, which starts by relaxing. The file `courier-fleet-day-p07.pddl` is the round without its calendar: everything open, no deadline, the same wishes.

```
$ 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
```

Every wish granted, in less than a tenth of a second. So the model of the wishes is right, and the planner handles two robots and five preferences easily. What it does not handle in three seconds is all of that *with* timed literals. The verbose output of the round names the difference: `engine M3: timed initial literals: actions must`

`happen within time windows`. Without a calendar Kairos uses its engine M2, which searches the problem almost as a classical one; a calendar needs the engine that searches over starts and ends (Chapter 11), which is slower, and the preferences then multiply a space that was already large.

When the planner disappoints, relax the problem until it does not: the feature whose removal fixes the answer is the one that costs.

This is a fact about Kairos 1.0.6 and not about PDDL, and another planner would put the limit elsewhere. But a model is used with a planner, and the criteria Guidance and Portability exist to make room for facts like this one. We cannot remove the calendar. We can change what we ask in one call.

Step L: three ways to plan the wishes

A, one problem. The round as written: two robots, one hard goal, five preferences (`p02`).

B, one robot at a time. Two problems for the same domain. The first has `r1` alone, the urgent box and every wish (`p05`). The second has `r2` alone and the wishes that the plan of `r1` left out (`p06`). The two plans are merged into one file, and the validator judges it against problem A.

C, a ladder of hard goals. No preferences: plan for the urgent box; then for the urgent box and the routine one; and so on down the manager's order, keeping the last problem that has a plan. `p08` is the third rung: the two boxes and the sample.

```
$ 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]
```

Alone, `r1` delivers the urgent box and the sample and goes home; it forgoes the routine box and the meal, cost 5 + 2 = 7, found in under a second. `r2` is then asked for those two and for its own return: it delivers the box, goes home and forgoes the meal, cost 2. The merged plan is `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
```

Everything but the meal. Before taking that to the manager, read the plan as she would. `r1` crosses the lobby at 4, where the meal lies, and then stands at the shutter of the pharmacy for twenty seconds. It had no reason to take the meal: in its own problem the urgent box came first, and a robot that takes the meal to ward B is not back at the pharmacy in time for the ICU. But *we* decided that `r1` would choose first. The other order is two more files, `p10` (`r2` alone, with the urgent box and every wish) and `p11` (`r1` alone, with what is left). Alone, `r2` charges and then does what `r1` did: the urgent box, the sample and home, cost 7. And `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
```

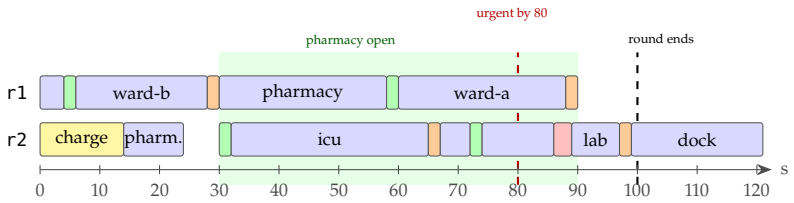


Figure 20.4.: The round in version 3 (metric 1, makespan 121.014). Colors as in Figure 20.3. `r1` delivers the meal before the pharmacy opens and the routine box after it; `r2` charges, waits at the shutter from 24 to 30, and delivers the urgent box at 67 and the sample at 99, one second before the end of the round. Only `r1` does not get home.

Figure 20.4 draws the second merged plan. With `r2` on the urgent box, `r1` is free to use the half minute before the pharmacy opens, and every delivery arrives in time. The one loss is `r1-home`: `r1` ends in ward A with 18 of battery, 26 meters from the dock. A day with the meal, the routine box and the way home is 108 meters, and the battery holds 100. We believe that 1 is the best possible cost; we have not proved it, and neither has the planner.

The ladder was less lucky. Its third rung, three hard deliveries under the calendar, is a problem that has a plan (either of the two above) and gets this answer:

```
$ 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: one problem	B: one robot at a time	C: ladder of hard goals
Fidelity	Exact: the card, line by line	Loses the plans in which the second robot should have chosen first; any plan it returns is valid for A	Loses the weights: an order only. A rung that fails says nothing about a cheaper mix
Size	76 durative actions, 5 preferences	Half of it, twice	As A without preferences, several times
Guidance	Poor with a calendar: cost 12 in 3 s, 7 in 25 s	Good: each call ends improving in under a second. Total 2 with `r1` first, 1 with `r2` first	Poor: the third rung times out
Clarity	One file is the whole round	Two files and a merge; who goes first is a new decision	Many files; the answer is spread over runs
Portability	Timed literals together with preferences: few planners	The same features, smaller problems	Timed literals only

Table 20.7.: Step L for version 3: three ways to plan the round of wishes, with Kairos 1.0.6 and three seconds per call.

Verdict. A is the model and B is how we plan with it. Problem A stays as the specification, because it is the only one of the three that says what the manager said, and the validator prices any plan against it: `metric 2`, `metric 1`. B is a decomposition, and it is honest only because of that last step. A merged plan can fail where the robots meet, here at the one charger, and the validator of the joint problem is what would catch it. And B has a cost, which we have just paid and recovered: it decides who goes first before planning. Our first choice had a sensible reason (`r1` can leave at once, `r2` must charge) and was worse by a delivery. With two robots there are two orders, and we run both; with five there are a hundred and twenty, and B becomes a search of its own. With another planner, or another year's version of this one, A alone may do; the files are the same.

What version 3 cannot say

In Figure 20.4 `r2` stands at a closed shutter for six seconds, and in our first plan `r1` stood there for twenty. In the model those seconds are free. The manager said that a waiting robot spends "a little". Whether a little matters depends on the numbers, and to find out we need a model in which the battery goes down when nobody acts.

20.6. Version 4: the battery belongs to the world

In versions 2 and 3 the battery changes when an action of the robot says so. Chapter 17 asked of every change who causes it, and for a battery the honest answer is: the world. It drains while the robot is on, faster while it drives, and a charger fills it at its own pace and lets go when it is done. Version 4 is the fleet with the battery of Chapter 18: the level that the decision guide of that chapter calls "mixed", durative actions together with processes and events.

Steps O and D: what changes

Table 20.8 is the new answer to "who causes each change". Two rates replace the consumption per meter: 0.2 per second whenever the robot is on and 0.8 more while it drives, so that a meter still costs 1; the charger gives 5.2, which is 5 net. The numbers are those of Chapter 18.

Change	Caused by	In the domain
The robot goes to the next room	the robot	durative action `move`; it sets `(driving ?r)` while it runs and needs `(powered ?r)` throughout
The robot is plugged in	the robot	action `plug`; takes `idle` and the charger
The battery drains	the world, while on	processes `drain` and `motor-drain`
The battery fills	the world, while plugged	process `charging`
The charger lets go	the world, at an instant	event `battery-full`; returns `idle` and the charger
The robot switches off	the world, at an instant	event `shutdown`
The pharmacy opens	the clock	timed initial literal

Table 20.8.: Steps O and D for version 4: every change with its cause, as Chapter 17 asks.

```
  ; 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))))
```

Listing 20.4: Version 4: plugging in, and what the world does (part of `courier-fleet-plus-domain.pddl`).

The sentence "a robot that plugs in stays until it is full" has found its natural form. There is no `unplug` and no duration to compute: `plug` takes the token `idle`, and only the event `battery-full` gives it back. A plan does not mention the event; it happens.

Two things were left out to make this version work, and both are decisions shaped by the planner. *Handling is instantaneous*: `pick` and `deliver` are plain actions, as in the PDDL+ couriers of Chapter 17 and Chapter 18. With durative ones the search in steps of time still solved the easy tests, but not the cheat test of this section within five seconds. *There are no samples*: the type, `seal` and the rule of the lab are gone. With the disjunctive condition of Listing 20.1 in `move`, Kairos starts with a search phase that, on this domain, finds nothing and uses half of the time limit: the trivial test went from a hundredth of a second to three seconds of a limit of six. We could have rewritten the rule in another form. We chose to write in the Scope what this version is for: *Version 4 is about the battery; the lab is tested in versions 1 to 3.* A variant does not have to carry everything.

Step E: one robot and a shutter

The step E of a hybrid model follows the six points of Chapter 18, and the first is the pencil. The cheat test, `courier-fleet-plus-p04.pddl`, has a robot at the door of the pharmacy with 32 of battery; the pharmacy opens at 30 and the delivery is 28 meters away. Version 3 has an opinion about this problem, which we kept as `courier-fleet-day-p09.pddl`: wait at the door, pick at 30, arrive with 4 to spare.

```
$ 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]
```

With a pencil: thirty seconds of waiting cost 6, the robot leaves with 26, and 28 meters need 28. The same plan in the vocabulary of version 4 is `courier-fleet-plus-p04-wait.plan`, and the validator, which integrates the processes exactly, agrees with the pencil:

```
$ 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
```

The robot switched off two meters from ward A. The planner, with the same files:

```
$ 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
```

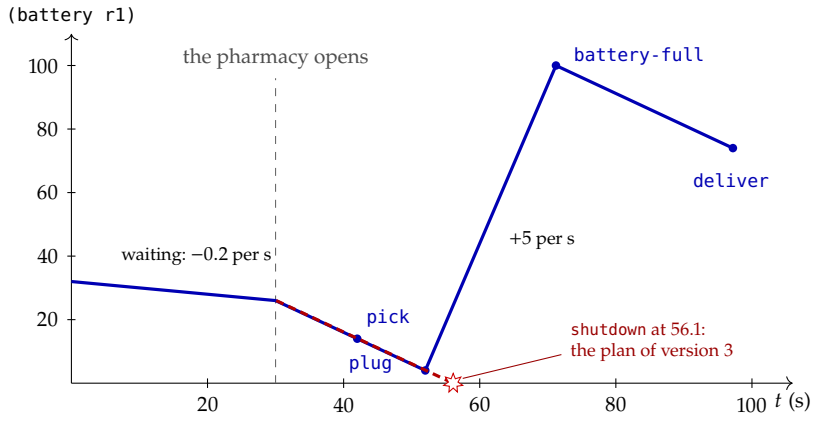


Figure 20.5.: The battery in the cheat test of version 4. Dashed: the plan that version 3 found acceptable; the robot leaves the door with 26 and switches off after 26 meters of the 28. Solid: the plan of version 4, which goes to the dock with the box and charges. The event `battery-full` at 71.2 is not in the plan; the robot moves on when it fires.

Figure 20.5 replays both, as Chapter 18 advises for any hybrid plan. The plan is right where it matters: it fetches the box, takes it to the dock, arrives with 4, charges to full and delivers with 74 left. It is also a plan with habits that we know. It does nothing until 30, because the search lets time run to the next thing that changes, the opening. It then walks to the lobby and back before picking, twelve seconds that nothing requires: Kairos says in a note that for PDDL+ *the search minimizes the number of actions; the metric of the problem is reported for each plan but does not guide the search yet*. A person would pick at 30 and be done by 85. The model accepts that plan too; the planner did not look for it.

The typical test is the morning of `r2` with its real numbers (`p02`: battery 30, the box for ward A, back to the dock). Its plan plugs in at 0, is released by the charger at 14, leaves at 30 and is home at 94, with the delivery at 68. A plan written by hand that leaves at 14.2 is valid and ten seconds shorter (`courier-fleet-plus-p02-hand.plan`, makespan 84.1). And the test without a plan, `p03`, has 5 of battery at the closed shutter:

```
$ 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)
```

Waiting empties the battery in 25 seconds and the dock is 10 meters away. In version 3 this robot would wait for ever at no cost and fail only at the first move.

Step E: the test sets of versions 3 and 4

Test	File	Expected, and obtained
Trivial	`...-day-p01`	67.005; the pick waits for 30
Typical	`...-day-p02`	one call: cost 12. Per robot and merged: metric 2 (p05, p06), then 1 (p10, p11)
No plan	`...-day-p03, p04`	unsolvable (proved); timeout, and by hand 67 > 60
Cheat	`two plans`	the early pick and the plan of version 2 are rejected
Trivial	`...-plus-p01`	28.1: pick, 28 m, deliver
Typical	`...-plus-p02`	94; by hand 84.1, valid
No plan	`...-plus-p03`	no plan found (exit 14); by hand, $5/0.2 = 25 < 30$
Cheat	`...-plus-p04`	97.2 with a charge; the wait of version 3 is rejected

Step L: does the round need version 4?

The comparison is between two levels, and Chapter 18 has the general table. Ours is about this round (Table 20.9).

	Version 3: battery taken by each move	Version 4: battery as processes and events
Fidelity	Waiting is free: accepts the plan of Figure 20.5 that ends in a shutdown	Waiting costs; the charger decides when a charge ends. No samples, instantaneous handling
Size	76 durative actions for the round	17 ground actions for one robot, and a state with a clock
Guidance	Seconds for one robot with wishes	Under a second for one robot and one goal; two robots and two deliveries did not fit in our five seconds; the makespan does not guide
Clarity	A reader sees the timetable in the plan	Half of what happens is not in the plan; a plot is needed to read it
Portability	Temporal planners with timed literals	PDDL+ with durative actions: Kairos

Table 20.9.: Step L for version 4: the same battery at two levels.

Verdict. Version 3 plans the round, and version 4 audits it. In the plan of Figure 20.4 the robots stand still for 8 and for 17 seconds, picks and deliveries included; at 0.2 per second that is 1.6 and 3.4 units, and the tighter robot, `r2`, gets home with less than 7 where version 3 said 10. The audit passes, with little to spare. We do not replace version 3: it is the only one that carries the wishes and both robots within our limits. We use version 4 for what it alone can say, on the pieces where waiting and battery meet, one robot at a time; and we carry its lesson back to version 3

in the cheapest possible way, as a margin (keep 10 units in reserve) rather than as a process. The question of Chapter 18, *which is the lowest level that does not lie about what the problem turns on*, has a different answer for the round, which turns on the calendar, and for a robot parked in front of a shutter.

20.7. From plan to execution

We have a plan: the 31 lines of Figure 20.4. A planner's work ends there. The robots have not moved.

What a plan is for a robot

For the planner a plan is a proof: started in the initial state, under the assumptions of the model, these actions at these times reach the goal. For a robot it is a set of instructions, and each line needs three things that the model does not contain.

Somebody to carry it out. `(move r1 lobby pharmacy)` is one line and six seconds in the plan. On the robot it is a navigation task with a map, a localization, a path, obstacles and a controller: the level of detail that Chapter 1 left "to the component that knows about geometry". Every action of the domain needs a piece of software that performs it and reports back: done, failed, so far along.

A reason to start. The plan says `32.001`. But `r2` leaves the pharmacy at 32.001 because the pick ended at 32, not because of the clock; if the pick takes three seconds, the move must start at 33. The times of a plan are one consistent way of meeting its real content, which is a set of dependencies: this action needs what that one produces, these two must not overlap, this one cannot start before the pharmacy opens. An executor that reads the plan as a timetable is **time-triggered**; one that starts each action when the actions it depends on have ended, and its conditions hold, is **event-triggered**. Sending each action to its performer at the right moment is called **dispatching**. The simple temporal networks of Chapter 11 are the usual tool to keep the dependencies with their time bounds (Dechter et al., 1991).

A model that is still true. The plan was valid in a world with the durations, the battery rates and the schedule of the problem file. The executor is in the other one.

What can go wrong

Table 20.10 lists what happens to plans. Our plan tells us where to look. The urgent box arrives at 67 with a deadline at 80: thirteen seconds of slack, over 43 meters. The sample is delivered at 99.011 and the round ends at 100. The validator says `plan valid`, and it is; a robot that slows down for a stretcher in the corridor makes it false. A plan is as robust as its smallest slack, and neither the metric nor the planner looked at that.

What happens	In the round	A reasonable reaction
An action takes longer	a move of 12 s takes 15	nothing, while the dependencies and the deadlines still hold; otherwise plan again
An action fails	the box slips at the pick	try again; after some attempts, plan again without that item
A condition breaks while an action runs	the pharmacy closes early during a pick	stop the action, plan again
The state is not what the plan assumed	the charger gave `r2` 92, not 100	check the conditions of what remains; plan again if one fails
The task changes	a new urgent request at 30	plan again with the new goal

Table 20.10.: What can go wrong between a plan and its execution.

Planning again

Almost every reaction of the table ends in the same step: stop, look at the world, write a new problem and call the planner. **Replanning** is the default repair of model-based autonomy, and it is cheap precisely because of the split of Chapter 1: the domain does not change, only the problem (Figure 20.6).

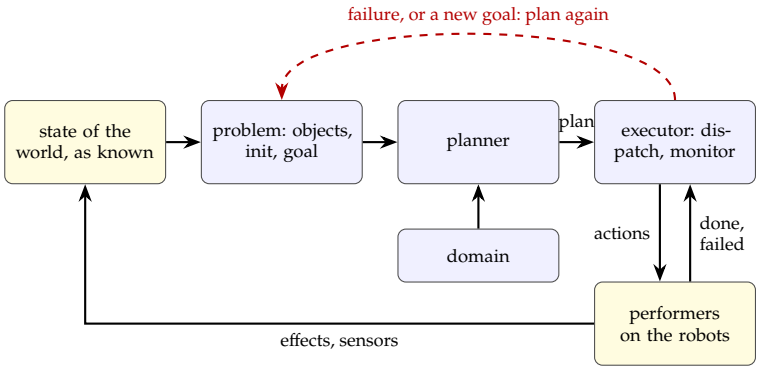


Figure 20.6.: Plan, execute, monitor, plan again. The domain is fixed. The problem is written anew each time from the state of the world as the system knows it.

Three things make replanning harder than the figure, and the first two are about the model.

The state in the middle must be a state of the model. Suppose that at 70 the door of ward A is found blocked, with `r1` in the corridor carrying the routine box. The new problem starts with `(carrying r1 meds2)`, and in our domain there is exactly one way to stop carrying: `deliver`, at the destination, which cannot be reached. `r1` is useless for the rest of the round. "No boxes in corridors" removed `drop`, and with it every way to give up a delivery. A model that will be executed needs the actions of recovery (take the box back to where it came from), even if no good plan ever uses them.

Time has passed. A problem written at 30 has a calendar that starts at 30: the round ends in 70 seconds, not in 100. Every timed literal is shifted, and those in the past become plain facts of the initial state. And *while* the planner thinks, the clock runs: a plan that took three seconds to find must be valid for a world three seconds older.

Something is always running. At 30 `r2` is in the middle of a pick. The honest choices are to wait for a quiet instant, to cancel what runs, or to let it finish and plan from the state it will leave. Exercise 20.7.1 takes the third.

Execution needs performers, dependencies instead of clock times, and a way back to the planner. A model meant for execution keeps actions for recovery and slack for delays.

20.8. PlanSys2

PlanSys2, the ROS 2 Planning System (Martín et al., 2021), is an open-source framework that implements the loop of Figure 20.6 for robots that run ROS 2. This section describes it as it is in its sources at version 4.0.0; we state only what we read there, and we did not run the fleet on it.

Four nodes

A PlanSys2 system is four ROS 2 nodes (Figure 20.7), and the comment in its start-up program says so: "the four nodes of PlanSys2, and no other". All four are *lifecycle nodes*, which are configured and then activated in order. They can run in one process or separately, in which case a fifth program, a lifecycle manager, starts them.

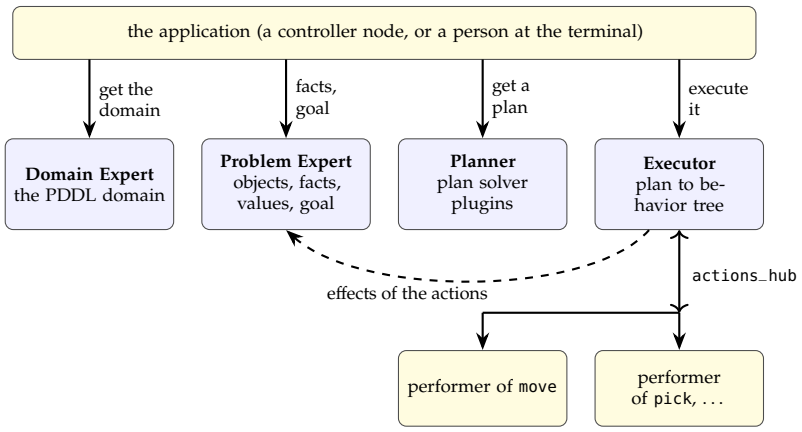


Figure 20.7.: PlanSys2: four nodes, the application that drives them, and one node per action of the domain. The executor applies the effects of the actions to the Problem Expert as they happen.

The Domain Expert holds the PDDL domain. It reads it when it is configured, from the file named by its parameter `model_file`. The parameter may list several files separated by colons; they are merged into one domain, so that each component of a robot can bring its own actions. The domain must pass three checks: PlanSys2's own parser must read it; it must contain nothing that PlanSys2 cannot execute; and it is then checked as PDDL, by default with the checker of Kairos used as a library, the one behind `kairos check`. The node then answers questions about the domain: its types, predicates, functions, actions and durative actions, and the details of each.

The Problem Expert holds the other half: the instances (objects), the predicates and functions that are true now, and the goal. It is the system's knowledge of the world. Every change requested is verified against the domain, and at any moment the node can write what it holds as the text of a PDDL problem. It can also start from a problem file.

The Planner turns a domain and a problem, both as text, into a plan. The planners themselves are plugins, the *plan solvers*, listed in the parameter `plan_solver_plugins`. In the sources we read the default is Kairos, run as a library inside the node; a plugin for POPF (Coles et al., 2010) comes with the system, and it works as one would do by hand: it writes `domain.pddl` and `problem.pddl` in a temporary folder, runs the planner and parses what it prints. Each solver declares the requirements it supports, and a task is given only to solvers that cover what it uses. The time limit is the parameter `plan_solver_timeout`, 15 seconds by default. A plan is a list

of items with three fields, `time`, `action` and `duration`: the lines that this book has printed since Chapter 10.

The Executor receives a plan as the goal of a ROS 2 action called `execute_plan` and carries it out.

A plan becomes a behavior tree

The Executor does not walk down the list of the plan. It builds from it a *behavior tree*, a structure common in robotics for composing tasks that succeed, fail or are still running, and executes the tree (with the library BehaviorTree.CPP). The builder is a plugin too; the default one, `STNBTBuilder`, builds the tree from a simple temporal network of the plan. Each action of the plan becomes a small subtree. The template for a whole action is one of the most instructive files of the system for a PDDL modeler (the default builder uses the same nodes, split into a part for the start of the action and a part for its end):

```
<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>
```

This is the durative action of Chapter 10, executed. The subtree waits for the actions that this one depends on. It waits until the `at start` conditions hold in the Problem Expert (for a limited time, 30 seconds by default) and applies the `at start` effects there. Then it runs the action, and all the while it checks the `over all` conditions again and again: that is what a *reactive* sequence does. When the action reports that it ended, the `at end` conditions are checked and the `at end` effects applied. The three places of a condition and the two places of an effect, which were a matter of semantics in Chapter 10, are here five nodes of a tree. A condition in the wrong place is checked at the wrong moment on a real robot.

If a subtree fails (the action reported a failure, or a condition did not hold), the plan fails: the Executor cancels the actions that are running and ends the `execute_plan` action with the result `FAILURE`. It does not plan again by itself. That decision belongs to the application, and the Executor helps with it in two ways that we found in the sources: it can give the problem "of now", from which to plan again while a plan runs, and it accepts a new plan while the old one is running, in which case the actions that appear in both go on.

Action performers

What executes `(move r1 lobby pharmacy)`? A node written by the user of PlanSys2, one (at least) for each action of the domain: an **action performer**. The base class is `plansys2::ActionExecutorClient`; the node says in its parameter `action_name` which action of the domain it performs, and implements one method, `do_work`, which is called periodically while the

action runs. This is the `move` of the simplest example that comes with PlanSys2, which only pretends to move:

```
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_;
};
```

A real performer would send a goal to the navigation system and call `finish(true, ...)` when the robot arrives, or `finish(false, ...)` when it cannot. The arguments of the action (which robot, from where, to where) are available to it, and a parameter called `specialized_arguments` lets a performer accept only some of them, for instance only the moves of `r1`: in a fleet, each robot runs its own.

The Executor and the performers do not call each other directly. They talk on a topic, `actions_hub`, with a small protocol. The Executor publishes a *request* for an action with its arguments; the performers that can do it and are free answer; the Executor *confirms* the first and *rejects* the others; the chosen one sends *feedback* while it works and a *finish* message at the end, and can be *cancelled*. The Executor needs no list of who does what, and a second robot is one more set of nodes that answer. A performer may also be a behavior tree itself: the package `plansys2_bt_actions` provides one that runs a tree given in a file.

The application

The last piece is the program that knows what the robot is for. It tells the Problem Expert what exists and what is true, sets a goal, asks for a plan and sends it to be executed. In the patrolling example of PlanSys2 the calls are these (shortened):

```
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());
}
```

The domain and the problem travel as strings, the same text that this book has kept in files. That example is a small state machine around

these calls: when a plan ends, it sets the next goal and plans again; when one fails, it decides what to do. For interactive use there is a terminal with commands such as `set instance`, `set predicate`, `set goal`, `get plan` and `run`.

What this asks of the model

Which PDDL can be used? The parser of PlanSys2 reads domains with types, constants, predicates, functions, actions and durative actions, and also derived predicates, processes, events and constraints, and it accepts the requirements of every chapter of this book as names. Two other filters matter more. The plan solver must support what the task uses: the POPF plugin, for instance, declares `:strips`, `:typing`, `:equality`, `:universal-preconditions`, numeric fluents, `:durative-actions` and `:timed-initial-literals`. And the Executor must be able to check and apply the conditions and effects of each action against the Problem Expert, since that is how the plan is run; a domain with constructs it has no way to execute is refused when it is loaded, with the message `PlanSys2 cannot execute this domain`. We have not tried our four versions against that filter, and a modeler who targets PlanSys2 should do it first, with the terminal, before writing a single performer.

Whatever the result, the framework explains several decisions of this chapter after the fact.

- **Actions are what performers can do.** `move`, `pick`, `seal`, `deliver` and `charge` are five pieces of software that a roboticist can write and test alone. The granularity of step D is also the software architecture of the robot.
- **Conditions are checked on a running robot.** `(over all (is-open ?l))` in `pick` is evaluated while the arm moves. It is true or false in the Problem Expert, so somebody must tell the Problem Expert that the pharmacy opened. A timed initial literal is a promise about the world; in execution something has to keep it or report that it was broken.
- **The full charge is one action with a natural end.** A performer for `charge` docks, waits for the charger and reports. The four one-second charges of Section 20.4 would be four requests, four dockings.
- **The state is facts.** Whatever the application may need to say about the world in the middle of a round must be a predicate or a function of the domain. A robot holding a box that it cannot deliver was such a state, and our domain had no action for it.

20.9. What PDDL leaves out

Chapter 1 listed the classical assumptions and promised to lift them one by one. The book has lifted three: numbers made the state space infinite, time made actions last and overlap, PDDL+ made the world change by itself. Two are still standing after twenty chapters, and execution is where they show.

Determinism. In every model of this book an action has one outcome. In the corridor a pick sometimes fails, a door is sometimes blocked, a move takes between ten and twenty seconds. PDDL as we have studied it cannot say “sometimes”, and cannot say “usually”. A plan is one

sequence, not a strategy with a branch for each outcome. The standard practical answer is the one of this chapter: plan as if the world were deterministic, with the most likely outcome and some margin, watch, and plan again. It works well when failures are rare and not dangerous, and when no action is irreversible in a way that matters. It works badly when the right behavior is to *prepare* for a failure: no deterministic plan takes a second bag in case the first one tears.

Full knowledge. The initial state of a problem file is complete: the closed world assumption of Chapter 2 says that what is not listed is false. A robot does not know that much. It may not know whether the sample is ready, in which ward the cart was left, or whether the door is locked until it gets there. To find out it must look, and looking is an action whose effect is on what the robot *knows*, not on the world: a **sensing** action. A model with them has plans that branch on what is seen, and a state that is a set of possibilities instead of a set of facts; this is planning under **partial observability**. Our models hide it in the executor: the performer of `pick` looks for the box with a camera, and the planner only hears "done" or "failed".

Other things are out for humbler reasons. *Geometry*: whether the box fits, where exactly to stand. *Other agents* with their own goals: the people in the corridor are neither actions nor processes. *Quantities that are learned*, not known: every duration in our problems is somebody's estimate. *New objects*: a problem lists its objects, and a request that arrives during the round is a new problem.

Planning research has languages and algorithms for most of this: nondeterministic and probabilistic models, contingent and conformant planning, models that mix task and motion. The textbooks named in the further reading cover them. For the modeler the practical point is where to put the boundary. Each thing left out of the PDDL model does not disappear; it lands in a performer, in the monitor, or in the margin of a number. The model card has a line for exactly this, Scope, and at the end of a project it is the line worth reading again: for each item under "Out", who takes care of it now?

What the model leaves out does not disappear: it goes to a performer, to the monitor or into a margin. The Scope line of the model card should say where.

20.10. The MODEL method: one story, four models

The method was complete in Chapter 19; a case study adds no step. It adds experience of how the steps behave at this size, and we collect it as questions, as every chapter did.

What the case adds to the method

M	What did I have to ask the author of the story? (Each answer is a line of the card.) In which order will the versions come, and what will each one be unable to say?
O, D	For each version: what is added to the tables of the previous one? If the answer is "almost everything", the jump is too big.
E	Does each version keep its four tests? Was every plan of the previous version given to the validator of this one, and is it rejected for the reason I expected? For each test without a plan: does the planner prove it, give up, or time out, and can I compute the answer by hand?
L	Does the comparison include the planner I will use and the time I can give it? Which sentence of the story, if it were different, would change the verdict?
After	Who performs each action? What is the smallest slack of the plan? Which state in the middle of a plan could the model not start from?

Two of these deserve a comment. The first is the role of the story in step L. Three of our four verdicts were decided by a sentence of the manager: "it must not happen" chose the precondition over the conditional effect; "stays until it is full" chose the fixed charge; "a little" sent the PDDL+ battery to the role of an auditor. The five criteria do not choose in the abstract. Fidelity is fidelity *to a card*, and a different card gives a different winner with the same table.

The second is the role of the planner. Twice a correct model got a poor answer in our time limit, and both times the reaction had the same three moves: *relax* the problem to find which feature costs; keep the full model as the *specification*; and get plans from an easier question, with the validator of the full model as judge. The validator is the fixed point of the whole chapter. It read plans against a constraint in version 1, the plan of one version against the next, merged plans against a joint problem, and a hand-written plan in continuous time in version 4.

Alternatives compared

The alternatives of each version were compared in their sections (Tables 20.3, 20.6, 20.7 and 20.9). What remains is to compare the versions themselves, as answers to the page of Section 20.1. Table 20.11 takes the three that have time; version 1 is their common base.

Table 20.11.: The versions as models of the same page, with their plans for the round or, for version 4, for one robot of it.

	Version 2: time, battery	Version 3: calendar, wishes	Version 4: PDDL+ battery
Fidelity	Impossible plans: picks at a closed pharmacy, deliveries after the round. No wishes	The card, except that waiting is free	The battery as it is; no wishes, no lab
Size	82 durative actions	76, and 5 preferences	17 for one robot
Guidance	Best: both robots, 113.017 in under a second	One robot at a time: cost 1 with four calls of 3 s; one call: cost 12 in 3 s	One robot, one goal: under a second
Clarity	Plans read as timetables	The problem file reads like the card	Plans need a plot
Portability	Most temporal planners	Planners with timed literals and preferences	Kairos

Verdict. No version replaces the others, and that is the normal end of a modeling project. *Version 3* is the model of the round: use it to plan, one robot at a time while the planner needs it. *Version 2* is what to fall back on when there is no calendar, for instance a night shift with everything open; its plans come at once and are the best of the three. *Version 4* is the instrument for questions about energy and waiting, one robot at a time. *Version 1* still earns its place: it is where the rules about who may carry what are tested in milliseconds and proved, before time makes proofs rare. A team that kept only the last version would have lost its fastest tests.

20.11. Common mistakes

Writing the last version first

The temptation with a page like ours is to open a file and write durative actions, a battery, timed literals and preferences in one sitting. When the planner then answers

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

there is no way to know whether a rule is wrong, a number is wrong, or the problem is only hard. **Fix:** grow the model in versions. When version 3 disappointed us we could say at once that the rules were right (version 1 proves them), that the durations and the battery were right (version 2 plans them), and look only at what was new.

Taking the first answer of a problem with wishes

```
; better plan 2
; cost = 12
; makespan = 67.005
```

The plan is valid and the exit code is 0. It grants one wish of five. With soft goals, "solved" means that the hard goals are met and nothing more. **Fix:** know before running what a good cost would be (here: at most the weight of the cheapest delivery, 2), read the cost and not only the status, and if the gap is large, relax the problem to find out why.

Reading a timeout as "there is no 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)
```

This was the answer for `courier-fleet-day-p08`, which has a plan: we had it in a file. The book has now met three different answers for "I return no plan": `unsolvable` (exit code 10, a proof), `no plan found` (14, the search ran out without proof) and `timeout` (11, the search did not finish). A ladder of problems that treats the third as the first drops deliveries that were possible. **Fix:** for the problems that decide something, have a plan written by hand, or a computation,

on the other side.

A valid plan with no slack

```
97.011: (deliver r2 sample1 lab) [2.000]
```

The delivery ends at 99.011 and its deadline is 100. `plan valid`, says the validator, and the first slow door makes it false. Planners that minimize a makespan or a cost push actions against the deadlines, because nothing in the model says that being early has value. **Fix:** put the margin in the model, where the planner must respect it: a deadline of 95 in the problem for a real one of 100, a reserve of battery. And after planning, compute the smallest slack of the plan as one more test.

A model that cannot start from the middle

Our domain has no `drop`, by decision. Give the planner the state that an interrupted round leaves behind, a robot carrying a box whose destination is closed, and any goal for that robot is unreachable: the note says that `deliver` "is never applicable", as for `p03` of version 1. In planning this was a faithful model. In execution it is a robot lost for the day. **Fix:** before a model goes to an executor, list the states in which a plan can be interrupted and check that each one has a way out; add the recovery actions, with a cost or a preference against them if good plans should avoid them.

20.12. Summary

One page of a logistics manager became four tested models. Table 20.12 gathers what each one was for.

Version	It says	Its plan for the round
1, classical	who carries what; the rule of the lab; hand over only at the destination	31 actions, proven optimal, all by one robot
2, temporal and numeric	durations, parallel work, battery, a charger for one	113.017 s, one charge; ignores the calendar
3, calendar and wishes	opening hours, deadlines, required and wished	metric 1: every delivery in time, one robot not home
4, PDDL+ battery	waiting costs; the charger ends the charge	one robot: valid where version 3 would switch off

Table 20.12.: The four versions of the morning round.

- **Step M is a conversation.** The sentences that decided our encodings were answers to questions that the page had not anticipated.
- **Versions, each with its tests.** Each jump was made for something the previous version could not say, and the plans of one version tested the next.
- **Alternatives were run.** Three encodings of a rule, three of a charge, three ways to plan wishes, two levels for a battery; in each case the discarded ones are files that still pass their tests.
- **The planner is part of the decision.** Durations at their bounds, a slow engine under a calendar, a costly search phase for a disjunction in PDDL+: each shaped an encoding, and each is written down next to the decision.

- **The validator is the judge.** Of hand-written plans, of plans from another version, of merged plans, of plans in continuous time.
- **A plan is not yet behavior.** Execution needs performers, dependencies, a monitor and the way back to the planner; PlanSys2 provides these for ROS 2, with the domain as the contract between all the parts.
- **What is out of the model is somebody's job.** Uncertainty, sensing and geometry went to the performers, the monitor and the margins.

The book ends where Chapter 1 began: a general solver and a model. The solver is somebody else's work and gets better every year. The model is yours.

20.13. Further reading

For acting with plans, the reference is the second book of Ghallab, Nau and Traverso (Ghallab et al., 2016), which is organized around the idea that deliberation continues during execution; their first book (Ghallab et al., 2004) has chapters on planning under uncertainty and on planning with time. The simple temporal networks used to dispatch temporal plans were introduced by Dechter et al. (1991). Geffner and Bonet (Geffner & Bonet, 2013) give a compact account of the models beyond the classical one, with non-deterministic actions and with partial observability. PlanSys2 is described by its authors in (Martín et al., 2021); its sources and tutorials are the place to check anything said here, since the system keeps changing. The validation of plans, on which this chapter leaned at every step, goes back to VAL (Howey et al., 2004), and Kairos is described in (Martín Rico et al., 2026). For the features combined in the case, the defining papers are those of PDDL 2.1 (Fox & Long, 2003), PDDL 2.2 (Edelkamp & Hoffmann, 2004), PDDL3 (A. Gerevini & Long, 2005) and PDDL+ (Fox & Long, 2006).

20.14. Exercises

The exercises of this chapter are projects. Each asks for files that pass their tests and for the reasons of the decisions; the reference solutions are in `examples/ch20/solutions`. Give temporal problems a time limit of three seconds, as the chapter does.

Exercises for Section 20.3 — Version 1: who carries what

Exercise 20.3.1. The ICU becomes a restricted room: only a robot with a keycard may enter, and only `r1` has one. Extend version 1 with the vocabulary of Chapter 5 (`restricted`, `has-keycard`), keeping the round as the typical test. Write the four tests. Which line of the model card changes? Does the best plan still use one robot?

Exercise 20.3.2. Section 20.3 claims that the precondition of `move` guards the door of the lab and the constraint guards the room. Write a problem on which the domain of Listing 20.1 and the one with the constraint (`courier-fleet-rule-domain.pddl`) disagree. Then repair the first domain so that they agree, and check that the round still has its 31-step plan. What does this tell you about the Fidelity row of Table 20.3?

Exercises for Section 20.4 — Version 2: time and battery

Exercise 20.4.1. The hospital installs a second charger in ward B. Write the problem (no change in the domain is needed: why?) and run the round. Compare the plan with the one of Figure 20.3 and explain what you see. Is a model with more options always planned better?

Exercise 20.4.2. Suppose that the manager had answered "a robot can leave the charger whenever it wants". Write the third encoding of Chapter 11, a charge in blocks of 5 seconds, as a variant of version 2, run the round and add a column to Table 20.6. Which encoding would you now choose, and what did the block size decide?

Exercises for Section 20.5 — Version 3: a calendar and wishes

Exercise 20.5.1. The reception of the lab closes at 90, and an item can only be delivered to somebody: the destination must be open during the whole delivery. Change version 3 (which action, which kind of condition?) and plan the day of `r1` when it chooses first (`p05`). What does `r1` lose, and could the other robot recover it? What happens to the plan of Figure 20.4?

Exercise 20.5.2. Write the deadline of the urgent box as a constraint of Chapter 15, `(within 80 (delivered meds1))`, in the trivial test of version 3, without its timed literal. Run it. Then say which of the timed literals of the round could be written that way and which could not.

Exercise 20.5.3. The kitchen complains, and the meal is now worth 6. Plan the day of `r2` when it chooses second (`p06`) with the new weight. Which delivery does it make? Check what was said about the weights after the model card: does it still hold? Explain to the manager, in two sentences, what her new number means for the medicines.

Exercises for Section 20.6 — Version 4: the battery belongs to the world

Exercise 20.6.1. Give the robots of version 4 the switch of Chapter 17: an idle robot can be powered off and on, and a robot that is off does not drain. Write by hand the plan for the cheat test `courier-fleet-plus-p04` that waits switched off, and validate it. Then run the planner on the new domain. Does it find your plan? What would you tell somebody who concludes from the planner's answer that the switch is useless?

Exercises for Section 20.7 — From plan to execution

Exercise 20.7.1. The round of Figure 20.4 is being executed. At 30, when `r1` has just delivered the meal in ward B, a new urgent request arrives: a sample, `sample2`, is waiting in ward B and must be in the lab before the round ends. Write the replanning problem for `r1`: decide what to do about `r2`, which is in the middle of a pick; shift the calendar; compute the battery of `r1`; decide what becomes of the routine box that `r1` was going to fetch. Plan it. Then check, on the timeline, that the new plan and the rest of the plan of `r2` do not conflict.

Exercises for Section 20.10 — The MODEL method: one story, four models

Exercise 20.10.1. A new story. *"Our warehouse works at night with two forklifts. Pallets arrive at the gate and each one has its place in rack A or rack B. Between the gate and the racks there is one aisle, and it is narrow: a forklift fits, two do not, and the same is true in front of each rack. The gate area is wide. Driving from one place to the next takes about 8 seconds, loading or storing a pallet about 5."* Take it through the method: the model card (with the questions you would ask), a classical version with its four tests, and a temporal version. Pay attention to what "a forklift fits, two do not" means while a forklift is on its way from one place to the next.

Appendices

Kairos A.

Kairos is the planner used for every example of this book: an open-source temporal and numeric PDDL planner written in C++20, usable as a library, as a command and from Python (Martín Rico et al., 2026). It is developed within the PlanSys2 organization (https://github.com/PlanSys2/kairos) and does not depend on ROS. This appendix collects what a reader of the book needs: how to install it, the commands and options that the chapters use, how to read what it prints, and a short look at its Python module. The complete reference is the documentation that comes with the planner (`README.md` and `docs/cli.md`). The outputs printed here were obtained with version 1.0.6.

A.1 Installing 485
With pip 485
From source 485
In a colcon workspace . . 485
A.2 The commands 486
A.3 Planning 486
Options 487
Engines and searches . . 487
A.4 Plan formats 488
A.5 Status and exit codes . . 488
A.6 Checking a model 489
A.7 Validating a plan 490
A.8 Looking inside 491
A.9 From Python 491

A.1. Installing

With pip

The quickest way, on Linux (x86-64 and arm64), is the Python package. It installs the `kairos` command together with the Python module of Section A.9:

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

From source

Kairos builds with plain CMake. It needs CMake 3.22 or newer and GCC 11 or Clang 17, or newer. From the root of its repository:

```
$ cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
$ cmake --build build
$ ctest --test-dir build --output-on-failure
```

The tests are optional; they use GoogleTest. The Python module can also be built from the source tree, with `pip install .`

In a colcon workspace

Robotics users often keep their software in a colcon workspace. With the repository of Kairos inside the `src` folder of the workspace:

```
$ colcon build --packages-select kairos
$ source install/setup.bash
```

After the `source` line, `kairos` is on the `PATH`, with completion of commands, options and PDDL files in bash and zsh. Outside colcon, the completion is turned on with one line in `~/.bashrc`:

```
$ eval "$(kairos --completion bash)"
```

For zsh the argument is `zsh`. For fish, the line is `kairos --completion fish | source`.

A.2. The commands

Command	What it does
`kairos D P [options]`	plans; the plan goes to the standard output
`kairos check D [P]`	checks a domain, or a problem against its domain, without planning
`kairos check --problem P`	checks a problem alone
`kairos validate D P PLAN`	checks that a plan solves a problem
`kairos --features`	lists the PDDL features and how far Kairos plans with each
`kairos --version`, `--help`	the version; the usage text

Table A.1.: The commands of Kairos. `D` is a domain file and `P` a problem file. The first command can also be written `kairos solve D P`.

Two rules hold for all of them. The standard output gets only results (plans, the version, the usage text); everything else (diagnostics, progress, statistics, the verdicts of `check` and `validate`) goes to the standard error, so that redirecting the output to a file gives a clean plan. And every command ends with an exit code that a script can test (Section A.5).

A.3. Planning

With the courier problem of Chapter 2, from the folder `examples/ch02`:

```
$ 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 reads and checks the two files, plans, and validates every plan against the problem before printing it. The search is *anytime* by default: after the first plan it keeps looking for better ones and prints each as it is found, separated by a line `; better plan N`. The last plan printed is the best. The search ends when it proves that nothing better exists, when the time runs out, or when one of the stop options says so.

The **cost** of a plan, printed in its first line, is the value of the `:metric` of the problem when it has one; otherwise it is the number of actions for a sequential plan and the makespan for a temporal one.

Options

Option	Meaning (default)
`--timeout S`	time for the search, in seconds (30)
`--no-anytime`	stop at the first plan
`--target-cost C`	stop at the first plan of cost C or less
`--no-improvement S`	stop when S seconds pass without a better plan
`--output FILE`	write the plans to files, not to the standard output
`--format F`	plan format: `ipc`, `popf` or `json` (`ipc`)
`--threads N`	phases of the search run at once (the cores, up to 4); with 1, runs are reproducible
`--memory MB`	memory limit (half of what is available)
`--engine E`	planning engine: `auto`, `m1`, `m2`, `m3` (`auto`)
`--search NAME`	search configuration (`auto`)
`--epsilon E`	separation between interfering happenings of a temporal plan, in seconds (0.001)
`-v`	progress and statistics on the standard error
`--debug[=CHANNELS]`	debug messages, all or of some stages (`--debug=search,ground`)
`--dump[=WHAT]`, `--dump-dir DIR`	write the intermediate structures to files

Table A.2.: Options of the planning command. Options and files can be given in any order; `--timeout 5` and `--timeout=5` are the same.

With `--output FILE` and anytime search, each plan goes to its own file, `FILE.1`, `FILE.2` and so on, and the highest number is the best plan. With `--no-anytime` the single plan goes to `FILE`.

```
$ kairos courier-domain.pddl courier-p01.pddl --output plan
$ ls plan*
plan.1
```

Engines and searches

The default, `auto`, chooses how to plan from the task, and is the right choice almost always. A task without durative actions or timed initial literals is searched directly (engine M1). A temporal task where no action needs to overlap another is first planned by compression (engine M2: each durative action becomes one instantaneous action, and the plan is scheduled afterwards) and then, with anytime search, improved by engine M3. Engine M3 plans with the start and the end of each durative action as separate steps over a temporal network, as the planner POPF does (Coles et al., 2010); it is the one for tasks that need actions to run concurrently (Chapter 11) and for timed initial literals (Chapter 13). PDDL+ (Chapter 17) is planned in steps of time.

The searches are named after the planners they follow. The one `auto` uses for a classical task, `lama`, is a greedy best-first search guided by two heuristics, the relaxed plan of FF and a count of landmarks, as in LAMA (Richter & Westphal, 2010); Chapter 3 explains these words. The others (`ff`, `eager-lama`, `novelty`, `lama-novelty`, `ehc`, `goals`) are described in `docs/cli.md`.

A.4. Plan formats

The default format, `ipc`, is the one of the planning competitions and of the validator VAL (Howey et al., 2004). A sequential plan is one action per line, as above. A temporal plan gives for each action its start time and, in brackets, its duration; this is the temporal template of Appendix B, from the folder `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]
```

The format `popf` prints what the planner POPF prints, which is what the POPF plugin of PlanSys2 (Martín et al., 2021) reads: a marker, the cost, and every action with a time and a duration, also in sequential plans, where the time is the position.

```
$ 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]
```

With `popf` on the standard output only the best plan is printed, when the search ends, and the exit code is always 0. The format `json` is for scripts: one line per plan, an object with the fields `temporal`, `cost`, `makespan` and `steps`, each step with its `time`, `duration` and `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. Status and exit codes

When a task is not solved, or always with `-v`, a status line on the standard error says how the run ended:

```
kairos: STATUS (STOP REASON)[, cost C]
```

Code	Status	Meaning
0	`solved`	at least one valid plan, also when the search was then stopped by the time, the memory or an option
10	`unsolvable`	proved: no plan exists
11	`timeout`	out of time before the first plan
12	`out of memory`	memory limit reached before the first plan (also `grounding too large`)
13	`unsupported`	the task uses something Kairos cannot plan with
14	`no plan found`	the search ran out of states, but it is not complete for this task: probably no plan exists, and nothing proves it
2	`invalid input`	wrong options, unreadable files, errors in the PDDL
1		internal error, or stopped by a signal before any plan

Table A.3.: Exit codes of the planning command. Codes 10 and 14 are different answers: the first is a proof, the second is not.

The stop reason of a solved task is one of `proven optimal` (sequential tasks: nothing cheaper exists), `exhausted` (the searches ended; on temporal tasks this is no proof that the makespan is the smallest), `timeout`, `no improvement`, `target reached`, `stopped` and `memory limit`. Without a plan, a note above the status line says why and which option could change it; Chapter 3 shows the three common cases. A script reads the exit code:

```
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. Checking a model

`kairos check` verifies PDDL without planning: syntax, declarations, types, arities, variables. It prints a verdict and the PDDL features that the files use. For the typed courier of Chapter 4, in `examples/ch04`:

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

The verdict is `valid, plannable`; `valid, not plannable yet`, for correct PDDL that uses a feature Kairos does not plan with (none today: `kairos --features` lists every feature as `full`); or `invalid`. The exit code is 0 for valid files and 2 otherwise.

A diagnostic has a fixed shape: the file, the line and the column; the severity (`error`, `warning` or `note`); the message; a code in brackets; and, below, the offending line with a mark under the exact place. If we swap the two arguments of `(robot-at r1 dock)` in the problem:

```
$ 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
```

Read a diagnostic from the left: where, how serious, what. A `note` that follows an error usually carries the fix, as here; so does a question in parentheses, as when `:effect` is misspelled in the domain:

```
$ 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
```

Warnings do not make a file invalid. The most common one in a course is about a feature used and not declared in `:requirements` (`semantic.undeclared_requirement`). The planning command shows the warnings and notes of parsing only with `-v`; errors are always shown.

A.7. Validating a plan

`kairos validate` checks that a plan solves a problem: every step is applicable at its time, interfering happenings are separated, and the goal holds at the end. The plan can be one written by Kairos, by hand, or by another planner: it accepts timed lines (`t: (action) [d]`, the duration optional), plain lines (`(action)`), the whole output of POPF or OPTIC, and comments that start with a semicolon.

```
$ kairos validate courier-domain.pddl courier-p01.pddl plan.1
plan valid, makespan 7
```

The verdict is `plan valid` (exit code 0), with the makespan and the metric when the problem has one, or `plan invalid` (exit code 1), after an error that names the step and the condition that failed. Chapter 3 shows both kinds of invalid plan: a step that cannot be applied, and a goal that is not reached.

One caution: the anytime output of the planning command holds several plans one after the other, and read as one file they are not a plan. To validate a plan of Kairos, write it with `--no-anytime` or with `--output` (and validate the last file).

A.8. Looking inside

Three options show what the planner does with a model; Chapter 3 uses them all.

`-v` prints, on the standard error, the warnings of parsing, one line per stage with its time (`[ground]`: the atoms and ground actions; `[sas]`: the variables; `[orchestrator]`: the engine and the search chosen, and the final status; `[search]`: each plan), the status line, and two lines of statistics: where the time went, and the size of the task and of the search (states, expanded, evaluated, plans, peak memory).

`--debug` adds debug messages. Without a value it turns on every channel; with a list, only those: `--debug=search` prints each new best value of the heuristic. The channels are `general`, `parse`, `semantic`, `normalize`, `ground`, `sas`, `search`, `heuristic`, `temporal`, `postprocess`, `validate`, `orchestrator` and `memory`. The equals sign is required.

`--dump` writes the intermediate structures to files, in the current folder or in the one given with `--dump-dir`:

Name	File	Contents
`ast`	`ast-domain.pddl`, `ast-problem.pddl`	the files as parsed, printed back as PDDL
`lifted`	`lifted.txt`	the task after normalization, before grounding
`ground`	`ground.txt`	the ground atoms and actions
`sas`	`sas.txt`	the variables, the groups of exclusive facts and the actions over them
`plan-trace`	`plan-trace.txt`	the best plan step by step, with the facts that hold after each step

Table A.4.: What `--dump` writes. `--dump=ground,plan-trace` writes only those two.

Two of them are worth knowing when a model misbehaves. `ground.txt` shows which actions were instantiated: one that is missing points to a precondition that can never hold. `plan-trace.txt` shows the state after each step of the plan, which is the fastest way to see where a plan does something the modeler did not intend.

A.9. From Python

The package `kairos-planner` gives the module `kairos`. Planning from files takes three lines:

```
import kairos
result = kairos.solve("courier-domain.pddl", "courier-p01.pddl",
    timeout=10)
print(result.status, result.plan)
```

A longer example loads the model once, plans, and validates the plan:

```python
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)")
```

Run in `examples/ch04`, it prints the plan of seven actions and `True`. The rest of the module follows the command:

- **Options** are keywords of `solve`: `timeout`, `memory_limit_mb`, `anytime`, `engine`, `threads`, `epsilon`, `target_cost` and others.
- **Anytime**: `Planner().plans(problem, timeout=30)` yields each better plan as it comes, and `Planner().start(problem, on_plan=...)` runs the search in its own thread.
- **Statuses** are those of Table A.3: `SOLVED`, `UNSOLVABLE`, `NO_PLAN_FOUND`, `TIMEOUT`, `OUT_OF_MEMORY`, `UNSUPPORTED`; `result.diagnostics` says why there is no plan.
- **Errors in the PDDL** raise `kairos.PddlError`. Each of its `diagnostics` has a `code`, a `message` and a `location`: the same information as the diagnostics of Section A.6.
- **Checking and validation**: `check_domain`, `check_problem`, `validate`, `validate_plan`, `parse_plan` and `pddl_support()`.

PDDL quick reference

B.

This appendix is for looking things up. It has three parts: the requirements of PDDL, with what each one enables and the chapter that teaches it; the syntax of a domain and a problem in five templates, one per level of the language, each a small version of the courier world that Kairos checks and solves; and a table of which planners plan with which features.

B.1 Requirements 493

B.2 Syntax templates 494

Classical: PDDL 1.2 494

Numeric: PDDL 2.1, level 2 496

Temporal: PDDL 2.1, levels 3 and 4, and PDDL 2.2 . . 497

Preferences, constraints and object fluents: PDDL 3 499

Hybrid: PDDL+ 500

B.3 Which planners support what 502

B.1. Requirements

A domain declares in `:requirements` the features it uses. Tables B.1 and B.2 list them in the order of the book. The version is the one of PDDL that introduced the feature.

Table B.1.: The requirements of PDDL: classical and numeric planning.

Requirement	What it enables	PDDL	Ch.
`:strips`	actions with a conjunction of facts as precondition, add and delete effects	1.2	2
`:typing`	`:types`, typed objects, parameters and predicates; `either`	1.2	4
`:equality`	`(= ?a ?b)` between objects	1.2	4
`:negative-preconditions`	`not` in preconditions and goals	1.2	5
`:disjunctive-preconditions`	`or` and `imply` in preconditions and goals	1.2	5
`:existential-preconditions`	`exists` in preconditions and goals	1.2	5
`:universal-preconditions`	`forall` in preconditions and goals	1.2	5
`:quantified-preconditions`	the two before, together	1.2	5
`:conditional-effects`	`when` in effects; with `forall`, effects on many objects	1.2	6
`:adl`	all of the above, in one word	1.2	6
`:numeric-fluents`	`:functions` with numeric values, comparisons, `increase`, `decrease`, `assign`, `scale-up`, `scale-down`; written `:fluents` in PDDL 2.1	2.1	8
`:action-costs`	`(total-cost)`, increased by actions and minimized by the metric	3.1	9

Requirement	What it enables	PDDL	Ch.
`:durative-actions`	`:durative-action`, with `at start`, `over all`, `at end`	2.1	10
`:duration-inequalities`	a duration between bounds, chosen by the planner	2.1	11
`:continuous-effects`	change during an action, with `#t`	2.1	12
`:derived-predicates`	`:derived`: facts defined by a rule	2.2	13
`:timed-initial-literals`	`(at t (fact))` in `:init`: facts that change at known times	2.2	13
`:timed-initial-fluents`	the same for numeric values; an extension that several planners accept	—	13
`:preferences`	`preference` in goals and preconditions, `is-violated` in the metric	3.0	14
`:constraints`	`:constraints` with `always`, `sometime`, `within` and the others	3.0	15
`:object-fluents`	functions whose value is an object	3.1	16
`:fluents`	since PDDL 3.1: numeric and object fluents together	3.1	16
`:time`	`:process` and `:event`	PDDL+	17

Table B.2.: The requirements of PDDL: time, PDDL 3 and PDDL+.

A `:metric` needs no requirement of its own. When a file uses a feature that it does not declare, Kairos plans all the same and gives a warning (`semantic.undeclared_requirement`); other planners may reject the file, so declare what you use.

B.2. Syntax templates

The five templates are files of `examples/apB`. Each domain uses as many constructs of its level as fit in a page, and marks them with comments; each problem has a plan. They are not models to imitate, since a real model uses only the features it needs, but places to copy the syntax from.

Classical: PDDL 1.2

Types with a hierarchy, constants, equality, the logical conditions (`not`, `or`, `imply`, `exists`, `forall`) and conditional effects (`when`, here under a `forall`). The plan found has 15 actions: the robot takes the keycard before anything else, because the pharmacy is restricted.

```
;; 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)))
```

Listing B.1: Classical PDDL: the domain.

```
;; 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)))))
```

Listing B.2: Classical PDDL: a problem.

Numeric: PDDL 2.1, level 2

Functions, numeric preconditions, the five numeric effects, a numeric goal and a metric. The function `(total-cost)` is the one of `:action-costs`; here it is increased by the distance, a static function. Some planners, POPF and OPTIC among them, accept a numeric condition or effect only inside `(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)))
```

Listing B.3: Numeric PDDL: the domain.

Listing B.4: Numeric PDDL: a problem.

Temporal: PDDL 2.1, levels 3 and 4, and PDDL 2.2

Durative actions with a fixed duration (`pick`), a computed one (`move`) and one that the planner chooses (`charge`); conditions and effects at the start, at the end and over the whole action; a continuous effect; a derived predicate, defined by a recursive rule; and, in the problem, timed initial literals and the metric `(total-time)`. A temporal plan lists each action with its start time and its duration; Appendix A shows the one of this problem.

```
;; 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)))))
```

Listing B.5: Temporal PDDL: the domain.

```
;; 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)
```

```
    ; 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
```

Listing B.6: Temporal PDDL: a problem.

Preferences, constraints and object fluents: PDDL 3

A preference in a precondition and three in the goal, one of them under `forall`; constraints in the domain and in the problem, one of them as a preference; a metric that weighs the violated preferences; and an object fluent, `(location-of ?r)`, in the place of the predicate `robot-at`. The best plan has cost 6: it serves the meal, and accepts not to return to the dock (2), to enter the ICU (3) and to enter a quiet room (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))))
```

Listing B.7: PDDL 3: the domain.

Listing B.8: PDDL 3: a problem.

The operators for constraints (Chapter 15) are:

- without times: `at end`, `always`, `sometime`, `at-most-once`, `sometime-before` and `sometime-after`;
- with times: `within`, `always-within`, `hold-during` and `hold-after`.

Hybrid: PDDL+

Actions, processes and events. A process runs while its condition holds and has continuous effects; an event fires at the instant its condition becomes true. In the plan found the robot takes the sample out, plugs itself in for 2.2 time units, unplugs and goes to the lab.

```
;; 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)
    (robot-at r1 dock) (powered r1)
```

Listing B.9: PDDL+: the domain.

```
    (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)))))
```

Listing B.10: PDDL+: a problem.

B.3. Which planners support what

Table B.3 comes from the documentation of Kairos, the file docs/pddl-support.md. It was obtained by running each planner, not by reading its manual: one small, solvable domain and problem per feature, ten seconds per planner, and every plan checked with `kairos validate`. A check mark means that the planner returned a valid plan. The planners are Kairos, OPTIC (Benton et al., 2012), TFD, POPF (Coles et al., 2010), LPG-td, ENHSP (Scala et al., 2016), Metric-FF (Hoffmann, 2003) and LAMA (Richter & Westphal, 2010); the last seven in the versions distributed by planutils 0.10.25, tested in October 2026.

Feature	*Kairos*	*OPTIC*	*TFD*	*POPF*	*LPG-td*	*ENHSP*	*Metric-FF*	*LAMA*
STRIPS and types	✓	✓	·	✓	✓	✓	✓	✓
`either` types	✓	✓	·	✓	✓	·	✓	·
Domain constants	✓	✓	·	✓	✓	✓	✓	✓
Negative preconditions	✓	·	·	·	✓	✓	✓	✓
Equality	✓	✓	·	✓	✓	✓	✓	✓
Disjunctive preconditions	✓	·	·	·	✓	✓	✓	✓
Quantified preconditions	✓	✓	·	✓	·	✓	·	✓
Conditional effects	✓	·	·	·	·	✓	·	✓
Negative goals	✓	·	·	·	✓	✓	✓	✓
Derived predicates (recursive)	✓	·	·	·	✓	·	·	✓
Numeric fluents	✓	✓	·	✓	✓	!	✓	·
Action costs (`total-cost`)	✓	✓	·	✓	·	·	·	✓
Metric over numeric fluents	✓	✓	·	✓	·	✓	·	·
Durative actions	✓	✓	✓	✓	✓	·	·	·
Required concurrency	✓	✓	✓	✓	·	·	·	·
Duration inequalities	✓	✓	·	✓	·	·	·	·
Continuous effects (`#t`)	✓	·	·	✓	·	·	·	·
Timed initial literals	✓	✓	·	✓	·	·	·	·
Preferences (PDDL 3)	✓	✓	·	✓	·	✓	·	·
Trajectory constraints (PDDL 3)	✓	✓	·	!	·	·	·	·
Object fluents (PDDL 3.1)	✓	·	·	·	·	·	·	·
Processes (PDDL+)	✓	·	·	·	·	·	·	·

Table B.3.: PDDL features by planner. ✓: a valid plan. ·: an error, an unsupported feature or no plan in ten seconds. !: a plan that fails validation. Duration inequalities were tested with `?duration` in the effects.

The table tests one small problem per feature, so a check mark says that the planner reads the construct and plans with it, not how well it does on large problems. The notes of the same document explain some cells:

- **TFD** plans only domains with durative actions, so every classical row fails.
- **OPTIC and POPF** reject a numeric effect or condition that is not inside `(and ...)`: `:effect (increase (fuel) 4)` is a syntax error for them, and `:effect (and (increase (fuel) 4))` is fine. They have no negative preconditions, disjunctions or conditional effects.

- **ENHSP** returned an invalid plan on the numeric test, and does not accept `either`.
- **Metric-FF** was run with its classical search; its default search needs a metric.
- **LAMA**, a configuration of Fast Downward, rejects `either` in action parameters and has no numeric fluents.
- **LPG-td** has no quantified preconditions nor conditional effects.

For the modeler the table is the criterion Portability of the MODEL method in one picture: plain STRIPS with types, constants and equality runs on every planner of the table that accepts actions without duration, and almost every feature added to it removes planners from the list.

C. Solutions to the exercises

This appendix gives the solutions of the exercises proposed at the end of each chapter, explained. Each solution carries the number of its exercise (chapter, section and number): the solution of Exercise 5.2.1 is in the section of Chapter 5, under the same number. The PDDL of the solutions is in the `solutions` folder of each chapter's examples, tested like every other file of the book.

Many exercises ask for a model, and a model can be written in more than one way. A solution that differs from the one given here is not wrong for that reason: check it with the test set of step E of the MODEL method, and compare it with the one printed using the five criteria of step L.

C.1 Solutions for Chapter 1 504
C.2 Solutions for Chapter 2 508
C.3 Solutions for Chapter 3 515
C.4 Solutions for Chapter 4 521
C.5 Solutions for Chapter 5 527
C.6 Solutions for Chapter 6 532
C.7 Solutions for Chapter 7 537
C.8 Solutions for Chapter 8 542
C.9 Solutions for Chapter 9 546
C.10 Solutions for Chapter 10 549
C.11 Solutions for Chapter 11 554
C.12 Solutions for Chapter 12 560
C.13 Solutions for Chapter 13 565
C.14 Solutions for Chapter 14 570
C.15 Solutions for Chapter 15 573
C.16 Solutions for Chapter 16 576
C.17 Solutions for Chapter 17 579
C.18 Solutions for Chapter 18 585
C.19 Solutions for Chapter 19 591
C.20 Solutions for Chapter 20 597

C.1. Solutions for Chapter 1

Section 1.1 — Deciding before acting

Solution 1.1.1. (a) No: the thermostat reacts to the current temperature, and no sequence is involved. (b) Yes: the cars block one another, so the order of the moves matters and a move that looks useless (pulling a car away from the exit) can be necessary. (c) No, in the sense of the chapter: sorting is solved by a fixed algorithm that works for every list; nothing has to be searched for each new input. (d) Yes, a small one: what goes in first ends at the bottom, so the actions interact and the order must be decided beforehand. (e) No: it is a reaction. The test is always the same: does the right action now depend on the actions that will follow it?

Solution 1.1.2. Delivering the medicines first: dock to lobby, lobby to pharmacy, pick `meds1`, pharmacy to lobby, lobby to corridor, corridor to ward B, drop `meds1`, ward B to ICU, pick `sample1`, ICU to ward B, ward B to corridor, corridor to lab, drop `sample1`. That is 13 steps: 9 moves, 2 picks and 2 drops. In the other order the robot goes to the ICU (4 moves), picks, goes to the lab (3 moves), drops, goes to the pharmacy (3 moves), picks, goes to ward B (3 moves) and drops: 17 steps. The first order is better because ward B is on the way to the ICU: the first delivery ends one room away from where the second one starts. The two requests are independent as goals, but not as plans, and a planner has to consider them together to see it.

Section 1.2 — A general solver plus a model

Solution 1.2.1. One possible answer for each. (a) Tea. State: whether the kettle has water, whether the water is hot, where the tea bag is, whether the cup holds water or tea. Actions: fill the kettle (needs: kettle empty; change: kettle has water); boil (needs: kettle has water; change: water is hot); pour (needs: hot water, cup with the bag in it; change: the cup holds tea). Goal: the cup holds tea. (b) Elevator. State: the floor of the cabin, whether the doors are open, who is inside, on which floor each waiting person is. Actions: go up or down one floor (needs: doors closed); open and close the doors; a

person boards or leaves (needs: doors open, cabin and person on the same floor). Goal: every person is on the floor they asked for. (c) 8-puzzle. State: which tile is in each of the nine positions, one of them being empty. Action: slide a tile into the empty position (needs: the tile is next to it; change: the tile and the empty position swap). Goal: every tile in its place. In all three, notice what the state leaves out (the brand of the tea, the weight of the people): only what is needed to decide.

Solution 1.2.2. (a) The situation: a new place and its connections. (b) The situation: a new goal. (c) The description of what the robot can do: the rule "one item at a time" is part of the actions. (d) The situation: one more robot exists; what any robot can do does not change. (e) The planner, or the computer it runs on; the model stays as it is (although Chapter 3 will show that a better model can also make planning faster). In no case does anybody write a new sequence of steps.

Solution 1.2.3. Eight. The item must be in ward A, so the robot does not carry it, and the robot can be in any of the eight places. The whole problem has 72 states (the item in one of 8 places with the robot in one of 8, plus the robot in one of 8 places carrying the item), and the goal selects 8 of them. A goal is a set of states, usually a large one.

Section 1.3 — The classical assumptions

Solution 1.3.1. (a) Instantaneous actions; Part IV. (b) Static: the world changes at a known time whatever the robot does; Part IV, Chapter 13. (c) Deterministic; not lifted in PDDL, handled by executing, observing and planning again (Chapter 20). (d) Finite: a number is not a true-or-false fact; Part III. (e) Fully known; not lifted in PDDL (Chapter 20). (f) Instantaneous actions again: with one action after another nothing ever happens at the same time; Part IV.

Solution 1.3.2. Good enough: deciding which rooms the robot must visit and in which order to pick and drop, when the floor is small and any route is acceptable. In the plan of `courier-p01.pddl` the order of the steps is right however long each corridor is. Clearly bad: choosing between two robots, one of them next door and the other at the far end of a long building. For a model without distances both are "some moves away", and it may send the far one if the number of rooms crossed happens to be smaller. The same model is right for one decision and wrong for another: a model is judged against the decisions that are asked of it.

Section 1.4 — Planning and its alternatives

Solution 1.4.1. A program for one request:

```
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) Two items and two trays: the program needs a new case that visits both origins before the destination, a decision about which origin to visit first, and a rule for what to do when a request names three items. Each case is new code, and the cases combine. (b) Closed connections: `route` must now search the map of the day;

if no route exists the program needs a way to say so, and if the robot is caught half-way with an item, another rule. The program is turning, case by case, into a planner written for one problem. In a model, (a) changes one rule of the domain (how many items a robot may carry) and (b) removes some connections from the problem; the plans, including the answer "no plan", follow without more work.

Solution 1.4.2. (a) Hand-written behavior: few situations, all known, and the machine must be predictable. (b) Learning: nobody can write the rule that tells a ripe tomato from an unripe one in an image. (c) Planning: the rules are clear, the situations are countless, and the goals change all day. (d) Hand-coded search (or a solution computed once and stored): one fixed problem does not repay the effort of a general tool, although a planner would also solve it. Reasonable people can disagree on (d); what matters is the reason given.

Section 1.6 — Domain, problem and PDDL

Solution 1.6.1. Domain: (b) and (e), which are rules about what any robot can do. Problem: (a), (c), (d) and (f), which describe one floor, one day and one request. The map (a) is the case that surprises: it feels like a permanent rule, but it is a fact about this floor, and keeping it in the problem lets the same domain serve any building.

Solution 1.6.2. (a) PDDL 2.1: numbers. (b) PDDL 2.2: facts that change at known times (and PDDL 2.1 for the clock itself: actions need durations for "8:00" to mean anything). (c) PDDL 3.0: a preference. (d) PDDL 2.1: actions with duration that overlap. (e) PDDL+: the warming is a process that nobody starts, and the spoiling an event. (f) PDDL 3.0: a constraint on the whole plan. Requirement (f) can also be written in PDDL 1.2 as a condition of the move action; Chapter 15 compares both ways.

Section 1.7 — Meeting Kairos

Solution 1.7.1. The goal is the last line of the problem, and changing `ward-a` to `icu` there is enough. The result is the file `courier-s01.pddl`, in the folder `solutions` of `examples/ch01`. The prediction from the map: the same three steps to get the medicines (two moves and the pick), then pharmacy, lobby, corridor, ward B, ICU (four moves) and the drop: 8 steps.

```
$ 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)
```

Solution 1.7.2. The file is `solutions/courier-s02.pddl`. The planner answers that the problem has no 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)
```

Read from the top: the medicines get to ward A only by a drop; a drop needs a robot that carries them; carrying comes only from a pick; and a pick needs a robot that is *free*. The only way to become free is a drop, which needs carrying: a circle with no entry. The deleted text was the statement "robot `r1` carries nothing at the start". Without it the planner does not assume that the robot's tray is empty: it knows only what the problem says, and for it a robot that is not declared free can never pick. Chapter 2 gives this rule a name, the closed world assumption.

Solution 1.7.3. A valid plan is in `solutions/courier-p02.plan`: it is the plan that Kairos printed in the chapter, one step per line. Any plan that respects the rules is accepted, also a longer one that wanders. The file `courier-p02-wrong.plan` is the same plan with its third and fourth steps swapped:

```
$ 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
```

The message gives the file and the line (5, because two comment lines come first), the step of the plan (the third) and its time (steps are numbered from time 0, one unit each), and the reason: the pick has a precondition, the robot must be in the ICU, which is false at that moment. The validator stops at the first step that cannot be done; what comes after it is not checked.

Section 1.8 — The MODEL method: five steps and five criteria

Solution 1.8.1. One possible card. *Story*: a robot vacuum cleans the rooms of a flat and goes back to its base. *Agents*: the vacuum, alone. *Success*: every room is clean and the robot is at its base. *Scope*, in: the rooms, which ones are connected, which ones are clean, where the robot is. Out: the battery, the dust bin, furniture and the exact path inside a room, people and pets, closed doors, how dirty each room is. *Assumptions*: the classical ones; a cleaned room stays clean. A bad plan from leaving out the battery: in a large flat the plan cleans every room in one tour, and the real robot stops in the fourth room with an empty battery, far from its base. If that can happen, the battery belongs in the scope.

Solution 1.8.2. (a) Step M: it is a line of the Scope. (b) Step E: it is one of the four tests. (c) Step D. (d) Step O. (e) Step L: an alternative

encoding, tried and compared. (f) Step E: every plan of the test set is read.

Solution 1.8.3. *Fidelity*: poor for most requests. A plan "move to the ward side, drop" does not say which rooms to cross or where on the ward side the item ends; a request for ward A and a request for the ICU become the same request. It also allows impossible plans: it picks an item in the pharmacy while the robot is at the dock, since both are the same place. *Size*: tiny, 2 places and 2 moves. *Guidance*: trivial, every plan has three or four steps. *Clarity*: the two names are easy, but nobody can read a delivery out of the plan. *Portability*: classical. Verdict: it is good enough only for a question such as "how many times must the robot cross the lobby today?", where the two sides are all that matters. For deliveries it loses fidelity at once. Too coarse and too fine both fail; the right level is set by the decisions that are needed, as the verdict of the chapter said.

C.2. Solutions for Chapter 2

The files of these solutions are in `examples/ch02/solutions`; the commands are run from `examples/ch02`.

Section 2.1 — Lists inside lists: the shape of PDDL

Solution 2.1.1. (a) A closing parenthesis is missing: `(and (on desk) (off hall))`. (b) The comma is not a separator, the space is: `(on desk)`. (c) The `and` and its parts must be a list of their own: `(:goal (and (on desk) (off hall)))`. (d) The semicolon starts a comment, so the planner reads `(on desk)` and never sees the rest of the line; the intended text was `((on desk) (off hall))` or, more likely, `(and (on desk) (off hall))`. This last mistake is the worst of the four because it may not be an error at all: a fact has silently disappeared. The tree of (a) has the root `and` with two children, `on` and `off`; `desk` hangs from the first and `hall` from the second. It is Figure 2.1 without its top node.

Section 2.2 — Predicates, objects and facts

Solution 2.2.1. (a) Each of the six facts is in the set or not: $2^6 = 64$ sets. (b) A sensible state has, for each lamp, exactly one of its two facts: $2^3 = 8$ states. (c) {`(on hall)`, `(off hall)`, `(off desk)`}: the hall lamp is lit and dark, and the porch lamp is neither. The `:init` of a problem produces it if it lists both facts of the hall lamp and forgets the porch lamp. The language accepts all 64 sets; keeping the model inside the 8 sensible ones is the job of the initial state and of the actions.

Solution 2.2.2. (a) False: it is not in the state (and it should not be, since the robot carries the sample). (b) False: a carried item is in no room. (c) True: it is one of the static facts of the map. (d) False: no such connection is listed. (e) False. None is unknown: under the closed world assumption every fact is true (it is in the state) or false (it is not).

Section 2.3 — Actions: preconditions and effects

Solution 2.3.1. (a) `(move r1 lobby lab)`: not applicable, `(connected lobby lab)` is false. `(pick r1 meds1 lobby)`: applicable. `(pick r1 sample1 lab)`: not applicable, `(robot-at r1 lab)` is false. `(drop r1 meds1 lobby)`: not applicable, `(carrying r1 meds1)` is false. `(move r1 lobby dock)`: applicable. (b) After the pick: `(robot-at r1 lobby)`, `(carrying r1 meds1)`, `(item-at sample1 lab)`; the pick deleted `(free r1)` and `(item-at meds1 lobby)`. After the move: `(robot-at r1 corridor)`, `(carrying r1 meds1)`, `(item-at sample1 lab)`.

Solution 2.3.2. The effect forgets `(not (item-at ?i ?l))`. After the pick the state has `(carrying r1 meds1)` and still `(item-at meds1 pharmacy)`: the box is on the robot and on the shelf. After the move and the drop it has `(item-at meds1 pharmacy)` and `(item-at meds1 lobby)`: the box has been copied. None of the four tests of the chapter notices by its result, since the plans keep their lengths; this is a forgotten delete that does not shorten anything in them. Two things catch it. Reading the plan on the state, as in Table 2.3, shows the extra fact at once. And a new test of the kind "must have no plan" aimed at the rule "an item is in one place": a goal that asks for `(and (item-at meds1 lobby) (item-at meds1 pharmacy))` must be unsolvable, and with this domain it has a plan of three steps. A test set grows with every rule of the model card.

Section 2.4 — The problem: objects, initial state and goal

Solution 2.4.1. The file is `switch-s01.pddl`: the goal becomes `(and (on hall) (on desk) (on porch))`. The hall lamp is already lit, so two steps are enough, in any order:

```
$ kairos switch-domain.pddl solutions/switch-s01.pddl
; cost = 2
(switch-on porch)
(switch-on desk)
```

A fact of the goal that is already true costs nothing, as long as no action of the plan deletes it.

Solution 2.4.2. The file is `switch-s02.pddl`. There is no 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)
```

Only `switch-on` adds `(on porch)`, and it needs `(off porch)`. That fact is not in the initial state, so it is false, and the only action that adds it, `switch-off`, needs `(on porch)`: a circle. For us a lamp that is not on is off. For the planner `on` and `off` are unrelated, and a lamp with neither fact is out of the reach of every action. The initial state must be complete.

Solution 2.4.3. The power is one more thing that is on or off, and it is not a property of any lamp: two predicates without parameters, and an action without parameters.

```
6   (:predicates
7     (on ?l) (off ?l)
8     (power-on) (power-off))   ; predicates without parameters
```

```
9
10  (:action power-up
11    :parameters ()
12    :precondition (power-off)
13    :effect (and (power-on) (not (power-off))))
14
15  (:action switch-on
16    :parameters (?l)
17    :precondition (and (off ?l) (power-on))
18    :effect (and (on ?l) (not (off ?l))))
```

`switch-off` is unchanged. The problem `switch-power-p01.pddl` starts with `(power-off)` and three dark lamps:

```
$ kairos solutions/switch-power-domain.pddl \
    solutions/switch-power-p01.pddl
; cost = 3
(power-up)
(switch-on porch)
(switch-on desk)
```

`power-up` is in the plan although no goal mentions the power: the planner found that the lamps need it. `(power-on)` is a precondition of `switch-on` and no effect deletes it, so one `power-up` serves both lamps.

Section 2.5 — Modeling the courier world

Solution 2.5.1. The file is `courier-s01.pddl`. The best order fetches the meal first: corridor to lobby, pick, lobby to corridor to ward B, drop (5 steps); then ward B to corridor to lab, pick, lab to corridor to ward B to ICU, drop (7 steps). Twelve steps. Starting with the sample takes 13, because the robot crosses ward B with an empty tray and must come back to it. Kairos prints both, the plan of 13 steps first and then the better one:

```
$ 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)
```

Solution 2.5.2. Only the problem changes: the map is data. The fact `(connected ward-a corridor)` is removed and `(connected corridor ward-a)` stays (`courier-s02.pddl`). The domain needs nothing, because `connected` always had a direction. With the goal `(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)
```

Each half of the goal can be reached, and the note does not name an unreachable fact. But the medicines get into ward A only if the robot goes in, and a robot that went in never comes out: the two halves cannot hold together. Ward A is a *dead end*, a state from which the goal cannot be reached any more. With the goal `(item-at meds1 ward-a)` alone the problem has its usual plan of 7 steps, and the robot stays in the ward for ever.

Solution 2.5.3. The row of the action table. Parameters: the giver, the taker, the item, the location. Applies when: both robots are in the location, the giver carries the item, the taker is free. Changes, property by property: where the robots are, no change; where the item lies, no change (it lay nowhere and still lies nowhere); what is carried, the taker carries the item and the giver no longer does; free, the giver becomes free and the taker stops being free.

```
  (:action hand-over
    :parameters (?giver ?taker ?i ?l)
    :precondition (and (robot ?giver) (robot ?taker) (item ?i)
                       (location ?l)
                       (robot-at ?giver ?l) (robot-at ?taker ?l)
                       (carrying ?giver ?i) (free ?taker))
    :effect (and (not (carrying ?giver ?i)) (free ?giver)
                 (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)
```

Without the new action the plan would need a drop by `r1` and a pick by `r2`: three steps. The planner uses `hand-over` because it gives a shorter plan, and for no other reason. Notice also that a robot cannot hand an item to itself: the action would need `(carrying r1 meds1)` and `(free r1)` at once, and the actions never make both true.

Section 2.6 — Exercising the model

Solution 2.6.1. `courier-p02-long.plan` takes a walk to the dock and back before doing the work: seven steps, all of them applicable, and the goal holds at the end.

```
$ 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
```

A valid plan need not be a good one: validity says that the model allows the plan and that it reaches the goal, nothing about its length. In the second plan the move is fine (a robot may always walk) and

the failure comes at the drop, the second step, whose precondition `(carrying r1 meds1)` no action made true.

Solution 2.6.2. *Trivial*: one dark lamp that must be lit; expected, one step. *Typical*: `switch-p01.pddl`; expected, two steps. *No plan*: a lamp that the initial state forgets, as in Solution 2.4.2; expected, unsolvable. *Cheat*: the rule of this world is that a lamp is lit or dark and never both, so we ask for both (`switch-s03.pddl`, goal `(and (on desk) (off desk))`); expected, unsolvable.

```
$ 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)
```

If `switch-on` forgot its delete effect this last test would find a plan of one step, and it is the only one of the four that would notice.

Section 2.7 — Two classics: blocks world and gripper

Solution 2.7.1. The file is `blocks-s01.pddl`. Every block except the last must be taken from the old tower and put on the new one. `d` goes to the table (unstack, put-down), `c` and `b` go from tower to tower (unstack, stack, twice), and `a` is already on the table (pick-up, stack): 8 steps.

```
$ 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)
```

Solution 2.7.2. (a) If stacking on `?y` does not delete `(clear ?y)`, a second block can be stacked on the same one: two blocks directly on a third. (b) The goal `(and (on a c) (on b c))` with three blocks on the table. With the wrong domain (`blocks-noclear-domain.pddl`) it has a plan; with the right one (`blocks-s02.pddl`) it has none.

```
$ 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) A cheat test: it asks for a state that breaks a rule of the world (one block on top of each block) and expects no plan. The typical problem `blocks-p01.pddl` gives the same six steps with both domains and notices nothing.

Solution 2.7.3. *Model card.* Story: a ferry takes cars between ports. Success: each car waits in its destination port. Scope and assumptions: one

ferry, one car on board at most, every port reachable from every other, no times or distances. *State table.* Port and car, type predicates, static. Where the ferry is, dynamic, `(ferry-at ?p)`. Where a car waits, dynamic, `(car-at ?c ?p)`. Which car is on board, dynamic, `(on-board ?c)`. The ferry carries no car, dynamic, `(empty-ferry)`. There is one ferry, so it is not an object, as the robot of gripper. *Action table.* `sail`: the ferry is in the origin; it is then in the destination and not in the origin. `board`: car and ferry in the same port, ferry empty; the car is on board, no longer in the port, and the ferry is no longer empty. `debark`: the car is on board and the ferry in the port; the mirror image.

```
  (:action sail
    :parameters (?from ?to)
    :precondition (and (port ?from) (port ?to) (ferry-at ?from))
    :effect (and (ferry-at ?to) (not (ferry-at ?from))))

  (:action board
    :parameters (?c ?p)
    :precondition (and (car ?c) (port ?p) (car-at ?c ?p)
                       (ferry-at ?p) (empty-ferry))
    :effect (and (on-board ?c)
                 (not (car-at ?c ?p)) (not (empty-ferry))))

  (:action debark
    :parameters (?c ?p)
    :precondition (and (car ?c) (port ?p) (on-board ?c)
                       (ferry-at ?p))
    :effect (and (car-at ?c ?p) (empty-ferry)
                 (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)
```

The ferry never sails empty: each crossing takes a car. For step E the test set would add a cheat test, two cars in the same port for the same destination, which must take three crossings and not one.

Section 2.8 — The MODEL method: the first full pass

Solution 2.8.1. Blocks: the five predicates (`on`, `on-table`, `clear`, `holding`, `hand-empty`) all appear in effects; all are dynamic and nothing is static. Gripper: `room`, `ball` and `gripper` are static; `at-robby`, `at`, `empty` and `holds` are dynamic. `(unstack ?x ?y)`, property by property: `on`, false for `?x` on `?y`; `on-table`, untouched; `clear`, true for `?y` and false for `?x`; `holding`, true for `?x`; `hand-empty`, false. Four of the five properties change, in five effects: the listing has exactly those five.

Solution 2.8.2. The domain without a hand:

```
  (: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)
```

Fidelity: both produce only real arrangements of blocks. The model without a hand cannot speak about the hand: no goal or condition can mention a held block, and if the world had two hands the model could not say so. *Size*: for the three blocks of the problem, `kairos -v` reports 24 ground actions with four actions and 45 without a hand. The action `move` has three parameters, so it alone gives $3^3 = 27$ ground actions, and with n blocks n^3, against n^2 for `stack` and `unstack`. (The 27 include nonsense such as `(move a a a)`, which is never applicable; STRIPS cannot say that two parameters must differ, and Chapter 4 can.) *Guidance*: plans are half as long, 3 steps against 6, and the states in which the hand holds a block do not exist. *Clarity*: both read well; the one without a hand is shorter. *Portability*: both are `:strips`. Verdict: without a hand when only the arrangement matters and the number of blocks is small; four actions when the hand is part of the story, or when the blocks are many, since the cube of their number grows fast. The same trade as encodings A and B of the courier world: merging two actions into one shortens plans and adds parameters.

Section 2.9 — Common mistakes

Solution 2.9.1. The effect of `drop` does not add `(free ?r)`. After its first drop the robot is not free and nothing will make it free again, so it can never pick a second time. (a) `courier-p01`, `courier-p02` and `courier-p04` still give the right answers: the first two need a single pick, and the third must be unsolvable and is. `courier-p03` and `courier-p05` need two picks and fail. (b) The planner answers that they have no plan (the files are `courier-nofreeadd-domain.pddl` and `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)
```

Eleven states: the robot's world ends right after its first delivery. (c) A missing precondition makes an action applicable more often than in the world: the model allows too much, and the planner finds plans that are too short. A missing add effect takes away something that the world gives back: the model allows too little,

and plans disappear. The first kind of mistake is found by problems that must have no plan or must have long ones; the second by typical problems that must have a plan. That is why the test set needs both.

C.3. Solutions for Chapter 3

The files of these solutions are in `examples/ch03/solutions`. The numbers were obtained with `--threads 1`.

Section 3.1 — What a model means

Solution 3.1.1. Without the static facts, the state s_4 has two facts: `(robot-at r1 lobby)` and `(carrying r1 meds1)`. The fact `(free r1)` is not in it, and neither is any `item-at` fact: a carried item is in no room. Three ground actions are applicable:

- `(move r1 lobby pharmacy)` replaces `(robot-at r1 lobby)` by `(robot-at r1 pharmacy)`; the robot still carries the medicines. This is s_3.
- `(move r1 lobby corridor)` replaces it by `(robot-at r1 corridor)`. This is s_5.
- `(drop r1 meds1 lobby)` deletes `(carrying r1 meds1)` and adds `(item-at meds1 lobby)` and `(free r1)`; the robot stays in the lobby. This is s_6, a state with three facts.

No pick is applicable (the robot is not free) and no other drop (the robot is in the lobby).

Solution 3.1.2. Any detour will do. The file `courier-p01-long.plan` starts by going to the corridor and coming back:

```
(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` answers `plan valid, makespan 7`. A longer plan is a plan: the definition asks for applicable actions and a final state that satisfies the goal, and says nothing about length. In Figure 3.2 it is a path that leaves s_0, returns to it and then follows the short path. Preferring short plans is a separate matter, the one of Section 3.6.

Solution 3.1.3. The first action is applicable in s_0 and leads to s_1. The second one, the drop, needs `(carrying r1 meds1)`, which is not in s_1: the sequence is rejected at step 2, and the third action is not even looked at. Even with the drop and the pick in the right order the sequence would not be a plan: it would be applicable to the end, and fail on the goal. The validator says (the 3 after the file name is the line of the file, which starts with a comment):

```
$ 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
```

Section 3.2 — Grounding: from variables to objects

Solution 3.2.1. Atoms: `robot-at` for 1 robot and 8 rooms (8), `item-at` for 2 items and 8 rooms (16), `carrying` for 1 robot and 2 items (2), `free` (1): 27. Ground actions: 14 moves (one per `connected` fact), $1 \times 2 \times 8 = 16$ picks and 16 drops: 46. The `[ground]` line of `kairos -v` says `27 atoms` and `46 actions`.

Solution 3.2.2. Number the discs from the smallest, 1, to the largest, n. Disc i may rest on the $n - i$ larger discs and on the 3 pegs: $n - i + 3$ things. Both `?from` and `?to` range over them, so disc i has $(n-i+3)^2$ instances, and the total is

$$\sum_{i=1}^{n} (n - i + 3)^2 = 3^2 + 4^2 + \cdots + (n + 2)^2.$$

For 4 discs: $9+16+25+36 = 86$. For 6 discs: $86+49+64 = 199$. Both are what Kairos reports. For 2 discs the formula gives $9 + 16 = 25$. The instances with `?from` equal to `?to` can never be applied (the disc would have to be on a thing that is clear): there are 4 for `d1` and 3 for `d2`, so 7 of the 25. The other 18 are each applicable in some reachable state.

Solution 3.2.3. The domain is `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))))
```

Listing C.1: Gripper in STRIPS without types.

Estimate for `gripper-p01.pddl` (2 rooms, 2 grippers, 4 balls). `move`: 2 choices for each room, 4 instances; nothing in STRIPS as we know it says that the two rooms differ, so `(move rooma rooma)` is among them, and it is even applicable (it does nothing). `pick`: $4 \times 2 \times 2 = 16$. `drop`: 16. Total 36. Atoms: `at-robot` 2, `at` 8, `free` 2, `holding` 8: 20.

Kairos reports `20 atoms` and `36 actions`, a first plan of 13 actions and a best one of 11: five actions for each pair of balls (two picks, a move, two drops) and one move back between the two pairs.

Section 3.3 — Why the space explodes

Solution 3.3.1. The two robots are each in one of 8 rooms: $8^2 = 64$ ways. For the four items there are three cases. Nobody carries anything: $8^4 = 4\,096$ ways. One robot carries one item: 2 robots times 4 items, and the other three items in rooms: $8 \cdot 8^3 = 4\,096$. Both carry: 4 items for the first robot, 3 for the second, the other two in rooms: $12 \cdot 8^2 = 768$. Total: $64 \cdot (4\,096 + 4\,096 + 768) = 573\,440$.
With three robots: $8^3 = 512$ ways to place them, and $8^4 + (3 \cdot 4) \cdot 8^3 + (3 \cdot 4 \cdot 3) \cdot 8^2 + (4 \cdot 3 \cdot 2) \cdot 8 = 4\,096 + 6\,144 + 2\,304 + 192 = 12\,736$ ways to place the items (zero, one, two or three robots carrying). Total: 6 520 832 states. One more robot, eleven times more states.

Solution 3.3.2. The 4×4 board has 24 pairs of neighbor cells (4×3 in rows and as many in columns), so 48 `adjacent` facts and $15 \times 48 = 720$ ground actions. Atoms: `tile-at` for 15 tiles and 16 cells (240) plus 16 `empty`: 256. Reachable states: $16!/2 = 10\,461\,394\,944\,000$, about 10^{13}. At 52 bytes each they would take about 5.4×10^{14} bytes: more than 500 terabytes. The model is four times larger than that of the 8-puzzle; the state space is 58 million times larger. No planner proves anything about the 15-puzzle by exhausting it; a first plan is another matter, and comes from the heuristic.

Section 3.4 — Forward search

Solution 3.4.1. Name a state by the pegs of the two discs: (a, b) means `d1` on peg a and `d2` on peg b (when $a = b$, `d1` is on top of `d2`). There are $3 \times 3 = 9$ states. The small disc can always go to either of the other two pegs; the large one can move only when the small one is on a different peg, and only to the peg where the small one is not. So every state has two or three successors. The start is $(1, 1)$ and the goal $(3, 3)$.
Breadth-first search, with the loop of Section 3.4: it expands $(1, 1)$ and finds $(2, 1)$ and $(3, 1)$. It expands $(2, 1)$ and finds $(2, 3)$; it expands $(3, 1)$ and finds $(3, 2)$. It expands $(2, 3)$ and finds $(1, 3)$ and $(3, 3)$; it expands $(3, 2)$ and finds $(1, 2)$ and $(2, 2)$. It expands $(1, 3)$, nothing new. The next state out of the list is $(3, 3)$: the goal. Six states were expanded and all nine were met. The plan is the path $(1, 1), (2, 1), (2, 3), (3, 3)$, three moves, which is what Kairos prints for `hanoi-p01`:

```
(move d1 d2 peg2)
(move d2 peg1 peg3)
(move d1 peg2 d2)
```

Section 3.5 — Heuristics: asking a simpler problem

Solution 3.5.1. Forward: at the start the robot is in the corridor and free, and the medicines are in the pharmacy. A move adds the robot in the lobby; another adds it in the pharmacy; the pick adds `carrying`; a drop adds the goal fact. Backward, the relaxed plan is

```
(move r1 corridor lobby)
(move r1 lobby pharmacy)
```

```
(pick r1 meds1 pharmacy)
(drop r1 meds1 corridor)     ; the robot never left the corridor
```

Four actions; the true distance is 6. The relaxation forgets the whole way back, two moves, because `(robot-at r1 corridor)` was true at the start and nothing deletes it. This is the state where the heuristic is most wrong in Figure 3.6, and the one where the robot stands furthest from where it has to go.

Solution 3.5.2. In the initial state only `d1`, `peg2` and `peg3` are clear. The goal facts `(on d1 d2)`, `(on d2 d3)` and so on, up to the one before the last, are true from the start, and in the relaxation they stay true. The only goal fact missing is the largest disc on `peg3`. To move disc n it must be clear; it becomes clear, and stays so, when disc $n-1$ is moved away, which needs disc $n-1$ clear, and so on down to `d1`. So the relaxed plan moves each disc exactly once, from the smallest to the largest: n actions. Nothing ever has to be put back, since nothing was ever really taken away. With `--debug=search`, `hanoi-p04` (6 discs) prints `initial h: ff 6.000000`; its shortest plan has 63 moves.

Solution 3.5.3. With deletes ignored, a tile that slides is afterwards in both cells, and the cell it left is empty *and* still holds the tile. Empty cells only accumulate. So after a few slides around the hole every cell is empty, and each tile can walk to its goal cell by the shortest way, as if the other tiles were not there. A relaxed plan is then little more than the sum of the distances of the tiles to their places: the tiles never block one another. The estimate is too low, since in the real puzzle the tiles spend most moves getting out of each other's way. For `puzzle-p02` Kairos prints `initial h: ff 22.000000, lmcount 13.000000`; the best plan has 31 slides.

Section 3.6 — A plan, a better plan, the best plan

Solution 3.6.1. With `--no-anytime`: a plan of 357 moves, and the status `solved (no improvement)`: the search was told not to improve. With `--target-cost 300`: the first plan (357) is not good enough, the search goes on, and the next plan it finds has 255 moves; the status is `solved (target reached)`. The target asked for 300 or less, and the planner does not stop *at* 300 but at the first plan that is good enough. With no option: the same two plans, and then `solved (proven optimal)`: 255 is 2^8-1, the minimum for 8 discs.

Solution 3.6.2. The problem is `puzzle-p04.pddl`; its `:init` ends with

```
   (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 finds a first plan of 10 slides and then one of 8, which it proves to be the best after meeting 520 states: far fewer than the 181 440 of the hard board. A proof of optimality only has to look where a plan shorter than the one found could still hide, and around a plan of 8 slides that region is small.

Section 3.7 — No plan, or no plan found

Solution 3.7.1. The file is `courier-busy.pddl`. The only action that adds `(free r1)` is `drop`, which needs the robot to carry something; the only one that makes it carry something is `pick`, which needs it free. Neither fact can become true first, even with deletes ignored, so

grounding proves that there is no plan and no search is needed. The note of Kairos, printed below, walks the circle once (drop, pick, drop, pick) and stops. With `-v`, the `[ground]` line reports 14 actions: the moves are all that is left. Forgetting `(free r1)` in the initial state is a classic of the closed world assumption: what is not said is false, so the robot starts with its hands full of nothing.

```
$ 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)
```

Solution 3.7.2. First, the status line and the exit code: after 30 seconds without a plan they are almost certainly `timeout (timeout)` and 11, which prove nothing; "impossible" needs `unsolvable` and 10. Second, the note above the status line and the `[ground]` line of `-v`: a number of ground actions far from the estimate points to a mistake in the model rather than to a hard problem. Then: give more time; try a smaller version of the same problem (fewer items, one robot) to see whether the model solves anything at all; and ask for less (`--no-anytime`), in case the time went into improving. If small versions are solved at once and the large one never is, the model may be right and badly guided: dead ends and symmetries are the suspects of Section 3.8.

Section 3.8 — What this means for the modeler

Solution 3.8.1. At most $2 \times 2 \times 4 \times 8 = 128$ instances, once the type predicates have done their work. Half of them, 64, have the same robot as `?from` and `?to`: a robot handing an item to itself. They are useless, and worse than useless if the effects are written carelessly (an effect that adds and deletes the same fact). With the STRIPS of Chapter 2 nothing in the domain can say that two parameters are different objects. Two ways out: the requirement `:equality`, in Chapter 4, or, staying in plain STRIPS, a static predicate `(different ?a ?b)` listed in the problem for every pair of distinct robots, and a precondition on it. The second is the same trick as `connected`: a static fact that prunes during grounding.

Solution 3.8.2. The runs give the numbers of the table: 33, 464, 4 670 and 43 595 states for the proof with one to four boxes. The problem `courier-p04`, with three different deliveries, needs 5 054: about the same as three identical boxes. The conclusion is that the planner gains nothing from the boxes being identical. To a person, three equal boxes are a simpler problem than three different errands; to a planner that names each object, they are the same kind of problem, with the added insult that most of the states it tells apart are the same situation. Symmetry is knowledge about the problem that this encoding does not give to the planner.

Solution 3.8.3. The robot can do nothing at all: no move starts in the ICU. Grounding keeps only actions that could become applicable, so it keeps none, and the goal fact is unreachable. The run below

shows zero ground actions and three atoms: the three facts of the initial state that are not static. The whole state space is one state. The last line of the note is less precise than the others: the pick is never applicable because the robot is never in the pharmacy.

```
$ 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)
...
```

Section 3.9 — The MODEL method: size and guidance in numbers

Solution 3.9.1. Estimate. Atoms: `robot-at` $2\times8 = 16$, `item-at` $4\times8 = 32$, `carrying` $2 \times 4 = 8$, `free` 2: 58. Ground actions: `go` $2 \times 56 = 112$, `pick` $2\times4\times8 = 64$, `drop` 64: 240. Kairos reports `58 atoms` and `240 actions`: the estimate holds.
The run gives plans of 18, 17 and 16 actions, and the proof for 16 after meeting 509 848 states, in about 3 seconds. Encoding A on `courier-p06` had 156 actions, a best plan of 27 and 544 039 states for the proof, in about the same time. The number of states is almost the same, as it must be: the two encodings describe the same situations (robots in rooms, items in rooms or in hands), and differ only in the arrows between them. C has 84 more ground actions and plans that are eleven actions shorter. The proof costs the same.

Solution 3.9.2. The goal must also say that the robot ends with empty hands. The solution, `courier-loaded-p02.pddl`, is the problem of the chapter with one more goal fact:

```
(:goal (and (item-at meds1 ward-a)
            (item-at sample1 lab)
            (item-at meal1 icu)
            (free r1))))
```

Now the plan has 19 actions, like that of encoding A, and ends with a drop. The new column B: Size, the same (36 atoms, 104 actions); Guidance, a first plan of 19 actions after 35 expansions and a proof after 5 054 states, the same number as A. The verdict does not change, it gets worse for B: with the repair it has no advantage left, and the repair is in the problem, where every future problem will have to remember it. With several robots `(free r1)` would not even be enough: every robot would have to be free. A model in which the natural goal is wrong is a model to avoid.

Solution 3.9.3. **Fidelity**: D still has `move`, so routes are in the plan for the walks with empty hands; but `deliver` jumps from room to room with the item, so the route *with* the item is not, and the robot is never seen carrying anything: no state has `carrying` true. Correct only where C is correct, and confusing where A is needed. **Size**: `deliver`

has four parameters; with R robots, I items and 56 reachable pairs it adds $56 \cdot R \cdot I$ ground actions (168 for one robot and three items) on top of those of A. It is the largest of the four. **Guidance**: plans are shorter than in A, since a delivery is one action plus the walk to the item; but `pick` and `drop` must stay in the domain if items are ever to be set down on the way, and then the planner has two ways of doing everything, which is symmetry of our own making. **Clarity**: worse; `deliver` repeats what `pick`, `move` and `drop` already say, and the two descriptions can drift apart. **Portability**: the same, plain STRIPS. D mixes two levels of detail in one model. Either the route matters, and then A; or it does not, and then C.

C.4. Solutions for Chapter 4

The files named in these solutions are in `examples/ch04/solutions/`.

Section 4.1 — What type predicates cost

Solution 4.1.1. `kairos check` answers `valid, plannable`: without types `robot-at` accepts any two objects, and `(robot-at dock r1)` is a legal fact. It says that the "robot" `dock` is in the "room" `r1`. The planner then proves that there is no 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)
```

The note reads the chain backwards from the goal. To have the item in ward A somebody must drop it there, which needs a robot in ward A; only `move` puts a robot in a room, and `move` is never applicable. Why not? Its preconditions ask for `(robot ?r)` and `(robot-at ?r ?from)` together. The only robot is `r1`, and no fact `robot-at` has `r1` as first argument. The note leads to `robot-at`, and from there the swapped fact is found by reading; a typed model points at it directly.

Solution 4.1.2. Untyped: 20 objects, three actions with three parameters each, so $3 \times 20^3 = 24000$ candidates. Applicable in some reachable state: a robot can move along each connection in each direction, $3 \times 30 = 90$ moves; any robot can pick or drop any item in any room (every room can be reached, so every item can end up anywhere), $3 \times 5 \times 12 = 180$ picks and 180 drops. Total 450. Typed: `move` has $3 \times 12 \times 12 = 432$ candidates, `pick` and `drop` 180 each: 792 in total, of which the same 450 are applicable. Types remove 23208 candidates without looking at any fact; the static predicate `connected` removes the remaining 342.

Section 4.2 — Types

Solution 4.2.1. The domain is `gripper-domain.pddl`, with three types:

```
(: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` mirrors `pick`.) There is one robot, so it needs no type and no parameter: its position is the predicate `at-robby` of one argument. The grippers, on the other hand, are objects, because `pick` must choose one. `gripper-p01.pddl` is solved with 9 actions: one trip with one ball and one with two, or the reverse. With 3 balls, 2 rooms and 2 grippers, `pick` has $3 \times 2 \times 2 = 12$ candidates, all of them applicable in some state. Without types the problem has 7 objects and `pick` $7^3 = 343$ candidates. Kairos grounds the typed problem to 28 actions: 12 picks, 12 drops and 4 moves. Four moves with two rooms means that `(move rooma rooma)` is among them: nothing separates `?from` and `?to`, a case for the inequality of Section 4.6.

Solution 4.2.2. `icu` comes after the last type of the list, so it has no type: it is a plain `object`. An `object` does not fit a parameter of type `location`, so every fact that uses `icu` as a room is rejected. In the usual map those are the two facts of the connection with ward 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
```

The message says "`'icu' is object`": that word is the hint that the declaration, not the fact, is wrong.

Solution 4.2.3. It is legal. `?a` and `?b` are robots: the type applies to all the untyped variables before it. `?c` and `?d` come after the last type and have none, so they are of type `object` and accept anything. The rewrite gives each of the first two its own type: `(?a - robot ?b - item ?c ?d)`.

Section 4.3 — Type hierarchies

Solution 4.3.1. (a) Legal: `porter1` is a porter, so a carrier; `meds1` is a medicine, so an item. It is not applicable in the initial state (the porter is in ward B and the medicines in the pharmacy), but that is

a matter of facts, not of types. (b) Not legal: the second parameter of `serve` is a `meal` and `meds1` is a `medicine`; they are siblings under `item`, and neither is above the other. (c) Not legal: the arguments are swapped, an item where a carrier is expected and a carrier where an item is expected. (d) Legal: the dock is a location like any other. The types cannot say that nobody eats in the dock; if that mattered, it would take a predicate. (e) Not legal: a meal is not a carrier.

Solution 4.3.2. The hierarchy has two trees. Staff: `keeper` and `vet` are subtypes of `staff` (both walk), and `senior` is a subtype of `keeper`, which is exactly "a senior keeper can do everything that a keeper does". Animals: `bigcat` and `herbivore` under `animal`, and the species under them.

```
(: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)))
```

Each parameter sits at the level of its rule: `feed` takes any `keeper` (seniors included) and a `herbivore`; `feed-bigcat` takes a `senior` and a `bigcat`; `treat` takes a `vet` and any `animal`. The species (`lion`, `zebra`...) pass the test for a type only weakly: no action tells a lion from a tiger. They are harmless and make the problems readable, but `leo - bigcat` would do. Kairos solves `zoo-p01.pddl` with 7 actions, in which the senior keeper feeds everybody (it is the only one who can feed the lion, and once it is walking, it is cheaper that it feeds the rest too) and the vet treats the giraffe. The plan with the plain keeper is rejected:

```
$ 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
```

Solution 4.3.3. With `?from ?to - place` in `fly`, the airplane lands anywhere. The plan for `logistics-p01` shrinks from 12 steps to 4: the airplane flies to the depot of Madrid, loads the package, flies to the depot of Paris and unloads. Fidelity suffers: the model now has plans that cannot be carried out. With `?t - vehicle` in `drive`, airplanes drive: the planner loads the package on the airplane at the depot, "drives" the airplane to the airport, flies, and drives it to the other depot. Again a loss of fidelity, of the kind that only

reading the plan reveals, because every step looks reasonable by its name. Typing a parameter too high is the typed version of a missing precondition.

Section 4.4 — A parameter of several types: `either`

Solution 4.4.1. With `either` (`courier-at-domain.pddl`) the predicate is declared as

```
(at ?x - (either carrier item) ?l - location)
```

and with a supertype (`courier-at-hier-domain.pddl`) the types gain one line and the predicate names it:

```
(:types
  location
  carrier item - locatable
  robot porter - carrier
  medicine sample meal - item)
...
(at ?x - locatable ?l - location)
```

The actions are the same in both files. Both solve the problem `courier-at-p01.pddl` with a plan of 13 steps. For the colleague, the version with `locatable`: every planner that reads types reads it, while `either` is rejected by some. Note what a single `at` costs in either version: `(at meds1 lobby)` and `(at r1 lobby)` now look alike, and an action that should apply to carriers only is protected by the type of its parameter and by nothing else.

Solution 4.4.2. A type has one supertype. "Can be borrowed" groups books, DVDs and audiobooks; "needs a player" groups DVDs and audiobooks. The second group is inside the first, so here a tree is still possible (`playable` under `loanable`), but the exercise asks for the general case, and it shows as soon as something that is not loanable needs a player too. The encoding of `library-player-domain.pddl` keeps the hierarchy for the main grouping and uses `either` for the one that cuts across:

```
(: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)))
```

Two actions are needed because, with what we know so far, a precondition cannot say "if it needs a player, then the member has one"; Chapter 5 gives the means. In `library-player-p01.pddl` Ana wants an audiobook, and the plan is to buy a player and then borrow it.

Section 4.5 — Constants

Solution 4.5.1. The first version (`courier-park-cheat-domain.pddl`) declares `(:constants dock - location)` and

```
(:action park
  :parameters (?r - robot)
  :precondition (and (robot-at ?r dock) (free ?r))
  :effect (parked ?r))
```

The problem `courier-park-p01.pddl` does not list `dock` in its objects. The 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)
```

The robot parks first and then does its work. The goal `(parked r1)` holds at the end, because nothing ever deletes it, but the robot is in ward A. The model lost the meaning of "parked": a parked robot does not move. The natural repair is a precondition "not parked" in `move`, which needs Chapter 5. With plain STRIPS, the repair is to make `park` delete what `move` needs (`courier-park-domain.pddl`):

```
  :effect (and (not (robot-at ?r dock)) (parked ?r))
```

A parked robot is no longer `robot-at` any room, so no `move`, `pick` or `drop` applies to it. The plan now has 11 steps and ends with the walk back to the dock and `(park r1)`, and the cheating plan (`courier-park-p01-early.plan`) is rejected by `kairos validate` at its second step. The price is in Clarity: a reader has to know that "parked" implies "in the dock", because the state no longer says where the robot is.

Solution 4.5.2. (a) If the table is an object at all, yes: every blocks problem has exactly one, and the actions that put a block on the table name it. The classic encoding avoids the object with a predicate `(ontable ?b)`, which is the same decision taken one step further. (b) No. No rule of logistics is about Madrid; it is part of a situation. (c) No, although they are in every problem: `pick` and `drop` treat both grippers alike and never name one. They could be declared as constants to save two words in each problem, and nothing would break, but the test (does a rule name it?) says that they are objects of the situation: a robot with three grippers is another problem, not another domain. (d) Yes, if every hospital of interest has one lab and `analyze` only happens there: `(item-at ?s lab)` in its precondition. If some hospitals have two labs, a static predicate and a parameter, as in Table 4.3.

Section 4.6 — Equality

Solution 4.6.1. The action, in `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))))
```

Without the first precondition (`courier-swap-noeq-domain.pddl`), `(swap r1 r1 meds1 meds1 lab)` is applicable: every precondition is about `r1` carrying `meds1` in the lab, twice. Its effects delete `(carrying r1 meds1)` and add the same fact. When an action deletes and adds one fact, the fact is true afterwards, so the step changes nothing; `kairos validate` accepts `courier-swap-p01-self.plan`, which starts with it, on the domain without the inequality and rejects it on the other. Self-swaps with two *different* items would need a robot that carries two items, which never happens. So here the missing inequality does not create impossible plans; it creates useless ground actions. On `courier-swap-p01.pddl` Kairos grounds 220 actions without the inequality and 156 with it: the 64 that disappear are the self-swaps, 2 robots by 2 items by 2 items by 8 rooms. Both domains find the same best plan of 9 steps, in which `r2` walks to the lab, the robots swap, and `r2` walks back with the medicines.

Solution 4.6.2. The file is `logistics-eq-domain.pddl`. Six actions disappear and 18 remain: each of the two trucks could "drive" from each of the two places of its city to the same place (4 actions), and the airplane could "fly" from each airport to itself (2 actions). None of them changes anything, so the plan is the same, with 12 steps.

Solution 4.6.3. Neither the plans nor the ground actions change: on `courier-p02` Kairos grounds 124 actions with or without it. A move from a room to itself needs `(connected l l)`, which no problem states, so those instances were already discarded. It would matter if the map had loops (a fact `(connected lobby lobby)` written by mistake or by a generator of maps), or in a world without a map, where any room is reachable from any other and `move` has no `connected` precondition: then, as in the gripper domain of Exercise 4.2.1, only the inequality separates the two rooms.

Section 4.7 — The MODEL method: kinds of things

Solution 4.7.1. Kinds, with the test "some predicate or action treats them differently":

Kind	Is a	Why
`place`		movers and pallets are in it
`shelf`, `station`	`place`	a pallet is packed only at a station
`mover`		it moves and carries a pallet
`forklift`, `cart`	`mover`	only a forklift reaches a high shelf
`pallet`		it is carried

"High" and "cold" are properties of places, not types: `(high ?s - shelf)` and `(cold ?s - station)` as static predicates. They could be subtypes (`high-shelf`), but two independent properties of places would soon ask for two supertypes. "Frozen" is a static property of pallets, `(frozen ?p - pallet)`; a subtype `frozen-pallet` is also defensible, since a pallet never thaws in this story. The rule "frozen goods only at a cold station" needs two unload actions with what we have (one for frozen pallets with `(cold ?s)` in its precondition, one for the others, told apart by type) or the implication of Chapter 5. The state table: where each mover is, where each pallet is or who carries it, whether a mover is free; static: the aisles, `high`, `cold`, `frozen`. No constant: no rule names a particular shelf or station.

Solution 4.7.2. *Fidelity*: both keep `r2` out of every plan, if no action accepts `broken-robot`; but the type must then sit outside `robot` in

the hierarchy, so `r2` cannot even appear in `(robot-at r2 dock)`, and the model cannot say where the broken robot stands (it may block a room). And a robot that breaks or is repaired *during* the plan cannot be modeled at all: types do not change. *Size*: the type version instantiates nothing for `r2`; the predicate version instantiates its actions and prunes them, since `(working r2)` is static and false. After grounding they are equal. *Guidance*: equal. *Clarity*: the predicate says what it means, in the initial state, where a reader looks for the situation; the type hides a fact about today in the declaration of the objects. *Portability*: both are plain typed STRIPS. Verdict: the predicate. "Out of order" is state, even if in this problem it happens not to change.

C.5. Solutions for Chapter 5

The files named in these solutions are in `examples/ch05/solutions/`, unless they were already used in the chapter.

Section 5.1 — Conditions about absence: negative preconditions

Solution 5.1.1. No fact: closed. `(open d)` alone: open. `(locked d)` alone: locked. Both: an open door with the lock engaged, which means nothing in this model. It cannot be reached. Only `open-door` adds `open`, and it requires `(not (locked ?d))`; no action adds `locked`. So if a door is not both in the initial state, it never is. The argument is an *invariant*: a property that holds at the start and that every action preserves. The domain `courier-doors-v0` of the chapter broke it, by letting `open-door` apply to a locked door.

Solution 5.1.2. `courier-doors-hands-domain.pddl` adds `(free ?r)` to the preconditions of `open-door` and `close-door`. Opening is no trouble: the robot opens the pharmacy door on its way in, with an empty gripper. Closing is. When it comes out, it holds the medicines. The plan of Kairos has 11 steps:

```
$ 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)
```

The robot delivers first and comes back to close the door with its gripper free. The plan most people predict (put the medicines down in the lobby, close the door, pick them up again) also has 11 steps: three extra actions in both cases. The planner is indifferent; the pharmacist, whose door stays open for a while, may not be. Saying so would take a constraint on the whole plan (Chapter 15).

Solution 5.1.3. In `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))` is essential: without it the robot could lock an open door, and the meaningless fourth combination of Exercise 5.1.1 would be reachable, because now an action adds `locked`. `(not (locked ?d))` only removes a useless action (locking what is locked). On `courier-doors-lock-p01.pddl` the plan has 10 steps: after leaving the pharmacy with the medicines the robot does `close-door` and then `lock`.

Section 5.2 — Negative goals

Solution 5.2.1. The goal of `courier-doors-p05.pddl` is `(and (item-at meds1 ward-a) (not (robot-at r1 ward-a)))`. The plan gains one step, `(move r1 ward-a corridor)`, and the robot ends in the corridor, in everybody's way. The nurse probably meant "go back to the dock". A negative goal says where the robot must not be and leaves the planner to choose the cheapest of all the other places. When there is a place where the robot *should* be, a positive goal says it better.

Solution 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
...
```

The plan is the one of `p01`. The door was closed before the line was written, because no fact said that it was open.

Section 5.3 — Alternatives in a condition: `or` and `imply`

Solution 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)))
```

A book does not need a player, so for a book the implication is true whatever the member has. In `library-imply-p01.pddl` the plan is `(buy-player ana)`, `(borrow bob book1)`, `(borrow ana audio1)`. Compared with Solution 4.4.2, the two actions became one, and "needs a player" became a fact of each problem and stopped being a union of types. A new kind of loanable thing no longer touches the domain.

Solution 5.3.2. With the four combinations of `a` and `b`:

a	b	(imply a b)	(or (not a) b)	(not (and a (not b)))
true	true	true	true	true
true	false	false	false	false
false	true	true	true	true
false	false	true	true	true

All three are false only when `a` holds and `b` does not. The third form of the rule is in `courier-access-nand-domain.pddl`: "it is not the case that the room is restricted and the robot has no keycard".

```
                    (not (and (restricted ?to)
                              (not (has-keycard ?r)))))
```

It solves `courier-access-p01` with the same plan and the same 64 ground actions: the planner pushes the negation inwards and arrives at the `or`. For a reader the `imply` is the clearest of the three.

Section 5.4 — Some object: `exists`

Solution 5.4.1. "Carries a meal" is about some meal, whichever, and the rule forbids it: a negated `exists`. In the file `courier-nomeal-domain.pddl` both demands on a restricted room sit under one `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)))
```

The type of the quantified variable does the selection: a robot that carries a medicine is not affected. `courier-nomeal-p01.pddl` (medicines to the ICU, a meal to ward B) has a plan of 13 steps. `courier-nomeal-p02.pddl` asks for the meal in the ICU and has none, although the robot has a keycard: Kairos searches the 480 reachable states and answers "unsolvable".

Solution 5.4.2. Only the room of the escort changes. The file is `courier-escort-inside-domain.pddl`:

```
              (exists (?e - robot)          ; inside
                (and (has-keycard ?e) (robot-at ?e ?to)))))
```

The old plan no longer works: `r2` waited in ward B, and now it must be in the ICU. The new plan has 9 steps, one more, in which `r2` goes on into the ICU before `r1` arrives. It is no shorter than letting `r2` carry the medicines itself, so with this rule the escort is worth it only when the robot with the keycard is busy or already inside.

Solution 5.4.3. The iron key breaks, the pine chest gives the silver key, which fits only the pine chest, already open. Nothing else can be done: a dead end, and the treasure is lost. The planner must try the oak chest first, or discover the dead end and go back. With `exists` and keys that last, opening a chest never takes anything away, so no order of actions can spoil the puzzle. A variable must be a parameter when an effect has to mention the object (here, to delete `(have ?k)`), or when the plan should name it.

Section 5.5 — Every object: `forall`

Solution 5.5.1. In `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))
```

and `move` gains the precondition `(not (floor-closed))`. That last part is what "after it, no robot moves" asks for, and it matters: without it the planner closes the floor as soon as both robots happen to be in the dock and then goes on delivering, the same cheat as the parked robot of Solution 4.5.1. Now that negative preconditions are available the repair is one line. In `courier-night-p01.pddl` Kairos finds a plan of 15 steps in which `r2`, which starts in the lab, closes the lab door behind it, makes the delivery, closes the pharmacy door and returns to the dock, where `r1` has been all the time; `(close-floor)` is the last step.

Solution 5.5.2. The `forall` is true of an empty type, so the goal holds in the initial state and the plan is empty. The `exists` is false of an empty type and no action creates objects, so there is no 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` is a name that Kairos gives internally to the quantified goal.)

Solution 5.5.3. The goal of `blocks-p02.pddl` is `(forall (?x - block) (on-table ?x))`, and the plan takes the tower apart from the top: `(unstack a b)`, `(put-down a)`, `(unstack b c)`, `(put-down b)`. The goal also forces the hand to end empty, since a held block is not on the table. "`?x` is clear" with a universal quantifier: `(forall (?z - block) (not (on ?z ?x)))`.

Solution 5.5.4. The negation of "every `d` satisfies `c`" is "some `d` does not satisfy `c`", and an implication fails only when its first part holds and its second does not:

```
(exists (?d - door)
  (and (joins ?d ?from ?to) (not (open ?d))))
```

In words: some door on this passage is not open. Note the `and`: the rule of thumb of the chapter (`imply` under `forall`, `and` under `exists`) is this equivalence.

Section 5.6 — What a planner does with rich conditions

Solution 5.6.1. One action for each way of choosing a part from each `or`: $2 \times 2 \times 2 = 8$. For the escort with robots `r1` and `r2`, the `exists` becomes an `or` over the two robots, and the alternatives are four: (1) the room is not restricted; (2) `?r` has a keycard; (3) `r1` has a keycard and is in `?from`; (4) `r2` has a keycard and is in `?from`. Every ground `move` can appear in up to four copies. With a keycard

only for `r2`, the static facts remove alternative (3) everywhere and alternative (2) for `r1`, which is how the count stays at 98.

Section 5.7 — The MODEL method: absence, any, every

Solution 5.7.1. Measured on `blocks-p01` and `blocks-clear-p01`:

Criterion	Without `clear`	With `clear`
Fidelity	Same plan of 6 steps. Needs the inequality in `stack`.	Same plan. A `clear` fact missing in the initial state makes a block unusable.
Size	21 facts, 32 ground actions.	29 facts, 40 ground actions.
Guidance	93 states visited.	93 states visited.
Clarity	Short effects; the state is only the towers. The precondition is harder to read.	Easy preconditions; four effects per action to keep `clear` right.
Portability	Negation, equality and a quantifier.	Plain typed STRIPS.

For a course, the version without `clear` once quantifiers are known: it says what "clear" means. For a temporal planner such as POPF or OPTIC, which have no negative preconditions, the classic one is the only choice.

Solution 5.7.2. *Step O.* Kinds: `cup` and `drink` (the recipes). The machine is unique, so it is not an object; its properties are predicates without arguments.

Property	About	Changes?
`under-nozzle`	a cup	yes
`full`, `holds`	a cup (and a drink)	yes
`with-milk`	a drink	static
`has-water`, `has-beans`, `has-milk`, `clean`	the machine	yes

Step D. The domain is `coffee-domain.pddl`; three of its rules have a rich condition. "Only one cup fits" is about the absence of any cup, and `place` requires `(not (exists (?o - cup) (under-nozzle ?o)))`. "A cappuccino also needs milk" is an implication on a static property of the recipe:

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

"Cleaned only when no cup is under the nozzle" is again a negated `exists`, in `clean-machine`. The plan for `coffee-p01.pddl` has 9 steps: `(add-milk)`; then, for each cup, `place`, `make`, `remove` and `(clean-machine)`. The cup must be removed before cleaning, and the machine cleaned before the second drink: both orders come from the negative conditions. A complementary predicate `(nozzle-free)` would avoid the quantifier, at the price of two more effects to maintain.

C.6. Solutions for Chapter 6

The PDDL files of these solutions are in `examples/ch06/solutions`.

Section 6.1 — Effects that depend on the situation

Solution 6.1.1. Each push action has the parameters of a move (one robot, one cart, 14 ordered pairs of connected rooms) and k item parameters, so it has $14 \cdot 5^k$ candidates: $14 + 70 + 350 + 1750 = 2184$ for `push-cart-0` to `push-cart-3`. A `push-cart-4` would add $14 \cdot 5^4 = 8750$, four times all the others together.
For `courier-cart-cases-p02` (3 items) the same count gives $14\,(1+3+9+27) = 560$ candidates, and Kairos reports 343 ground actions for the *whole* domain, of which 224 are pushes. The grounding discards the candidates whose precondition can never hold. Here they are the ones that repeat an item, because of `(not (= ?i ?j))`: the versions for two and three items keep $3 \cdot 2 = 6$ ordered choices each, and $14\,(1 + 3 + 6 + 6) = 224$. The six orders of the same three items remain: six ground actions that do the same.

Solution 6.1.2. (a) A fixed list is enough: the action names the valve and the pipe. (b) Needs a quantifier: "every pipe" is a universal effect. (c) Needs both: every person, when inside the elevator, changes floor. (d) Needs a condition in the effect: the button can always be pressed, and whether it lights depends on where the elevator is. (If the button could not be pressed with the elevator at the floor, it would be a precondition.) (e) A fixed list is enough, provided the item is a parameter of the action, as in `drop`.

Section 6.2 — Conditional effects: `when`

Solution 6.2.1. All conditions are read in the state before each application.

1. In $\{p\}$ only the first condition holds: q is added. Result $\{p, q\}$.
2. In $\{p, q\}$ the first two hold: q (already there) and r are added. Result $\{p, q, r\}$.
3. In $\{p, q, r\}$ the three hold: q and r are added again and p is deleted. Result $\{q, r\}$.
4. In $\{q, r\}$ the last two hold; p is deleted again, which changes nothing. Result $\{q, r\}$, and so for ever.

Executed in order like a program, the first application would add q, then find q and add r, then find r and delete p: it would reach $\{q, r\}$ in one step instead of three.

Solution 6.2.2. The three levels are encoded with two predicates; off is the absence of both. Each case of the button is one `when`, and the three conditions exclude each other:

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

For the problem `dimmer-p01.pddl` the plan has four steps, two presses for each lamp: low to high to off for `l1`, off to low to high for `l2`. A level cannot be skipped, because the button has no other action. Note the second `when`: it deletes `low` and adds `high` in the

same effect. Had we forgotten the delete, a lamp would be low and high at once, a state that means nothing.

Solution 6.2.3. No plan changes, and the number of ground actions stays at 31 for the three problems with a sample: `(seal r1 sample1)` exists in both versions, since the grounding cannot know in advance that a sample will never be sealed twice. What changes is where the action is applicable. Without the precondition, sealing a sealed sample is allowed and does nothing: every state with a sealed, carried sample gets an action that leads back to itself.
For keeping the precondition: it describes the world (a sealed bag is not sealed again) and it removes useless choices from the search. For removing it: the action becomes simpler, and adding `(sealed ?i)` twice is harmless. The first argument is the stronger one. A precondition that forbids an action without effect costs nothing and saves the planner from considering it.

Section 6.3 — Universal effects: `forall`

Solution 6.3.1. The alarm changes every door, whichever doors the problem has:

```
  ; One effect for all the doors of the problem, however many.
  (:action raise-alarm
    :parameters (?r - robot ?l - location)
    :precondition (and (robot-at ?r ?l) (alarm-button ?l))
    :effect (forall (?d - door)
              (and (not (locked ?d)) (open ?d)))))
```

In `alarm-p01.pddl` the robot is in the lobby, the door to the pharmacy is locked and the only alarm button is in the corridor. The plan has five steps: open the door to the corridor, go there, raise the alarm, come back, and enter the pharmacy through the door that the alarm unlocked and opened. The effect needs no `when`: unlocking a door that was not locked and opening one that was open change nothing.

Solution 6.3.2. The goal is a *condition*: it is true in a state if no lamp is lit there, and it needs `:universal-preconditions`. The effect is a *change*: after the action no lamp is lit, whatever was true before, and it needs `:conditional-effects`. With six lamps the planner expands both when it grounds the task. The goal becomes a conjunction of six negative literals that the search tests in each state; the effect becomes six delete effects of one ground action. A goal never changes the state, and an effect never tests it.

Section 6.4 — Both together: for every object such that

Solution 6.4.1. With the five lamps lit, pressing `b1` (lamps 1 and 2) and `b4` (lamps 3, 4 and 5) puts all of them out; `b2` and `b5` do it too. Kairos first finds a plan of six presses and then the one of two, `(press b4) (press b1)`. The order does not matter: each press changes its lamps whatever their state, so the final state of a lamp depends only on how many times it was changed. For the same reason pressing a button twice never helps: the second press undoes the first. A solution is a *set* of buttons, and a planner that searches sequences explores many orders of the same set. This symmetry is the kind of thing that the Guidance criterion looks at.

Solution 6.4.2. The single `move` has two universal conditional effects, one for the cart that the robot holds and one for the items on that cart:

```
  ; 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))))))
```

The second `forall` quantifies over a cart and an item at once, with both conditions joined by `and`; a `when` inside the `when` of the first one would not be standard PDDL. The plan for `p02` has the same 25 steps, with `move` wherever the chapter had `push-cart`. The domain has 95 ground actions instead of 109 (the 14 `push-cart` are gone) and one requirement less, since the negated `exists` of the old `move` is no longer needed. The cost is in reading plans: `(move r1 pharmacy lobby)` no longer tells whether a cart, and its load, moved too.

Solution 6.4.3. The problem is `courier-cart-twice-p01.pddl`, and Kairos finds a plan of nine steps for it: both robots go to the corridor and *both grab the cart*; then `r2` pushes it into the lab and `r1` pushes it to the ICU. The cart ends in two rooms. Two things are missing in the domain. `grab-cart` does not ask whether another robot already holds the cart, and `push-cart` trusts that a cart being pushed is where its robot is, which is true only if one robot holds it. The repair is one precondition:

```
  (: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)))
```

With it, the same problem (`courier-cart-two-p01.pddl`) is unsolvable, as it must be. The lesson is for step E: a model tested with one robot is tested for one robot. Each new kind of object in the problem deserves its own cheat test.

Section 6.5 — Effects to avoid: the sterile laboratory

Solution 6.5.1. The airlock sprays what enters, in a hand or on a cart. The `move` of the sterile lab is kept, and `push-cart` gains a second universal conditional effect, for the samples on the cart:

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

The two effects of `push-cart` quantify over different types: every *item* on the cart moves, every unsealed *sample* on it is contaminated. For `courier-cart-sterile-p01.pddl` the best plan has nine steps: pick, seal and load each sample in ward A, grab the cart and push it twice. The robot seals each sample while it holds it, before putting it on the cart, because `seal` requires `carrying`. A sample loaded unsealed would have to be picked up again.

Solution 6.5.2. In `courier-sterile-refuse-domain.pddl` the `move` has no conditional effect and one more precondition, a universal one:

```
; 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)))
```

The "no plan" problem is still unsolvable: an unsealed sample cannot get in at all. The "cheat" problem changes: its plan now has eight steps and seals the sample, although the goal does not ask for a usable sample, because sealing is the only way through the door. In this model `contaminated` can never become true and could be removed.

The first model is faithful to an airlock that sprays whoever walks in; the planner must be told, through the goal, that contamination matters. The second is faithful to an airlock with a sensor that keeps the door shut. If the real door has no sensor, the second model is unsafe in a subtle way: it produces only good plans, but it cannot *represent* what happens if something goes wrong, for instance to check a plan written by somebody else.

Section 6.6 — The requirement `:adl`

Solution 6.6.1. The file `lights-adl-domain.pddl` is valid, and `kairos check` reports the same three features as before: types, negative preconditions and conditional effects. The report lists what the file *uses*, and `:adl` announces four more things that it does not use: equality, disjunctive preconditions, and existential and universal preconditions.

A planner decides whether it can handle a domain from the requirements, before it reads the actions. One without disjunctive preconditions can accept the original list, where they do not appear, and is entitled to refuse `:adl`, which includes them. The precise list keeps the door open to more planners.

Section 6.7 — What planners do with conditional effects

Solution 6.7.1. After grounding, `(press b1)` has conditional effects for the two lamps wired to it, `l1` and `l2`. For each lamp exactly one of its two conditions holds (lit or not lit), so the combinations are the $2^2 = 4$ states of the two lamps. One of the four STRIPS actions, for instance, has the precondition `(and (lit l1) (not (lit l2)))` and the effect `(and (not (lit l1)) (lit l2))`. The button `b3` has three lamps and needs $2^3 = 8$ actions. The five buttons together need $4 + 8 + 8 + 8 + 4 = 32$, against five ground actions with conditional effects. On a board where each button changes five lamps, each would need 32.

Solution 6.7.2. The domain of the chapter has 31 ground actions and the one with cases has 32: the move from the corridor into the lab exists twice, as `enter-clean` and as `enter-exposed`, and the other 13 moves are the old ones. Both searches visit the same 55 states. The second domain is the first compilation, one action per combination of conditions, done by hand and with some care: the three conditions of the airlock would give 2^3 combinations, but seven of them have the same effect, and three actions with wider preconditions cover the eight. A modeler who splits cases by hand sees such savings; a compiler that does it blindly may not.

Section 6.8 — The MODEL method: effects that depend on the situation

Solution 6.8.1. State table: where the painter is (`painter-in`, dynamic), the doors (`door`, static), where each piece of furniture is (`in`, dynamic), and three properties that only become true: `covered`, `painted` and `stained`. Action table:

Action	Always changes	Also changes
`walk`	the painter changes room	
`cover`	the piece is covered	
`carry-out`	the painter and the piece change room	
`spray`	the room is painted	*every* piece, *when* it is in the room and not covered, is stained

The spray is always possible, so the furniture is in its effect, not in its precondition:

```
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))))))
```

For `paint-p01.pddl` the plan is `walk`, `cover` the lamp, `cover` the bed, `spray`: four steps. Carrying both pieces out would take longer. The goal must contain `(not (stained bed))` and `(not (stained lamp))`; with `(painted bedroom)` alone the plan is `walk`, `spray`, and the furniture is ruined. That problem is the cheat test of this model.

Solution 6.8.2. In `briefcase-implied-domain.pddl`, `put-in` deletes the `at` fact of the thing, `carry` moves the briefcase only, and `take-out` gains a place parameter and adds `at` for the place where the briefcase is. The domain is plain STRIPS with types. The plan for the

same story has four steps instead of three: put the report in, take the keys out, carry, and take the report out at the office, because only then is `(at report office)` true.

	Conditional effect	No `at` for things inside
Fidelity	a thing inside is at the place of the case	a thing inside is nowhere until it is taken out; goals about places force a `take-out`
Size	10 ground actions, 7 state variables	12 ground actions, 3 state variables
Guidance	3 steps	4 steps; simpler states
Clarity	one rule to explain: contents follow	three plain actions
Portability	`:conditional-effects`, `:negative-preconditions`	any planner

The conditional effect cost two requirements. It bought the right to ask where a thing is while it travels, and a goal that is true as soon as the briefcase arrives. If the only questions ever asked are about things taken out, the STRIPS version is the better choice.

C.7. Solutions for Chapter 7

The PDDL files of these solutions are in `examples/ch07/solutions`.

Section 7.1 — A problem from a story: the waiter

Solution 7.1.1. One possible card (others are as good, if they say what they decide):

Story. A coffee machine serves a queue of people, one drink each, and its containers must be refilled when they run out.

Agents. One: whoever operates the machine (it places cups, starts drinks and refills). The people of the queue are not agents: they only bring orders.

Success. Every person of the queue has the drink asked for.

Scope. *In*: the level of water, beans and milk, in cups; whether a cup is under the spout; which orders are served. *Out*: the temperature and the time a drink takes (the machine handles them); where the refills come from (a cupboard that is never empty); payment; cleaning; people who change their mind.

Assumptions. The order of the queue is fixed. A refill fills the container completely. The machine never fails.

Each "out" has an owner: the machine's electronics, the office supplies, nobody (we accept the simplification). The levels are small bounded quantities of which only the amount matters: a successor chain for each container.

Solution 7.1.2. The problem is `waiter-dessert-p01.pddl`: three dishes ordered by `t1` and two facts for the order, `(before soup1 fish1)` and `(before fish1 cake1)`. The domain does not change, because the order of courses is a static table. The best plan has nine steps and two trips: soup and fish together, then the cake. One trip is impossible with two slots and three dishes. Note that `(before soup1 cake1)` is not needed: the cake waits for the fish, which waits for the soup.

Solution 7.1.3. With the goal `(served soup1)` the plan is load, move, serve: the other three dishes stay on the pass, which the world accepts (the evening is not a success, but the goal did not ask for

one). There is no cheat in the sense of an impossible plan. There is a weak goal: a plan may also load a dish and leave it on the tray for ever. If that matters, the goal must say so, for instance with `(empty s1)` and `(empty s2)`.

Section 7.2 — Patterns for the state

Solution 7.2.1. The problem `river-p03.pddl` has four things and the same two `eats` facts, with the names changed. Kairos finds a plan of nine crossings: the classic seven, and one more round trip for the dog, which can be left with anybody. With one action per animal, a new animal means a new action, and a new enmity means editing the precondition of every action that could leave the two together. With the static table both are lines of the problem.

Solution 7.2.2. Mirrored facts express it by leaving one fact out: the problem keeps `(connected lobby corridor)` and drops `(connected corridor lobby)`. The `or` encoding cannot: it reads every fact in both directions. With the one-way connection `courier-p03` has no plan (Kairos searches 16 states and stops): the robot can deliver the medicines to ward A, and then the only way back to the dock goes from the corridor to the lobby. A one-way connection is a one-way action, and it has just created a dead end out of the whole right side of the map.

Solution 7.2.3. In both encodings the domain does not change. The slots problem gains an object and a fact (`s3`, `(empty s3)`); the chain problem gains an object and a fact (`n3`, `(next n2 n3)`). The files are `waiter-p06.pddl` and `waiter-count-p04.pddl`. The best plan has 11 steps in both, as with two places: with four dishes for two tables, a third place saves no trip.

States searched for `p02`	Slots	Chain
Tray for two	220	123
Tray for three	566	143

The chain grows a little, because there are more counts. The slots grow much faster: k dishes can sit in three slots in many more ways than in two, and each way is a different state for the planner and the same state for the waiter. Symmetry among slots grows with the number of arrangements, which is why slots are a poor encoding of a large capacity.

Solution 7.2.4. (a) Keeps it: the old value is known to be true and is deleted. (b) Does not: if the robot is not at `?from`, the delete does nothing and the robot gains a second position. The precondition is what ties the delete to the real old value. (c) Keeps it, by the rule that adds come after deletes: the token is deleted and added again. The action does nothing, which is a reason to forbid `?a` equal to `?b`, but it breaks nothing. (d) Breaks it: `(free ?r)` stays true, so the action can be applied again and the robot carries two items, then three. The delete of the *twin* is missing.

Section 7.3 — Patterns for control

Solution 7.3.1. The plan cuts the seat and one leg, advances to the joining phase, and can go on with the two joins, the second advance and two paints. It can never cut `leg2`: `cut` needs `(now cutting)`, and no action brings the token back. The goal is out of reach from the third step on, and nothing looked wrong when that step was taken. A phase token turns "too early" into a dead end. The planner

must find out that all the parts have to be cut first, and the delete relaxation does not tell it, since without deletes every phase stays on.

Solution 7.3.2. Without `wait` the problem is unsolvable, and Kairos says so after one state (`patrol-nowait-domain.pddl`). In the first turn the only move of the robot is into `h1`, where the machine is. No action of the robot is applicable, the turn never passes, and the world is frozen. `wait` models the most common decision of an agent in a world that moves: to do nothing now. With turns, doing nothing must be an action, because only actions pass the token.

Solution 7.3.3. The domain is `courier-narrow-domain.pddl`. The availability fact is the absence of `(taken ?l)`; `enter` acquires the room and `leave` releases it:

```
  ; 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))))
```

The plain `move` is restricted to rooms that are not narrow, so the three actions exclude each other. In `courier-narrow-p01.pddl` the two robots enter the pharmacy one after the other, six steps in all; the cheat test `p02`, with both inside, is unsolvable. This is one action per case (Chapter 6): a single `move` with conditional effects on `taken` is the alternative.

Section 7.4 — Patterns for the actions

Solution 7.4.1. `move` and `push-cart`: each is undone by the same action with `?from` and `?to` exchanged, as long as the map has mirrored facts. `grab-cart` and `release-cart` are an inverse pair. `pick` and `drop` are an inverse pair for an item on the floor. For an item on the cart the inverse of `pick` is `load`, which puts it back on the cart, and the inverse of `load` is `pick`. Every action has an inverse, so with one robot the domain has no dead ends: from any reachable state the robot can undo its way back and try something else. (The two-robot flaw of Exercise 6.4.3 is another matter: there the state stops making sense.)

Solution 7.4.2. The domain `blocks-coarse-domain.pddl` has three actions: `move-to-block`, `move-to-table` and `move-from-table`. Two would not do, because the table is not a block: it is never `clear` or not, and any number of blocks fit on it. The reversal of the tower takes three steps:

```
$ 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)
```

The four-action model needs six steps for the same thing, twice as many, since every move is a pick and a put. What it can express and this one cannot is everything about the hand: a goal or an initial state in which a block is held, two arms that share the blocks, a hand that fails between picking and placing. If none of that is in the story, the merged model is better: half the plan length and no `holding` facts.

Section 7.5 — Anti-patterns

Solution 7.5.1. The three anti-patterns:

- **Forgotten delete.** `pick` does not delete `(ball-at ?b ?r)`: a ball is in the gripper and still in the room. The plan for `p01` is not shorter (the balls must still travel), so the usual symptom is missing. The `[sas]` line gives it away: `13 variables (3 from 3 mutex groups, 10 binary)`, where the repaired domain has `7 variables (7 from 7 mutex groups)`. The planner could not prove that a ball is in one place. A cheat test shows it too: `gripper-buggy-p02.pddl` wants one ball in both rooms, and gets it in three steps.
- **Out-of-step twins.** `busy` is the twin of `free`; `pick` sets it and `drop` never clears it. Nothing reads it, so nothing fails yet. The repair is to remove it: a twin that no precondition uses is only a risk.
- **Idle parameter.** `move` has a gripper parameter that it does not use. The plans show it, `(move rooma roomb left)`, and so does the count: 40 ground actions against 36.

The repaired domain is `gripper-domain.pddl`. With it the cheat test is unsolvable. This exercise has a moral for step E: the typical problem passed with all three bugs in place.

Solution 7.5.2. It is a goal in the initial state. The sample starts in ward A, so it is not in the corridor, and it starts clean, so it is not contaminated: both negative literals are true before any action, and the empty plan is correct. Negative goals are prone to this because of the closed world assumption: whatever `:init` does not list is false, so a goal made only of negations is true at the start unless the initial state says otherwise. A goal should contain at least one thing that has to be *achieved*; here, `(item-at sample1 lab)`.

Section 7.6 — Debugging a model

Solution 7.6.1. Step 1 is enough. The note says that `(box-at c12)` can never be reached: `push` adds it, but is never applicable. To put the box in `c12` from `c22` the push must go north, with the player in `c32`; the instance needs `(adjacent c32 c22 north)` and `(adjacent c22 c12 north)`. A search for `north` in the file finds the direction declared as an object and used in no fact: the eight facts for going up are missing. Whoever typed the grid wrote the vertical neighbors in one direction only, which is the mistake of the half-mirrored facts. With the complete grid (`sokoban-p04.pddl`) the plan has four steps: three walks to get below the box and one push north.

Solution 7.6.2. Work backwards from the goal. `(served soup1)` is added only by `serve`, so relax `serve` first: remove its preconditions one at a time, starting with the ones that are most complex or were written last (here the `forall`, then `ordered`, then `on-tray`, then the position). If a removal brings a plan, that precondition names

the fact to chase, and the same question is asked one level down: which action should add it, and which of *its* preconditions fails? If `on-tray` is the obstacle, `load` is the next action to relax. In general: the last action of the plan you expected, then the one before it. Each round moves one step back along the plan that should have existed, until an action is found whose missing effect or impossible precondition explains everything after it.

Section 7.7 — The MODEL method: the classical method, consolidated

Solution 7.7.1. **Model card.** Story: a ferry takes cars between two shores, two at a time at most. Agents: the ferry. Success: every car on the shore it wants. Scope, in: where the ferry and each car are, how many cars are aboard; out: time, the order of cars in the queue, fares. Assumptions: a car aboard stays aboard until it is taken off; cars do not move by themselves.

State table. `(ferry-at ?s)` and `(car-at ?c ?s)`, dynamic, each with the at-most-one invariant; `(aboard ?c)`, dynamic (a car aboard has no shore: its position is that of the ferry, the "do not store what can be deduced" of Chapter 6); `(load ?n)`, dynamic, and `(next ?n ?m)`, static: a successor chain.

Action table. `sail` (the ferry changes shore), `board` (a car on the shore of the ferry goes aboard, the count goes up) and `debark` (the inverse). The three form inverse pairs, so there are no dead ends. The domain is `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)))

  (:action debark
    :parameters (?c - car ?s - shore ?n ?m - count)
    :precondition (and (ferry-at ?s) (aboard ?c)
                       (load ?m) (next ?n ?m))
    :effect (and (not (aboard ?c)) (car-at ?c ?s)
                 (not (load ?m)) (load ?n))))
```

Test set. Trivial: one car. Typical (`ferry-p01.pddl`): three cars to the north and one to the south, solved in 11 steps with three crossings; the ferry brings the southbound car on its way back. No plan and cheat in one (`ferry-p02.pddl`): three cars aboard at once, unsolvable.

Comparison. The rival of the chain is two slots, `(deck-free ?k)` and `(on-deck ?c ?k)`. Fidelity: the same, unless the positions on the deck matter (first on, last off). Size and Guidance: similar ground actions; the slots add the symmetry of which car is on which side of the deck. Clarity: slots read better in plans. Portability: both are STRIPS with types and equality. For the story as told, the chain; with a ferry that unloads in order, the slots, or a chain of positions.

Solution 7.7.2. There is no single answer. A good one names, for a domain of yours: the static tables and whether any rule is still hard-coded in an action; each single-valued property and the action that keeps its invariant; each capacity and whether it is a chain or slots; each action and its inverse, or the reason it has none. The measurement should report ground actions, state variables and

states searched before and after the change, on the same problem, and say which criterion of step L moved.

C.8. Solutions for Chapter 8

The PDDL files of these solutions are in `examples/ch08/solutions`.

Section 8.1 — What a fact cannot say

Solution 8.1.1. (a) Facts: six places, and the elevator moves between neighbors. A predicate `(lift-at ?f)` and a static `(above ?f1 ?f2)` are enough. (b) A quantity: prices of any amount are subtracted from it. (c) A name: nobody adds bed numbers. (d) It depends on how the oven is heated. If an action "preheat" takes it to the wanted temperature, a fact `(hot-enough ?o)` is all that recipes ask for: only a threshold matters. If heating adds degrees step by step and different recipes ask for different temperatures, it is a quantity. (e) A fact. (f) A quantity: jobs of different lengths subtract different amounts.

Section 8.2 — Functions: numbers in the state

Solution 8.2.1. Five functions: `(altitude ?d - drone)`, in meters, dynamic; `(payload-limit ?d - drone)`, in grams, static; `(payload ?d - drone)`, the grams carried now, dynamic; `(distance ?from ?to - site)`, static; and `(weight ?p - parcel)`, static. The story does not mention `payload`, but the limit is useless without the sum that is compared with it: it is the summary fluent of Section 8.4.

Solution 8.2.2. (1) `(battery r1 80)` writes a value as an argument: a syntax error, the checker rejects the file. It should be `(= (battery r1) 80)`. (2) The distance is given in one direction only. The file is valid; the robot can go from the lobby to the dock and never come back, because `(distance dock lobby)` is undefined. (3) `(= (consumption) 1)` forgets the robot: `consumption` has one parameter, so the checker reports a wrong number of arguments. Had the function been declared without parameters, the line would be right.

Section 8.3 — Comparing and changing numbers: the battery

Solution 8.3.1. The first delivery and the way back to the dock need $64 \times 1.5 = 96$ units and the second delivery 70.5, as before. With 50 units the robot cannot even do the first one (57 units to reach ward A), so it must charge before leaving, and again between the two deliveries: two charges, 20 actions. The problem is `courier-battery-half.pddl`, and the best plan of Kairos starts with `(charge r1 dock)` and then repeats the plan of the chapter. The first plans that it prints are longer (22 actions in our run, with moves that go and come back): the heuristic does not see the battery running out, as Section 8.7 explains.

Solution 8.3.2. The domain `courier-reserve-domain.pddl` adds a static function `(reserve ?r)` and changes the precondition of `move`:

```
    (>= (- (battery ?r)
           (* (distance ?from ?to) (consumption ?r)))
        (reserve ?r)))
```

The effect does not change. The plan of the chapter reached the dock with 4 units, below the reserve of 10, so its move from the lobby to the dock is no longer applicable. The round trip to ward A alone now needs 96 units out of the 90 that can be used. The planner finds a plan of 21 actions that splits the work in another way: it fetches the sample first and leaves it in the corridor, fetches the medicines, charges with 16 units left, and delivers both in the second round. A reserve of 10 costs two actions and a different order of work.

Solution 8.3.3. The domain does not change: the wish is a numeric condition in the goal (`courier-battery-end.pddl`):

```
(:goal (and (item-at meds1 ward-a) (item-at sample1 icu)
            (>= (battery r1) 50))))
```

The plan of the chapter ends with 29.5 units. The robot cannot charge after the last delivery and stay at the ICU, so the planner charges later instead: it picks the sample *before* the second visit to the dock, charges with the sample on board and goes straight to the ICU, where it arrives with 53.5 units. The plan has 24 actions and two charges.

Section 8.4 — Numbers that never change

Solution 8.4.1. In `courier-battery-load-domain.pddl` the consumption of a move is

```
               (* (* (distance ?from ?to) (consumption ?r))
                  (+ 1 (* 0.02 (load ?r)))))))
```

in the precondition and in the effect. The expression now reads a dynamic fluent, the load, so the planner cannot reduce it to a constant when it grounds the action. In the test the robot uses 2 units per meter. It goes from the dock to the lobby empty ($4\times2 = 8$, 92 left), picks both items (7 kg, a factor of 1.14) and covers $10 + 12 + 5 = 27$ meters: $27 \times 2 \times 1.14 = 61.56$. It arrives with 30.44 units, which is what the trace of Kairos shows.

Solution 8.4.2. Sixteen of the seventeen values belong to static functions (`distance`, `battery-capacity`, `consumption`): the planner replaces them by their values while grounding. Only `(battery r1)` is changed by some action. The load problem also reports one, `(load r1)`: the capacity and the three weights are static.

Solution 8.4.3. In `truck-p04.pddl` the truck is in Cuenca with 40 liters, Valencia is 200 km away (50 liters) and the station is in Teruel, 150 km from Cuenca and 140 from Valencia. The truck drives to Teruel (37.5 liters, 2.5 left), refuels four times (42.5; three times would give 32.5, short of the 35 that the last leg needs) and drives to Valencia: six actions.

Section 8.5 — Setting, scaling and asking

Solution 8.5.1. The domain `jugs-closed-domain.pddl` is the one of Listing 8.6 without `fill` and `empty`; the problem is `jugs-p03.pddl`. The shortest plan has seven pours, and Kairos proves it:

```
(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)
```

The states are (8,0,0), (3,5,0), (3,2,3), (6,2,0), (6,0,2), (1,5,2), (1,4,3), (4,4,0).

Solution 8.5.2. `jugs-when-domain.pddl` has one `pour` whose two conditional effects carry the conditions that told the two actions apart:

```
(when (<= (+ (amount ?from) (amount ?to)) (volume ?to))
  (and (increase (amount ?to) (amount ?from))
       (assign (amount ?from) 0)))
```

and the same for the other case. The plan has the same six steps. Fidelity is equal. Size: 6 ground actions instead of 8, each a little heavier. Guidance: about equal here. Clarity is a matter of taste: one action matches the single gesture of pouring, two actions make the two outcomes visible in the plan (`pour-all`, `pour-until-full`), which helps to read it. Portability favors the two actions: numeric conditions inside `when` are accepted by fewer planners than plain numeric preconditions.

Solution 8.5.3. The volume could be halved for ever: 2, 1, 0.5, 0.25... Every halving is a new state, so the reachable states never end, and for a problem without a plan the planner could not prove it (Section 8.7). The precondition is also true to the lab: a drop cannot be split with a pipette.

Solution 8.5.4. In `elevator-domain.pddl`, where the elevator and each person are is a matter of facts (`lift-at`, `person-at`, `inside`); how many people are inside is a number, `(passengers)`, compared with `(max-passengers)` in `board` and updated by `board` and `leave`. The count is a summary of the `inside` facts. The plan for three people and an elevator for two has 14 actions: the elevator comes down, takes two people up, comes down again and takes the third.

Section 8.6 — Values that were never given

Solution 8.6.1. `kairos check` says `valid, plannable`: nothing is wrong with the syntax. The planner answers `unsolvable`, with the note quoted in Section 8.9: `pick` is never applicable to the meal, because its precondition reads `(weight meal1)`. With a default of zero the meal would weigh nothing, the three items would travel together in one trip, and the plan would look better than the correct one. Nobody would suspect it.

Solution 8.6.2. Fidelity: the same plans, as long as every connection has its distance in one direction. Size: twice the candidate moves, of which grounding keeps the same number (a move whose distance is undefined is never applicable). Guidance: no change. Clarity: worse in the domain (two actions that differ in the order of two arguments, and plans with two names for moving) and better in the problem (half of the lines). Portability: no change. A fair verdict: it trades a common mistake in problems for a domain that is harder to explain, and it only moves the trap, since forgetting a connection in both directions is as silent as before. Generating the problem file with a script is a better cure.

Section 8.7 — Why numbers make planning harder

Solution 8.7.1. (a) Finite: the battery only takes values between 0 and the capacity that are the initial charge minus a sum of consumptions. (b) Finite: `packed` is limited by the space of the box, and the stock grows only from zero. (c) Infinite: the stock can be increased for ever. (d) Infinite, and it is easy to miss: the volume is bounded, but diluting and discarding half in turns brings the volume back to where it was and halves the concentration each time, without end. A lower limit for the concentration in `dilute` would close it. (e) Infinite twice over.

Solution 8.7.2. In our runs the planner searched about 307 000, 604 000 and 1 153 000 states: twice the time, twice the states, and the same answer, `timeout`, with exit code 11. None of the answers is "unsolvable". A timeout says that no plan was found in the states visited. It may be that the plan is further away, that the heuristic is misled, or that there is no plan and the state space is infinite; the planner cannot tell which. To learn more, bound the quantities, so that the planner can exhaust the states, or relax the goal as in the debugging procedure of Chapter 7.

Section 8.8 — The MODEL method: quantities

Solution 8.8.1. `courier-slots-domain.pddl` gives the robot slot objects; `pick` takes an empty slot of the robot and `drop` frees it. With two slots the three items need two trips and the plan has 12 actions, as with weights.

	Weights (numbers)	**Slots**
Fidelity	Limit in kilograms: two light items or one heavy	Limit in pieces: any two items, whatever they weigh
Size	62 ground actions	110: every pick and drop once per slot
Guidance	One numeric fluent; relaxation ignores the limit	Classical; but the two slots are interchangeable, a symmetry
Clarity	Close to the story	An extra type, three predicates and one more parameter
Portability	Needs `:numeric-fluents`	Plain STRIPS with types

Verdict: slots when the limit of the real robot is a number of places (a tray with two compartments) or the planner is classical; weights when the limit is a weight or a volume. They are not two encodings of the same thing unless all the items weigh the same.

Solution 8.8.2. Give `move` one more static fact and one more action. A fact `(long ?from ?to)` marks the two connections; `move` requires its absence and steps one level, and a second action `move-long` requires it and steps two levels, with three level parameters and two `next` preconditions, `(next ?mid ?before)` and `(next ?after ?mid)`. The size grows little, because the static facts fix the extra parameter. But each new amount needs its own action, and an amount of k levels needs k `next` preconditions: at that point the number is cheaper.

Solution 8.8.3. *Model card.* Story: a robot waters beds from a tank that it refills at a tap. Success: every bed has received what it needs. In: the water in the tank, the need of each bed. Out: distances and the battery, the time to water. Assumption: the tap never runs dry.
State table. Where the robot is: a fact, dynamic. Water in the tank: a quantity, dynamic, between 0 and the size of the tank, kept by the precondition of `water` and the `assign` of `refill`. Need of a bed:

a quantity, dynamic, from its initial value down to 0, kept by the precondition of `water`. Size of the tank and dose: static.
Action table. `go` (from, to); `refill` (at a tap, tank not full: tank becomes full); `water` (at a bed, a dose in the tank, the bed needs at least a dose: both decrease by a dose).
The files are `greenhouse-domain.pddl` and `greenhouse-p01.pddl`. The typical test gives 10 actions: refill, three doses to the tomatoes, back to the tap, refill, two doses to the lettuces. For the other tests: a problem with a single bed that needs 2 liters and a full tank (trivial); a bed that needs 5 liters, which doses of 2 never make zero (no plan, and proved, because both quantities are bounded); and a goal `(> (tank) 6)` for the cheat. Note that the goal uses equalities, `(= (need tomatoes) 0)`: safe here because all the amounts are whole numbers.

C.9. Solutions for Chapter 9

The PDDL files of these solutions are in `examples/ch09/solutions`.

Section 9.1 — When any plan is not enough

Solution 9.1.1. (a) The number of moves; plan length is exactly that. (b) Money: the fuel spent, in euros, against a fixed income per delivery. Plan length is not fair: a long move and a short one count the same. (c) Time, and since every action takes about the same, plan length is a fair stand-in. (d) The food thrown away, in grams or in euros. Plan length says nothing about it: two plans of the same length can use different ingredients.

Solution 9.1.2. Without a metric the number is the plan length, 4. With the metric it is the value of `(total-cost)` at the end of the plan, in meters: 20, then 16. In the first run every plan has four actions, so the first plan cannot be improved and nothing else is printed. In the second, the first plan found is the same one, a better one exists, and the anytime search finds it.

Section 9.2 — Action costs

Solution 9.2.1. In `courier-cost-handling-domain.pddl` the actions `pick` and `drop` gain an increase of 3 of the total cost. The best plan of `courier-cost-handling-p02.pddl` costs 183: the same 159 meters, and eight handlings, the least possible for four items, at 3 each. The plan of the chapter had ten handlings, because it left the sample in the corridor and picked it again: that would now cost 159 + 30 = 189. The planner finds another order of the four deliveries, again with `r1` alone, that covers the same meters without the stop. When handling was free the two plans were equally good and the planner had no reason to prefer the tidy one.

Solution 9.2.2. The cost of an action must be a non-negative number or a *static* function. `(load ?r)` is changed by actions, so the cost of a move would depend on the state, and the domain would have a second fluent that changes. The model is legal PDDL with `:numeric-fluents` instead: keep the load as in Chapter 8, add a counter such as `(effort)`, increase it in `move` by an expression that reads the load, and minimize the counter. A planner limited to action costs will not read it.

Solution 9.2.3. The plan has 18 moves: 4 + 6 + 6 + 10 + 12 to deliver the medicines (38), 12 + 10 back to the lobby with the sample left in the corridor (60), 10 + 12 to take the meal to ward B (82), 12 + 8 to bring the sample to the lab (102), 8 + 10 + 6 to the pharmacy (126) and 6 + 10 + 12 + 5 to the ICU: 159. The validator agrees: `plan valid, makespan 28, metric 159`.

Section 9.3 — Costs from data: a kitchen with prices

Solution 9.3.1. In `kitchen-p02.pddl` the price of the butter is 0. One of the two dishes takes the butter and the other a bottle of oil: fats cost 0 + 2 instead of 2 + 2, and the meal costs 8 instead of 10. Kairos puts the butter in the cake. Which dish gets it does not matter for the cost, so the omelette would have been as good.

Solution 9.3.2. With a metric, never in an optimal plan: the second egg costs its price and achieves nothing new, since `(with-egg cake1)` was already true. Without a metric the shortest plans do not do it either, because it is one more action, but nothing forbids a satisficing planner from returning such a plan. Whether the domain should forbid it is a question of fidelity: if the cook would never do it, a precondition `(not (with-egg ?d))` removes plans that are not wanted and makes the search space smaller, at the price of `:negative-preconditions`. A metric makes waste unattractive; it does not make it impossible.

Section 9.4 — The planner at work: anytime search

Solution 9.4.1. On our machine:

Option	Cost	Stop reason	Total time
`--no-anytime`	188	`no improvement`	2 ms
`--target-cost 165`	164	`target reached`	0.4 s
none	159	`proven optimal`	4.1 s

A robot that must start within a tenth of a second can use `--timeout 0.1`, which returned a plan of cost 168 in our runs: three plans are found in the first 14 milliseconds, and the anytime search gives the best of them when the time is up. `--no-anytime` wastes almost all of the tenth of a second.

Solution 9.4.2. (1) Change what is measured: time instead of meters. With durative actions and the makespan as the metric (Chapter 10), two robots that work at once finish earlier. This assumes that the hospital wants the round done soon. (2) Keep the meters and add a limit: a maximum of meters per robot and per round, with a counter for each robot and a precondition in `move`, as the budget of Section 9.7. This assumes that what matters is the wear or the battery of each robot. A third, weaker, option is a metric that punishes imbalance; PDDL has no maximum operator, so it is awkward to write, and the limit says the same more simply.

Solution 9.4.3. It runs to the end and answers `solved (proven optimal), cost 159`. The target only stops the search early when a plan meets it. No plan of 100 meters exists, so the search ends for its own reasons. Answering "unsolvable" would be wrong: the problem has plans, and a target is a wish about when to stop, not a part of the problem. (To *require* a cost of at most 100, write a limit.)

Section 9.5 — Metrics over any fluent

Solution 9.5.1. `courier-handlings-domain.pddl` adds `(handlings)`, increased by 1 in `pick` and in `drop`, and `courier-handlings-p01.pddl` has the metric

```
(:metric minimize (+ (energy-used) (* 5 (handlings))))
```

The best plan costs 112: `r1` does the two deliveries in turn, 92 units of energy and four handlings. The plan with the stop in the corridor uses 88 units and six handlings: 88 + 30 = 118. The stop saved 4 units of energy and costs two more handlings, 10 units at this weight. With a weight below 2 per handling the stop would come back.

Solution 9.5.2. It rewards plans that end with the robot as loaded as possible: the robot picks whatever fits before the end and drops nothing that the goal does not force it to. There is a best plan, because the load is bounded by the capacity, but it is not what anybody wants. The author probably meant "carry as much as possible in each trip", that is, few trips. That is a sum that accumulates: a cost on each move, or a counter of moves, minimized.

Section 9.6 — Several wishes in one number

Solution 9.6.1. "Both robots" is the point (68, 128) and "`r1` alone", with the stop, is (88, 88). Their costs are $68 + 128w$ and $88 + 88w$, equal when $40w = 20$: $w = 0.5$. Below it the plan with both robots wins, above it `r1` alone. The files `courier-quality-w04.pddl` and `courier-quality-w06.pddl` confirm it: with $w = 0.4$ the optimum is 119.2 (68 + 51.2) and with $w = 0.6$ it is 140.8 (88 + 52.8). At exactly 0.5 the two plans tie and either is optimal.

Solution 9.6.2. The small car burns 9.5 liters with the tunnel alone and 8 with both highways, and pays 6 or 10 euros of tolls. With fuel at p euros the costs are $6 + 9.5p$ and $10 + 8p$, equal at $p = 8/3$, about 2.67 euros. At 3 euros (`road-trip-p04.pddl`) the best plan is both highways, at a cost of 34 against 34.5.

Solution 9.6.3. No difference of energy between two plans of this problem can outweigh a single meter, so the energy never changes which amount of meters is chosen. It only decides among plans with the same meters: a tie-breaker. This is how to say "first of all few meters, and among those, little energy", an order of priorities and not a trade-off. The large weight must be larger than any difference that the second measure can make.

Section 9.7 — A limit or a metric

Solution 9.7.1. `road-trip-tank-domain.pddl` adds a static function `(tank ?c)` and, in both driving actions, the precondition that the fuel used after the stretch does not exceed it:

```
(<= (+ (fuel-used)
       (/ (road-km ?from ?to) (km-per-liter ?c)))
    (tank ?c)))
```

With 9 liters (`road-trip-tank-p01.pddl`) only one route is short enough: both highways through Burgo, 160 km and 8 liters. The plan pays 10 euros of tolls, the dearest of all, and it is optimal because it is the only plan. With 7 liters (`road-trip-tank-p02.pddl`) there is

no plan, and the planner proves it after 4 states. The metric wanted no tolls; the limit decided.

Solution 9.7.2. (a) A limit: an empty battery is a failure, not a worse result. It is the precondition of `move` in Chapter 8. (b) A limit if 5 kg is what the tray supports; if it is a recommendation, a metric that penalizes overweight trips. (c) A metric: less time is better and no particular time is a failure. If the sample spoils after an hour, there is also a limit, and both are needed. (d) A metric, as the words "when it can" say: a cost on entering the ICU. A limit would make deliveries to the ICU impossible.

Section 9.8 — The MODEL method: what makes a plan better

Solution 9.8.1. Without the metric (`kitchen-p01-nometric.pddl`) Kairos returns the plan with the farm egg, the butter and the honey, says that its cost is 7 and that it is proven optimal: by length, it is. The two encodings differ in Fidelity (length cannot tell an 18-euro meal from a 10-euro one) and a little in Clarity (the prices and the `increase` lines). Size is the same, since the prices are static; Guidance is about the same; and Portability hardly differs, because `:action-costs` is accepted by classical planners. When a domain has prices, there is no reason to leave them out.

Solution 9.8.2. *Step M.* What makes a plan better: less energy, in Wh, and less risk, on the operator's scale from 0 to 5 per flight. It is not one number until the operator says how many Wh a point of risk is worth; that weight goes in the model card with the operator's reason. "Not at any risk" may also hide a limit (never a flight of risk 5), to be asked.

The domain `drone-domain.pddl` has two static functions per route, `flight-energy` and `flight-risk`, and two counters, `(energy)` and `(risk)`, that `fly` increases. Over the square the trip costs 22 Wh and 8 points of risk; by the river, 55 Wh and 1 point. With a weight w for the risk the costs are $22 + 8w$ and $55 + w$, equal at $w = 33/7$, about 4.7. In `drone-p01.pddl` the weight is 10 and the drone goes around (cost 65 against 102); in `drone-p02.pddl` it is 2 and the drone crosses the square (38 against 57).

C.10. Solutions for Chapter 10

The files of these solutions are in `examples/ch10/solutions`.

Section 10.1 — What a sequence cannot say

Solution 10.1.1. Robot `a` works $5 + 3 + 8 + 2 = 18$ seconds and robot `b` $4 + 4 + 6 + 1 + 7 = 22$. One action after another, the plan takes $18 + 22 = 40$ seconds. With each robot on its own it takes as long as the busier robot: 22 seconds. The timeline has two rows; the row of `a` ends at 18 and is empty from there to 22.

Solution 10.1.2. (a) A cost is enough: one agent, and what matters is a sum (kilometers). (b) Time: two agents work at once and success depends on the finishing time, which a sum of durations does not give. (c) Time, and more than this chapter offers: the dishes must be ready within a short interval of each other, which is coordination

between actions (Chapter 11). (d) A cost: battery used is a sum over the actions, whatever their order or overlap.

Section 10.2 — A first durative action

Solution 10.2.1. The tempting prediction is three seconds more, 55.006. The planner returns 52.006, the same as before:

```
20.002: (pick r1 meds1 pharmacy) [5.000]
20.003: (move r1 pharmacy lobby) [10.000]
```

In this domain the pick only needs the robot at its start, so the robot leaves a millisecond later and the pick "runs" while it walks. It ends at 25.002, long before the drop at 50.006 needs the item. The pick could last 29 seconds without changing the makespan. A duration that can be changed without any effect on the plans is a sign that the action is not tied to the agent that performs it; the invariant of Section 10.4 ties it.

Solution 10.2.2. Each appliance is a resource of its own, with its own token (`toaster-ready`, `kettle-ready`), taken at start and returned at end.

```
;; 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)))))
```

Listing C.2: Breakfast (`breakfast-domain.pddl`).

The two actions share nothing, so they start together and the makespan is the longer duration:

```
$ kairos breakfast-domain.pddl breakfast-p01.pddl
; cost = 180
; makespan = 180.000
0.000: (toast) [120.000]
0.000: (boil) [180.000]
```

Solution 10.2.3. The two effects have no time specifier. The conditions are right. The file is `wrong-untimed-effect-domain.pddl`, and the checker reports each effect:

```
$ 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
```

The repair is not mechanical: the delete goes `at start` and the add `at end` (Table 10.5 shows what the other choices do).

Section 10.3 — What happens at each instant

Solution 10.3.1. Six happenings:

1. 3.001: `travel f1 f2` starts (deletes `(lift-at e1 f1)`).
2. 6.001: `travel f1 f2` ends (adds `(lift-at e1 f2)`).
3. 6.002: both `board` actions start. They are one happening: neither touches what the other needs (each deletes the position of its own passenger).
4. 8.002: `board ada` ends (adds `(boarded ada e1)`).
5. 10.002: `board bob` ends (adds `(boarded bob e1)`).
6. 10.003: `travel f2 f3` starts (deletes `(lift-at e1 f2)`).

Happenings 2 and 3 interfere through `(lift-at e1 f2)`: the end of the travel adds it and the boardings need it at start, hence 6.002 and not 6.001. Happenings 5 and 6 interfere through the same fact: the end of Bob's boarding needs it (`at end`) and the start of the travel deletes it, hence 10.003. Happenings 3, 4 and 5 do not interfere with each other: they only read the position of the lift.

Solution 10.3.2. The makespan is 42.500: five separations, not six.

```
$ kairos courier-time-domain.pddl courier-time-p01.pddl --
    epsilon 0.1
; cost = 42.50000000000001
; makespan = 42.500
...
```

Seven actions have six junctions, but at one of them the two happenings do not interfere: the end of the pick and the start of the move that follows it share an instant (Section 10.4). At the other five junctions the end of a move adds the `robot-at` fact that the next action needs at its start.

Section 10.4 — Conditions in time

Solution 10.4.1. The robot cannot pick from another room: an instant after the start, the invariant needs it in the room. The plan of `courier-time-overall-domain.pddl` is the honest one, one millisecond shorter, 42.004:

```
4.001: (move r1 lobby pharmacy) [6.000]
10.001: (pick r1 meds1 pharmacy) [2.000]
```

The pick starts at 10.001, the very instant at which the robot arrives, where before it started at 10.002. Without a condition at start, the start of the pick does not interfere with the end of the move, and the interval of `over all` begins just after it. Nothing impossible

happens, but the pick now relies on a fact at the exact instant at which it becomes true, which is what the separation by ϵ exists to avoid. Keep both lines.

Solution 10.4.2. The domain is `elevators-nowait-domain.pddl`. With the lift asked for only at the start, it leaves while people are still stepping in. The makespan drops from 28.008 to 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 starts boarding on the second floor at 6.002. A millisecond later the lift departs. When his boarding ends, at 10.002, the lift has been to the third floor and is on its way down, and Bob is "in" it. The same happens to Eve, and at the ground floor the lift is not asked to wait for anybody to leave either.

Section 10.5 — Durations that depend on the action

Solution 10.5.1. The five moves cover $4 + 6 + 6 + 10 + 12 = 38$ meters: 19 seconds at 2 meters per second. The pick and the drop still take 2 seconds each, and five junctions need ϵ: $19 + 4 + 0.005 = 23.005$. The planner agrees (`courier-time-fast-p01.pddl`):

```
; makespan = 23.005
0.000: (move r1 dock lobby) [2.000]
2.001: (move r1 lobby pharmacy) [3.000]
```

Solution 10.5.2. One action for each case, told apart by a condition on the gripper. The condition is also an invariant: the robot must not change from free to loaded in the middle of a move.

```
(: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 (free ?r))
                  (over all (free ?r)))
  :effect (and (at start (not (robot-at ?r ?from)))
               (at end (robot-at ?r ?to))))

(:durative-action move-loaded
  :parameters (?r - robot ?i - item ?from ?to - location)
  :duration (= ?duration (/ (distance ?from ?to)
                            (loaded-speed ?r)))
  :condition (and (at start (robot-at ?r ?from))
                  (at start (connected ?from ?to))
                  (at start (carrying ?r ?i))
                  (over all (carrying ?r ?i)))
  :effect (and (at start (not (robot-at ?r ?from)))
               (at end (robot-at ?r ?to))))
```

Listing C.3: Two moves with different speeds (from `courier-loaded-domain.pddl`).

`move-loaded` needs the item as a parameter to name what is carried; with one gripper it adds no ground actions that matter. The delivery takes 70.006 seconds: the three loaded moves last 12, 20 and 24.

```
10.002: (pick r1 meds1 pharmacy) [2.000]
12.003: (move-loaded r1 meds1 pharmacy lobby) [12.000]
```

There are six separations now. The loaded move needs `(carrying r1 meds1)`, which the end of the pick adds, so these two happenings interfere and the robot no longer leaves at the instant the pick ends.

Section 10.6 — Working in parallel

Solution 10.6.1. No relay (`courier-time-p06.pddl`): the best plan found gives one delivery to each robot and has makespan 47.006. The relay paid because `r2` was fast enough to fetch the medicines and hand them over before `r1` could have brought them; at the same speed the detour through the pharmacy costs `r2` as much as it costs `r1`. The makespan is set by `r1`, whose drop in the ICU ends at 47.006; `r2` finishes at 36.003.

Solution 10.6.2. `r2` is farther from the pharmacy (16 meters against 10) but twice as fast, and it makes the whole delivery in 26.005 seconds, against the 42.005 of `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` does not appear in the plan. An agent with nothing useful to do has no lines; there is no action for waiting and none is needed.

Section 10.7 — One thing at a time

Solution 10.7.1. The honest plan drops (2 seconds) and then disinfects (20): 22 seconds, 22.001 with the separation. The faulty domain plans this:

```
$ 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]
```

The drop does not ask for the token, so it runs under the disinfection. The correct domain gives 22.001. The test found the bug because its honest makespan was known beforehand; a problem with a goal that only needs one action at a time would have passed.

Solution 10.7.2. The token `(counter-free ?l)` belongs to the room. `pick` takes it at start and returns it at end, next to the token of the 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))))
```

Listing C.4: A pick that takes the counter (from `courier-counter-domain.pddl`).

The problem lists `(counter-free pharmacy)` in its initial state, and so for every room with a counter: a room without the fact would allow no pick at all. With the token the second pick waits for the first (makespan 12.003); without it both robots pick at 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]
```

Section 10.8 — Other domains: a kitchen and an elevator

Solution 10.8.1. With one burner, the row of the burner has no gaps and the row of the cook has many: the cook is not the bottleneck, and a second cook changes nothing. The planner confirms it: `kitchen-p03.pddl` (two cooks) has makespan 40.004, as before. With two burners no row is full, and the makespan, 28.003, is the chain of the soup behind the pasta: 1 + 6 + 20 + 1. A third burner (`kitchen-p04.pddl`) lets the cook prepare the soup first and still cook the rest in parallel: 27.002, which is the soup alone, 6 + 20 + 1. No number of burners will go below that with one cook; the next gain needs somebody to prepare the pasta and the sauce while the soup is being prepared.

Solution 10.8.2. `e1` needs 6 seconds to climb to the second floor, 4 for Bob to board (Ada boards meanwhile), 6 to come down and 4 for Bob to leave: 20 seconds, plus the separations. `e2` with Eve alone would need 2 + 9 + 2 = 13. The plan would end at about 20, later than the 19.005 of `e2` doing everything. `e2` passes the second floor anyway, and stopping there costs it only the 4 seconds of Bob's boarding.

Section 10.9 — The MODEL method: time

Solution 10.9.1. (a) A cost. One agent, and the quantity to minimize is a sum (meters). Fidelity loses nothing, and Guidance and Portability gain: the plan can be proved optimal by many planners. (b) Durative actions. Three agents work in parallel, success is the finishing time, and drying is a duration during which nobody works but something must not happen (the burner of the kitchen): a cost can express none of the three. (c) A cost. Several agents, but the measure is total fuel, a sum that does not depend on what overlaps; time would add search effort and buy nothing. If a deadline appeared, the answer would change.

Solution 10.9.2. The item is in the room two seconds before the robot has let go of it, so somebody else can pick it up during the drop (the domain is `courier-time-dropstart-domain.pddl`). In the relay of `courier-time-p03` the planner uses it at once:

```
18.004: (drop r2 meds1 corridor) [2.000]
18.005: (pick r1 meds1 corridor) [2.000]
```

`r1` starts picking the box a millisecond after `r2` starts putting it down, and the makespan goes from 41.007 to 40.007. It is the fault of "both at start" in Table 10.5: a product available before it has been produced.

C.11. Solutions for Chapter 11

The files of these solutions are in `examples/ch11/solutions`.

Section 11.1 — Plans that exist only if actions overlap

Solution 11.1.1. There is no plan. The mend needs `(lit ?m)` at its start, and the start of `light` is what adds that fact: the two happenings interfere, so the mend starts at least ϵ after the match is lit. It then ends at least ϵ after the match goes out, and its invariant is broken during that last ϵ. With `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)
```

The exit code is 14. Kairos answers 10 only when it has a proof, and for this task its proof (forgetting durations and times) does not apply: without durations, a mend inside a light is possible. What makes the problem unsolvable is arithmetic on times, which we did by hand.

Solution 11.1.2. (a) Invalid: at 0.000 the mend starts at the same instant as the light, and its condition at start does not see a fact that is being added at that instant.

```
$ 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) Valid (`plan valid, makespan 5`): the invariant covers the open interval from 3 to 5, and at 5 itself, when the light goes out, the mend asks for nothing. The end of the mend and the end of the light do not interfere. (c) Invalid: the mend runs until 5.5, and the validator reports `over-all condition broken after the happening at 5.000000`.

Solution 11.1.3. The lamp is the match and the photo is the mend; the cooling makes the lamp usable once, like a match. The camera is a token.

```
;; 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)))))
```

Listing C.5: A greenhouse with flash lamps (`greenhouse-domain.pddl`).

Two photos fit in the 10 seconds of a lamp (8 seconds, plus separations), and the third needs the second lamp:

```
$ 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]
```

The second lamp is lit at 2.003, not at 0 and not when the first one goes out. The third photo can start at 8.003, when the camera is free, and ends at 12.003; 2.003 is the earliest instant from which the 10 seconds of the lamp cover it. The makespan is the end of the last envelope.

Section 11.2 — The envelope: a door held open

Solution 11.2.1. The plan that Kairos returns has makespan 62.006. `r1` holds the door from the corridor while `r2` goes in and picks; then `r2` holds it from inside and `r1` comes *into* the lab, to hold it from inside in turn while `r2` leaves:

```
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]
```

The plan is valid, and `r1` ends locked in the lab, which the goal does not forbid. A simpler plan keeps `r1` in the corridor and makes it hold the door twice in a row (`courier-door-p03-hand.plan`, in the folder of the chapter):

```
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
```

Fifteen seconds shorter. A temporal planner returns valid timetables, not the shortest one; in Kairos 1.0.6, plans that repeat the same envelope action with a content inside each repetition are among

those that its search misses. Reading the plan and trying a better one by hand, with the validator as judge, is part of step E.

Solution 11.2.2. It is solvable, by swapping the roles. `r2` cannot pass in 15 seconds, but it can hold a door, and `r1` is fast enough to pass (`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]
```

The sample changes hands in the corridor and the slow robot becomes the doorman. (The hold of `r1` at 10.001 is useless: nobody passes during it. A valid plan may contain actions that do nothing for the goal.) The problem would be unsolvable only if both robots were too slow.

Section 11.3 — Temporally simple, temporally expressive

Solution 11.3.1. `courier-time` and `kitchen`: no. Every fact is deleted at start and added at end; no action provides anything only while it runs. Kairos says that its engine M2 compresses the durative actions of both. `cellar`: yes, the envelope is `light`. For `courier-door`, yes: it is `hold-door`, and the log names its four ground instances (two robots, two sides). `courier-door-lock`: no; the door is opened at the end of one action and closed at the start of another, and the log says `engine M2: 66 durative actions compressed`. `courier-crate`: yes, `grip`, once for each robot.

Solution 11.3.2. With the envelope domain, both problems give the warning `engine M2 forced on a task that may need actions to overlap` and end with `no plan found`: as a block, `hold-door` opens and closes the door in one step, and no `pass` ever finds it open. With the lock domain, M2 solves both (best makespan 24.003, the same as with the default engine): `open-door`, `pass` and `release-door` can be put in a sequence, and the scheduler then lets them overlap where they do not depend on each other. The lock encoding is temporally simple; that is its portability.

Section 11.4 — Working together: two robots and a crate

Solution 11.4.1. The raise needs both grips at its start and over all its 4 seconds. It can start ϵ after the second grip, and must end no later than the first grip does, at 8 (the two ends may coincide, since the invariant covers the open interval). So the second grip starts at $8 - 4 - 0.001 = 3.999$ at the latest:

```
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]
```

The validator answers `plan valid, makespan 11.999`. With the grip at 4.000 and the raise at 4.001, the invariant of the raise is

broken when the first grip ends, at 8.000, and the plan is invalid. The slack of the model is those 3.999 seconds.

Solution 11.4.2. Three parameters need three inequalities, one for each pair.

```
(:durative-action raise
  :parameters (?c - crate ?r1 ?r2 ?r3 - robot ?l - location)
  :duration (= ?duration 4)
  :condition (and (at start (not (= ?r1 ?r2)))
                  (at start (not (= ?r1 ?r3)))
                  (at start (not (= ?r2 ?r3)))
                  (at start (crate-at ?c ?l))
                  (at start (gripping ?r1 ?c))
                  (at start (gripping ?r2 ?c))
                  (at start (gripping ?r3 ?c))
                  (over all (gripping ?r1 ?c))
                  (over all (gripping ?r2 ?c))
                  (over all (gripping ?r3 ?c)))
  :effect (and (at start (not (crate-at ?c ?l)))
               (at end (on-cart ?c)))))
```

Listing C.6: A raise that needs three robots (from `courier-crate3-domain.pddl`).

The plan has three grips at 0.000 and `(raise crate1 r1 r3 r2 dock)` at 0.001. There are six ground instances of `raise`, one for each order of the three robots, and all of them mean the same: with `-v`, Kairos counts 9 durative actions, 3 grips and 6 raises. The symmetry grows as $n!$; for larger teams, count grips with a numeric fluent (Chapter 8) instead of naming the robots.

Section 11.5 — Durations that the planner chooses

Solution 11.5.1. From the dock to ward A there are 4 + 10 + 12 = 26 meters, 52 for the round trip, and 52 units of battery. The robot has 20 and needs 32 more: 16 seconds at 2 units per second. The plan lasts 16 + 52 = 68 seconds. Charging again on the way would not help: the only charger is in the dock, and the robot does not pass by it until the trip is over. Splitting the 16 seconds into several charges before leaving changes nothing either, except for the separations it adds.

Solution 11.5.2. The swap is kept away from the pump by the token of the pump, asked for at start and over all, and `pour` already needs its cup under the spout over all.

```
;; Not while the pump runs: pour needs the cup there over all.
(:durative-action swap
  :parameters (?out ?in - cup)
  :duration (= ?duration 3)
  :condition (and (at start (under-spout ?out))
                  (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)))))
```

Listing C.7: Swapping cups (from `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 seconds for the espresso (40 ml) and 4 for the lungo (100 ml): in both cases the instant at which the lower limit of the range is

reached. Neither is a bound of its action.

Solution 11.5.3. The new effect is `(at end (increase (total-cost) (* ?duration (price ?l))))`, in `courier-charge-cost-domain.pddl`. Kairos returns the plan with the full charge, at a cost of 120 cents, and says why it is not a good one:

```
$ 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]
```

A cost that depends on a chosen duration is not a fixed cost of the action, and the search does not use it; it finds the plan with fewest actions, which is the one with a single long charge. The plan by hand is much cheaper:

```
$ 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
```

The lesson is the last pitfall of the chapter: a planner is not an optimizer of real numbers. When money depends on a duration, read the note, and help with blocks or by hand.

Section 11.6 — How planners order time

Solution 11.6.1. Time points: the start and the end of each grip (g_1, g_1', g_2, g_2') and of the raise (r, r'). Durations: $g_1' - g_1 = 8$, $g_2' - g_2 = 8$, $r' - r = 4$. Conditions at start of the raise, which needs the two grips that the starts of the grips add: $r - g_1 \geq \epsilon$ and $r - g_2 \geq \epsilon$. Invariants, which the ends of the grips would break: $g_1' - r' \geq 0$ and $g_2' - r' \geq 0$. Nothing relates the two grips to each other. Earliest times: $g_1 = g_2 = 0$, $r = 0.001$, $r' = 4.001$, $g_1' = g_2' = 8$, as in the plan.

Solution 11.6.2. The arrow from the start of `light` to its end becomes $[1, 5]$. The path through the mend needs $\epsilon + 2 + 0 = 2.001$ from the start of the light to its end, so the match must burn for at least 2.001 seconds, and with that value the makespan is 2.001: the plan ends when the fuse is mended. The network finds this value by itself; a duration that only the clock depends on is the easy case of a chosen duration.

Section 11.7 — The MODEL method: coordination

Solution 11.7.1. (a) A lock, or an envelope with a chosen duration: the operator decides when to lower the bridge and there is no natural length; with the lock, the goal must ask for the bridge down. (b) An envelope of fixed duration, 8 minutes: the pass ends by itself and nobody controls it (strictly, nobody starts it either: Chapter 13 has the right tool for windows that open at known times). (c) An envelope for the hold, with a chosen duration up to what the arm can stand, and the weld inside it: the two-robot crate. (d) No

coordination; a fixed duration, and neither shorter nor longer will do.

Solution 11.7.2. In `cellar-lock-domain.pddl`, `strike` adds `(lit ?m)` at its end and `blow-out` deletes it. The problem that had no plan has one:

```
$ 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]
```

Three fuses with one match, which burns for six seconds and is never put out. The model has lost the one thing that made the cellar a problem: the match ends by itself after 5 seconds, whether the plan likes it or not. A lock gives the agent control over the end; when the world has that control, only the envelope is faithful.

C.12. Solutions for Chapter 12

The files named here are in `examples/ch12/solutions`, and all of them are run by `tools/check_examples.sh ch12`.

Section 12.1 — A staircase where the world has a ramp

Solution 12.1.1. (a) In the world the battery at second t of the move is $50 - t$: 40 when the camera starts and 30 when it ends. It crosses 35 at $t = 15$, so the camera has enough battery only for the first half of its work. (b) The model that takes 30 units at the start says 20 during the whole move: the camera cannot work at all. It is wrong between seconds 10 and 15, but on the safe side. (c) The model that takes them at the end says 50 during the whole move: the camera can work. It is wrong between seconds 15 and 20 in the dangerous direction, since it accepts a plan that fails in the world.

Solution 12.1.2. With a capacity of 100 (`lamp-discrete-s01.pddl`) the careful model has a 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]
```

Six charges of one second take the battery from 10 to 40, and only then may the lamp start. The makespan is 26 seconds where the continuous model needs 20. This time the careful model did not lose the existence of a plan but its quality: it cannot overlap the charge and the lamp, so every plan of it is at least 6 seconds longer than necessary.

Section 12.2 — Continuous effects

Solution 12.2.1. The domain is `tank-drain-domain.pddl`. The new action mirrors `fill`: its own lock (`valve-closed`), a `decrease` with `#t`, and an upper bound that is the time to an empty tank:

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

With 80 liters and 4 liters per second the prediction is one action of 20 seconds, and `kairos` returns `0.000: (drain t1) [20.000]` for `tank-drain-p01.pddl`. The bound computed at the start is right here for the same reason as in `fill`: the goal can be reached without the tap, so nothing else needs to change the level meanwhile. If tap and valve could usefully be open together, the bound would have to become an `over all` condition, as in the greenhouse.

Solution 12.2.2. (a) A correct continuous effect: the product may be written in either order. (b) An error: `#t` under `at start`; `kairos check` answers `#t can only appear in increase/decrease effects of durative actions and processes`. (c) An error with the same message: there is no continuous `assign`. (d) A correct discrete effect, the one of Chapter 11: the whole amount arrives at the end.

Solution 12.2.3. The planner prints two plans. The first is one fill of 50 seconds, up to a full tank. The last has forty fills of one second, with a makespan of 40.039: forty times 2 liters are the 80 liters of the goal, and forty seconds are fewer than fifty. Kairos tries each free duration at its bounds, here 1 second or the time to a full tank; the duration that we would choose, 40 seconds, is neither, and no condition of an action marks it. The chain of minimum fills is the planner's way to build it. If a tap cannot be opened and closed every second, the lower bound of 1 second is a false statement about the world, and the fix belongs in the domain: a larger lower bound (`(>= ?duration 10)`), or a tap that can be used only once (delete a fact `(tap-unused ?t)` at the start and never restore it).

Section 12.3 — Reading a number while it changes

Solution 12.3.1. Let the fill start at s. Before it, the bed drinks alone and the level is $5-t$, so the tap must open before the tank is empty: $s \leq 5$. During the fill the level rises 1 liter per second. A fill of 30 seconds ends with $5-s+30$ liters, which fits in the tank only if $s \geq 5$: the only valid start is $s = 5$, the one that Kairos found. A fill of 20 seconds ends with $25-s$ liters, which always fits, and what is left lasts until the end of the watering: any start from 0 to 5 is valid. The plans `greenhouse-fill-at-0.plan` and `greenhouse-fill-at-5.plan` are accepted by `kairos validate`; `greenhouse-fill-at-6.plan` is rejected with `(>= (level ?t) 0): -1 >= 0 is false`.

Solution 12.3.2. The file is `greenhouse-reserve-domain.pddl`. One number changes in the action `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]
```

The tap opens at 3, when the level has fallen from 5 to the reserve of 2, and stays open 28 seconds, until the tank is full at $t = 31$. The last 9 seconds of watering leave 21 liters. The plan still waits for the last moment; the reserve moved the last moment.

Solution 12.3.3. With the charge from $t = 0$ the battery rises 3 units per second from 10, and reaches the capacity of 30 after 6.67 seconds. A charge of 10 seconds would go on to 40: not a plan. The validator says so for `lamp-charge10.plan`: `(<= (battery ?r) (battery-capacity ?r)): 40 <= 30 is false`. A charge of 6 seconds ends at 28 units, and the lamp alone uses exactly 28 in its remaining 14 seconds: a valid plan, the one of `lamp-together.plan`, which ends with an empty battery.

Section 12.4 — The generator: when to start

Solution 12.4.1. A pouring that starts at s during the run ends with $90 - s + 10$ liters, at most 100 for every $s \geq 0$. Alone, the generator empties the tank at $t = 90$. The window is from 0 to 90: with 90 liters the jerrycan may be poured from the very start of the run, and the tank is exactly full when the pouring ends. Before the run it cannot be poured: 110 liters do not fit.

Solution 12.4.2. The first pouring leaves 85 liters, which last until 95.001. The second jerrycan fits as soon as the tank has room for its net gain of 10 liters, and it must start before the tank is empty; to end before the generator stops, at 110.001, it must start before 100.001. The plan `generator-p02-middle.plan` pours it at $t = 50$:

```
$ kairos validate generator-domain.pddl generator-p02.pddl
    solutions/generator-p02-middle.plan
plan valid, makespan 110.001
```

Solution 12.4.3. The file is `generator-safe-domain.pddl`. The rule is one more condition of the action `refuel`:

```
    (over all (stopped ?g))
```

The fact `(stopped ?g)` is deleted when the run starts and never comes back, so jerrycans can only be poured before the run. With 60 liters (`generator-safe-p02.pddl`) both jerrycans fit, the tank is exactly full, and the plan is two pourings and then the run, with a makespan of 120.002. With 95 liters (`generator-safe-p01.pddl`) not even one jerrycan fits before the run, the 95 liters do not last 100 seconds, and Kairos answers `no plan found`. The rule turned a problem of timing into a problem without a solution, which is what a safety rule sometimes does; the model is right to say so.

Section 12.5 — The oven: when to stop

Solution 12.5.1. The oven keeps its temperature when the heater is off, so the bread needs a final temperature between 200 and 220: a heating between 90 and 100 seconds. If the bread may go in while the

heater is on, the condition is still the same for the final temperature, because it must hold until the end of the baking, but the baking can start earlier. In `oven-overlap.plan` the heater runs 100 seconds and the bread goes in at 90: during the baking the temperature climbs from 200 to 220, never outside the range, and the plan is valid with a makespan of 100 seconds, one thousandth better than the plan of the chapter. A heating of 101 seconds would end at 222 degrees and no start of the baking could save it.

Solution 12.5.2. The domain is `oven-cool-domain.pddl`. The action `cool` is `heat` with a `decrease`; it takes the heater lock, so that heating and cooling do not overlap, and needs the oven empty:

```
(: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]
```

At 1 degree per second, 30 seconds take the oven from 150 to 120, the first temperature at which the meringue can go in. Again the duration is the one at which ending the action makes a condition of another action true.

Section 12.6 — What the planner must find

Solution 12.6.1. For the greenhouse the output has a line that starts with `durations: (fill t1):` and ends with `planned as PDDL+`: only steps of time. For the courier it has the lines of Section 12.6, with `engine M2` and `also in steps of 0.1 s`: discrete changes, with the steps beside them. The difference is in who reads the number. The level of the tank is read as large enough by `irrigate` and as small enough by `fill`: no instant is safe for a jump, since moving the water early may overflow and moving it late may run dry. The battery of the courier appears in conditions only as large enough, so a drain placed at the start of a move and a charge placed at its end can never hide a violation.

Solution 12.6.2. Without a lower bound (`courier-nobound-domain.pddl`) the plan is:

```
$ 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]
...
```

One charge of 16 seconds, up to a full battery, and a plan 11 seconds longer than with the bound. The only bound left to try is the upper one. As a statement about the world, the lower bound said that docking is not worth less than 5 seconds. For the search it was a second option, a short charge, and chains of short charges gave the planner amounts in between. A model that wants short charges must offer them: a free duration is only as free as its bounds and the conditions around it make it.

Section 12.7 — The courier world

Solution 12.7.1. In `courier-margin-domain.pddl` the condition of `move` is `(over all (>= (battery ?r) 10))`. The plan for `p05` does not change: its lowest values were 14, on arriving at the dock, and 13, at the end, both above the reserve. The plan for `p02` changes: one charge of 5 seconds left the robot with 7 units in ward A, below the reserve, so the last plan now has two charges of 5 seconds before leaving (from 20 to 70) and a makespan of 52.008 instead of 47.007. The robot arrives with 32.

Solution 12.7.2. The battery goes from 60 to 100 during the charge (0 to 8 seconds). It then only goes down, one unit per second of movement, with flat pieces at the two drops and the two picks: 90 after the way to the pharmacy, 57 on arriving at the ICU, 32 at the lab, and 10 at the dock, at $t = 106.016$, which is the lowest value. The round needs 90 units and the robot had 60, so 30 units, 6 seconds of charge, were enough: 2 seconds could be saved.

Solution 12.7.3. In `courier-handling-domain.pddl` both actions get the effect `(decrease (battery ?r) (* #t 0.5))` and the condition `(over all (>= (battery ?r) 0))`. The condition is needed: the question of the method box is whether anybody reads the quantity while the action runs, and here the action itself must respect a limit all along. Without it a robot with half a unit could start a pick and end it with a battery of -0.5. Since a pick takes `(idle ?r)`, nothing else changes the battery meanwhile, and a discrete version (ask for 1 unit at the start, take it at the start) would have the same plans. For `p02` the plan is the same as before, and the robot ends with 5 units instead of 7.

Section 12.8 — The MODEL method: continuous change

Solution 12.8.1. (a) The fuel changes during the trip, but nobody reads it meanwhile and nothing else changes it: a discrete decrease at the start, with a precondition for the whole trip. (b) The same: ink is used during printing, the cartridge is swapped only between jobs. Discrete. (c) The heater adds heat while the shower takes it, and the shower needs the water warm enough all along: producer and consumer overlap, and a limit holds in the middle. Continuous. (d) Two consumers overlap and no producer does. Each can take its whole amount at its start: if the battery has enough for both totals, it never runs out on the way, and if it does not, one of them could not finish anyway. A careful discrete model is exact here. It stops being exact the day the drone can charge in flight, or when the goal only needs part of an action.

Solution 12.8.2. The domain is `courier-discrete-domain.pddl`. The move asks for its whole consumption and takes it at the start; the charge credits its amount at the end:

```
    (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))))
```

On `p02` and `p05` it gives the plans of the chapter, with makespans 47.007 and 51.008, and it rejects the cheating plan at the same move, now with a precondition: `8 >= 12 is false`. *Fidelity*: equal, because a robot that does one thing at a time has a battery that nobody else touches or reads during an action. *Size*: equal. *Guidance*: the discrete model is an ordinary temporal problem; the continuous one is turned into it by Kairos. *Clarity*: the continuous model states a rate and a limit; the discrete one repeats the product of duration and rate in a condition and in an effect, and the two copies must be kept equal by hand. *Portability*: the discrete model needs `:duration-inequalities` and `?duration` in an effect, which more planners accept than `:continuous-effects`. For a planner without continuous effects the discrete version is the one to give, and for this courier nothing is lost.

C.13. Solutions for Chapter 13

The files named here are in `examples/ch13/solutions`, and all of them are run by `tools/check_examples.sh ch13`.

Section 13.1 — Facts that follow from other facts

Solution 13.1.1. (a) A consequence of `robot-at` and `item-at`: no robot and no item is in the room. (b) Basic: a door is open because somebody opened it. (c) A consequence of the load of the robot and its capacity; with the numbers of Chapter 8 it is a comparison, and needs no predicate at all. (d) A consequence of `robot-at`: there is a room in which both are. (e) Basic: it is one `item-at` fact.

Solution 13.1.2. With one item, `pick` always deletes `free` and `drop` always adds it. With two, what an action must do depends on the state: `pick` deletes `free` only if the robot already carries an item, and `drop` adds it always, but the fact was already there if the robot carried one. The effect of `pick` becomes conditional on how many items the robot holds, which the predicates cannot count without more bookkeeping (the slots of Chapter 7, or the numbers of Chapter 8). The fact is still a simple consequence of the `carrying` facts, "there are not two different items that the robot carries", and a rule states it in one line while the effects need a second mechanism.

Section 13.2 — Derived predicates

Solution 13.2.1. The domain is `courier-tidy-domain.pddl`: the typed courier domain, with `free` kept by effects as always, plus

```
(:derived (tidy ?l - location)
  (not (exists (?i - item) (item-at ?i ?l))))
```

For `courier-tidy-p01.pddl`, with the goal `(and (tidy corridor) (tidy lobby) (free r1))`, Kairos returns a plan of 9 actions that carries each item from the corridor to ward B and drops it there. The goal says where the items must not be and that the robot must end with empty hands; it does not say where the items go, and the planner chose a neighbor room of the corridor that the goal does not mention. Without `(free r1)` in the goal the cheapest plan would leave one item in the hands of the robot.

Solution 13.2.2. The check answers `derived predicate 'free' cannot be initialized` and rejects the problem. The redundancy is harmless only while it agrees with the rule. The same line in a problem where `r1` starts carrying an item would contradict the rule, and the language would have to say which of the two wins. By forbidding the statement it never has to: a derived fact has exactly one source of truth.

Solution 13.2.3. With one rule (`courier-keycard-domain.pddl`):

```
(:derived (may-enter ?r - robot ?l - location)
  (or (not (restricted ?l)) (has-keycard ?r)))
```

With two (`courier-keycard2-domain.pddl`):

```
(:derived (may-enter ?r - robot ?l - location)
  (not (restricted ?l)))

(:derived (may-enter ?r - robot ?l - location)
  (has-keycard ?r))
```

Both give the same plan for the test problem, in which only the robot with the keycard reaches the lab. Two rules read as a list of reasons, and a third reason (an escort) is a third rule that touches nothing else; they also avoid `:disjunctive-preconditions`. One rule keeps the whole definition in one place, which helps when the reasons are few and will not grow.

Section 13.3 — Rules that call themselves

Solution 13.3.1. Round 1, one passable step from ward A: the corridor. Round 2: the lab and ward B, and also ward A itself (to the corridor and back). Round 3: the ICU, through its open door. The lobby is behind the closed door, and with it the pharmacy and the dock: they are never derived. So `(reachable ward-a ward-a)` is true, as it is for every room that has a passable neighbor. For `go` it means that `(go r1 ward-a ward-a)` is applicable. It deletes and adds the same fact, changes nothing, and no plan gains from it; a condition `(not (= ?from ?to))` would remove these ground actions.

Solution 13.3.2. The domain is `pipes-domain.pddl`. Its only rule joins the two steps of the circuit, the link and the propagation:

```
(: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)
```

Water is sent through the shed, and *both* valves of the pool are closed: the one from the inlet and the one towards the garden, through which the garden would fill the pool from behind. It is the lesson of the kitchen and the oven again.

Solution 13.3.3. Three blocks are in the same tower exactly when every two of them are ordered by `above`, one way or the other (`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)))))
```

From a table with `c` on `b`, and `a` and `d` alone, the plan is one action, `(move-from-table a c)`. As for `p02`: in a tower, `above` is a strict order, so no state has `a` above `b` and `b` above `a`. Kairos proves it because this is a classical problem. Its state space is finite, the search visits all of it, and none of its states satisfies the goal: that is a proof. In Chapter 12 the plans depended on real-valued times, the search could only sample them, and running out of samples proved nothing.

Solution 13.3.4. Each rule uses the other predicate under a negation, so `on` depends negatively on `off`, which depends negatively on `on`: each depends negatively on itself. The rules are not stratified, and there is no order in which to compute them. The round-by-round procedure shows the trouble. In the first round nothing has been derived, so "not off" and "not on" both hold and both facts are derived for every lamp; but then each fact has destroyed the reason for the other. Kairos rejects the file `lamps-wrong-domain.pddl` with `derived predicate 'on' depends negatively on itself`, and the same for `off`. One of the two must be basic, and the other its negation.

Section 13.4 — What a planner does with rules

Solution 13.4.1. `courier-reach`: 92 actions and 518 axioms. `courier-steps`: 42 actions and 392 axioms. The actions differ by 50: `go` is grounded for all 64 ordered pairs of rooms, `move` only for the 14 connected ones, and the rest of the actions are the same. The rules differ by 126, which is what the planner built for the recursive rule; the 392 that both domains share are those of `passable`. The recursive predicate added a third to the rules and more than doubled the actions, and bought plans that are shorter and a vocabulary in which the goal can speak of reachability.

Solution 13.4.2. The recursive rule has three variables, so up to 100^3, a million ground rules, on top of the 100^2 pairs. If the robot does not open or close doors, reachability never changes during a plan: it is a static relation. It can then be computed once, outside the planner, and written in the initial state as basic facts `(reachable a b)`, at most ten thousand of them and usually far fewer, with no rule at all. If some doors do change, a middle way is to group the rooms into zones separated by those doors and to derive reachability between zones, where n is small.

Section 13.5 — A world with a calendar

Solution 13.5.1. The problem is `courier-hours-s01.pddl`. The pharmacy is open at time zero, so it is an initial fact, followed by three timed literals:

```
(is-open pharmacy)
(at 8 (not (is-open pharmacy)))
(at 40 (is-open pharmacy))
(at 60 (not (is-open pharmacy)))
```

The pharmacy is 10 meters from the dock, so the robot arrives at 10, two seconds after the first window closed: it cannot use it. It picks at 40, and the rest takes 32 seconds as in `p01`. The best plan that Kairos prints has a makespan of 72.004.

Solution 13.5.2. It is valid: `courier-hours-wait.plan` starts moving at 20 and `kairos validate` accepts it with a makespan of 62.006, two thousandths worse than the plan of the chapter, because the separations between consecutive actions now come after the opening time instead of before it. By the makespan the two plans are equal for any practical purpose. Outside the model there are reasons both ways. Waiting at the pharmacy protects the plan against a slow trip: a robot that leaves at 20 and is delayed loses the difference. Waiting at the dock keeps the robot on its charger and out of the way of people at the door. Neither reason is in the model, so the planner cannot weigh them; a modeler who cares adds the battery, or a preference (Chapter 14).

Section 13.6 — Windows and deadlines

Solution 13.6.1. The plan has no slack: the pick cannot start before 30, and every action follows the previous one. The delivery at the lab ends at 86.008 and needs the lab open throughout, so the lab must close after that instant. With whole seconds, a lab that closes at 87 still has the plan, and one that closes at 86 has none. For the pharmacy, a window that opens at T ends the last delivery at $T + 56.008$, which must come before 90: with whole seconds the latest opening is 33, and Kairos then returns a plan with makespan 89.008.

Solution 13.6.2. The files are `rover-domain.pddl` and `rover-p01.pddl`. Both actions have the condition `(over all (daylight))`, and the problem has the fact `(daylight)` at time zero and three timed literals, at 100, 150 and 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]
```

The pictures are taken on the first day, by 80. A drive back would need light until 110 and the night falls at 100, so the rover cannot even start it: the condition is `over all`. It spends the night at the rock and returns on the second day.

Solution 13.6.3. The plan is `satellite-volcano-first.plan`: observe the volcano as soon as it is visible, at 12, turn, and observe the crater

at 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
```

The window of the crater closed at 20. After the volcano, the camera cannot point at the crater before 21, and the exposure would have to start at 16 at the latest.

Section 13.7 — Numbers on the calendar

Solution 13.7.1. The problem is `courier-hours-s02.pddl`. Kairos prints two plans. In the first (makespan 58.006) the robot never stops: the move from the pharmacy to the lobby starts at 12, in the crowd, and takes 12 seconds, and the next one, 10 meters, starts at 24 and takes 20. In the second (54.003) the robot reaches the lobby at 24 and *waits* until 30.001, when the crowd has left, to cross to the corridor in 10 seconds. Starting at 24 it would arrive at 44; starting at 30 it arrives at 40. A robot in a hurry should stand still for six seconds. This follows from the durations being fixed at the start of each move, and a planner finds it where intuition does not.

Solution 13.7.2. For example: the charge rate of the dock, if the building limits power in the hours of high demand; the number of nurses on a ward, if a handover needs one to be available; the price of electricity, in a metric; the speed of the elevators, slower at shift changes. Each is a number that the planner reads and that nobody in the plan changes.

Section 13.8 — The MODEL method

Solution 13.8.1. It could (`courier-arrived-domain.pddl`, where `deliver` is again a plain `drop`):

```
(:derived (delivered ?i - item)
  (exists (?l - location)
    (and (destination ?i ?l) (item-at ?i ?l))))
```

The problem with the opening hours is solved with the same plan as before. Two things are lost. A derived fact has no memory: `delivered` is now true only while the item lies at its destination, and becomes false if a robot picks it up again, whereas the effect of `deliver` recorded an event. And the deadline has lost its place: it was a condition `at end` of the action that delivered, and a rule has no moment at which to check the calendar. The goal would be reached by an item that arrives late. A deadline needs either an action that marks the event, as in the chapter, or a constraint on the plan such as `within`.

Solution 13.8.2. (a) The calendar: a timed literal that removes `(is-open cafeteria)`. (b) The state: a rule over the positions of the robots. (c) The agent: an effect of an action. (d) The calendar. (e) The state: a rule over what has been delivered to the ward.

C.14. Solutions for Chapter 14

The PDDL files of these solutions are in `examples/ch14/solutions`.

Section 14.1 — More wishes than battery

Solution 14.1.1. Dock to lobby 4, lobby to pharmacy 6: 10 units to reach `meds2`. Pharmacy to the ICU through lobby, corridor and ward B: $6 + 10 + 12 + 5 = 33$, total 43. Back to ward B for the sample: 5, total 48. Ward B to the lab through the corridor: $12 + 8 = 20$, total 68, the value of the figure.

Solution 14.1.2. By hand: `meds1`, then `meds2`, then `sample1` needs $99 + 5 + 20 = 124$. The order `meds2`, `sample1`, `meds1` reaches the lab at 68, returns to the pharmacy ($8+10+6 = 24$) and goes to ward A ($6+10+12 = 28$): 120. Kairos agrees: `courier-battery-full.pddl` (battery 120) has a plan, in that order, and `courier-battery-short.pddl` (battery 119) is unsolvable. The order is cheap because the sample is picked up on the way back from the ICU: ward B is crossed anyway, and the only trip made with empty hands is the return to the pharmacy. The unsolvable answer for 119 is also information that no hand computation gives easily: it covers every order and every trick, such as leaving an item halfway and coming back for it.

Section 14.2 — A preference in the goal

Solution 14.2.1. The empty plan leaves the robot at the dock and violates the four preferences: cost 4. The best plan of `p02` ends in the lab with $100 - 82 = 18$ units, and the lobby is exactly 18 meters away ($8 + 10$): the robot can make two deliveries *and* end in the lobby, so the best cost is still 1. `courier-battery-lobby.pddl` confirms it: Kairos prints plans of cost 4, 3, 2 and 1, and the last one ends with `(move r1 corridor lobby)` on an empty battery.

Section 14.3 — What each wish is worth

Solution 14.3.1. The row "`meds1`, `sample1`" violates only `meds-icu`, so that weight must be smaller than each of the other two (the other two feasible pairs violate `meds-a` or `sample-lab`). For instance 3 for `meds-a`, 1 for `meds-icu` and 5 for `sample-lab`, as in `courier-battery-weights.pddl`: the best plan costs 1 and delivers the sample and `meds1`.

Solution 14.3.2. Work from the bottom. Optional: weight 1; all the optional deliveries together are worth 10. Normal: any weight above 10, say 11; all normal and optional ones together are worth $4 \times 11 + 10 = 54$. Urgent: any weight above 54, say 55. The weights grow fast: with many classes and many goals per class the numbers become huge, which is the practical limit of strict priorities written as weights.

Solution 14.3.3. Let w be the weight of `parked`. The best plan that does not return makes both medicine deliveries and costs $1 + w$. The best plan that returns serves the ICU and the lab and costs 3. The robot returns when $3 < 1 + w$, that is, for $w > 2$; at $w = 2$ the two plans tie. With $w = 1$ (`courier-battery-parked1.pddl`) the best plan costs 2 and ends in the ICU; with $w = 4$ (`p04`) it costs 3 and ends at the dock.

Section 14.4 — Wishes against effort

Solution 14.4.1. Delivering the meal after the medicine adds 103 − 40 = 63 to `total-cost`. The meal is delivered when its weight is above 63. With exactly 63 (`courier-cost-meal63.pddl`) both plans cost 103, and Kairos returns the one it finds first, the single delivery, because the anytime search only replaces a plan by a strictly better one. With 64 (`courier-cost-meal64.pddl`) it prints the plan with both deliveries, cost 103 against 104.

Section 14.5 — One preference, many instances

Solution 14.5.1. One medicine: 45 + 70 = 115. Both: 45 + 68 = 113, with no violation. Nothing: 140. The best plan delivers both medicines, with cost 113 and `(is-violated meds-icu)` equal to 0 (`courier-cost-meds70.pddl`).

Solution 14.5.2. `courier-cost-allmeds.pddl`:

```
(:goal (preference meds-icu
         (forall (?m - medicine) (item-at ?m icu))))
```

There is now one preference, violated unless *both* medicines arrive. Delivering one costs 45 and still pays the 50: 95. Delivering both costs 113. Doing nothing costs 50, and it is the optimum: Kairos prints only the empty plan. A plan that delivers one medicine can never be optimal, since it pays the effort and gets nothing for it. With this form the weight has to be above 113 before the robot moves at all.

Section 14.6 — Wishes about the way

Solution 14.6.1. With weight 6 (`courier-lobby-w6.pddl`) the back door costs 42 and the lobby 36 + 6 = 42: a tie. Kairos prints one plan, the back door, and stops: the other route is not strictly better. With weight 7 (`courier-lobby-w7.pddl`) the lobby costs 43, and the back door is the only optimum. Both outputs are the same five actions with cost 42; the difference is that at 6 another optimal plan exists.

Solution 14.6.2. The domain is `courier-night-domain.pddl`; its `move` is:

```
:precondition (and (robot-at ?r ?from) (connected ?from ?to)
                   (preference silence (not (quiet ?to))))
```

The domain needs `:negative-preconditions` besides `:preferences`. In `courier-night-p01.pddl` the facts `(quiet ward-a)` and `(quiet ward-b)` are in the initial state, the goal is hard and the metric is `(+ (total-cost) (* 25 (is-violated silence)))`. The best plan costs 126: 74 meters, 2 handlings, and *two* violations. The ICU is reached only through ward B, so the robot enters ward B on the way up and again on the way back to the dock. The second entry is the price of the hard goal `(robot-at r1 dock)`; if that goal were a preference worth less than 31 + 25 = 56, the robot would stay in the ICU.

Solution 14.6.3. The soft goal says "the robot should not be in the lobby when the plan ends". It speaks of one state, the last one. A robot that crosses the lobby five times with samples and ends in the lab satisfies it; a robot that never carried a sample and stops in the lobby violates it. The rule is about every moment at which a move

is made, and only a preference in the precondition of `move` (or a constraint of the next chapter) looks at those moments.

Section 14.7 — Two more worlds

Solution 14.7.1. `tourist-free.pddl`. The best plan now costs 2:

```
(bus hotel museum)
(visit museum)
(bus museum tower)
(visit tower)
(bus tower market)
(visit market)
(bus market cathedral)
(visit cathedral)
```

The tourist takes four buses (20 minutes), sees the four sights (220 minutes) and ends the day at the cathedral with no time to go back: the wishes `on-foot` and `dinner` are violated, 1 each. A dinner worth 1, less than the cheapest sight, is given up for the market. If that plan looks absurd, the weight was wrong, or the dinner was not a wish in the first place: this is the price test of step E.

Solution 14.7.2. `satellite-long.pddl`. Kairos finds a plan of cost 1 with five images in 76 seconds: `crops`, `flood`, `volcano`, `glacier` and `fire`. Only `harbor` is left out. Six images need 60 seconds of imaging alone, which leaves 20 for six slews, and the search ends without finding such a plan.

Section 14.9 — The MODEL method: wishes and their worth

Solution 14.9.1. *Fidelity*: the hard goal loses solutions (with it, both medicines can never be delivered); the preference is exact if the weight is right; two problems in sequence are exact for the first and blind for the second: the first plan may strand the robot in the ICU with a battery of 1, and then the second problem has no plan. *Size*: the same for the three, one goal atom or one pair of collect and forgo actions. *Guidance*: the hard goal guides best (every state is judged by its distance to the dock); the preference adds one more thing to weigh; the sequence gives two small searches. *Clarity*: the hard goal is the clearest; the preference needs its weight explained. *Portability*: only the preference needs `:preferences`. Verdict: hard goal if a robot away from the dock is a failure (no charger elsewhere, the night shift cannot fetch it); preference if it is a convenience; never the sequence when the two problems share a resource such as the battery, because the first plan does not know what the second one will need.

Solution 14.9.2. *Success, required*: the report is done. *Wished*: exercises (weight 6, "I should"), gym (4), reading (3), laundry (1), and not working while tired (2 for each task done tired). The weights are one reasonable reading of "should" and "would like". The model is `evening-domain.pddl` with `evening-p01.pddl`: a numeric fluent `(minutes-left)`, one action `work` whose conditional effect makes the student `(tired)` after a tiring task, a preference `fresh` in its precondition, and an action `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 returns, as its best plan, cost 3: report, exercises, laundry, and the gym *last*. The four tasks take exactly 180 minutes, the reading is given up, and nobody works tired because the tiring task was put at the end. The planner was not told to order the evening that way: the order follows from a preference about the way.

C.15. Solutions for Chapter 15

The PDDL files of these solutions are in `examples/ch15/solutions`.

Section 15.1 — A rule that the plan breaks

Solution 15.1.1. (a) A goal: it is about the last state. (b) A precondition of `pick`, with a counter or slots (Chapter 7, Chapter 8): only `pick` adds an item, so one action guards the rule. It is also an `always`, but here the precondition is natural. (c) The whole plan: `always`; a goal would allow leaving the sample there for a while. (d) A precondition: it is what the action needs to be possible. (e) The whole plan: `sometime`. (f) The whole plan: `(sometime-before (robot-at r1 ward-b) (robot-at r1 ward-a))`.

Section 15.2 — The states of a plan: always

Solution 15.2.1. `courier-rules-apart.pddl`:

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

The plan still has 16 actions, but a different shape. Under the rule of the ICU the robot loaded both items at the pharmacy and unloaded the waste first. Now the two may not share the basket at all, so it takes the medicines alone to the ICU, returns to the pharmacy for the waste and ends at the dock: the plan of `courier-rules-p02-ok.plan`. That plan satisfies both rules; the plan printed for `p02` satisfies only the rule of the ICU.

Solution 15.2.2. `unsolvable`. The robot is at the dock in S_0, the initial state belongs to the trajectory, and `always` must hold in every state: the constraint is false before any action. (The goal `(robot-at r1 dock)` contradicts it as well, at the other end.)

Section 15.3 — What must happen, and in which order

Solution 15.3.1. `courier-rules-no-return.pddl` is unsolvable. The ICU hangs from ward B and from nothing else, so stepping into the ICU and out again puts the robot in ward B twice, with the ICU in between: two stretches. The same happens with the dock, where the robot starts and where the waste must end. On a map shaped like a tree, any round trip revisits rooms.

Solution 15.3.2. (a) The robot is never in the lab: the first state in which it is there would need an *earlier* one in which it was already there. A convoluted way to write `(always (not (robot-at r1 lab)))`. (b) Nothing: when the robot is in the ICU, the same state provides the "after", since $j \geq i$ allows $j = i$. The constraint is always satisfied. (c) Nothing either: S_0 has `(sanitized r1)` and is earlier than any state in which the robot carries something, since it carries nothing in S_0. The rule looks like "sanitize before touching the medicines" and asks for nothing.

Solution 15.3.3. The problem is `courier-rules-clean.pddl`. An `imply` is a disjunction, so the requirement is `:disjunctive-preconditions`. The early plan is rejected: in its last two states the robot is in the lab and the gripper is dirty since the pick. The new plan picks the sample *first* and sanitizes on the way back, with the sample in the basket:

```
(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)
```

Same length, same actions, another order, and only this order is what the lab wanted. A `drop` does not dirty the gripper in this domain, so the robot stays clean while it is in the lab.

Section 15.4 — Rules with a clock

Solution 15.4.1. `courier-rules-within6.pddl` is unsolvable. From the dock the delivery needs two moves to the pharmacy, a pick, three moves to ward A and a drop: 7 actions, so the medicines are in ward A in S_7 at the earliest, and $t_7 = 7 > 6$.

Solution 15.4.2. The delivery needs $4+6+2+6+10+12+2 = 42$ seconds of actions. With a deadline of 43 (`courier-timed-within43.pddl`) Kairos finds a plan that goes straight for the medicines and drops them at 42.006. The six thousandths are the separations between one action of the robot and the next: each action starts 0.001 after the previous one ends, because the end of one and the start of the next touch the same facts (Chapter 10). So the medicines cannot be in ward A at 42 sharp, and with a deadline of 42 the search of Kairos runs until its time limit without a plan.

Solution 15.4.3. The first is one deadline, counted from the start of the plan: the gripper is clean in some state among the first five steps. The second is a deadline that starts again in every state in which the robot is in the ICU: five steps later at most, the gripper is clean. A plan that sanitizes at step 2, visits the ICU at step 9 and never sanitizes again satisfies the first and violates the second. A plan that never enters the ICU and never sanitizes satisfies the second and violates the first.

Section 15.5 — The ten operators, side by side

Solution 15.5.1. The robot is at: dock (S_0), lobby, pharmacy (S_2, S_3), lobby, corridor, ward B, ICU (S_7, S_8), ward B, corridor, lobby, pharmacy (S_{12}, S_{13}), lobby, dock (S_{15}, S_{16}). It carries `meds1` from

S_3 to S_7 and `waste1` from S_{13} to S_{15}. (a) Violated: the robot is never in the lab. (b) Violated: two stretches in the pharmacy. (c) Satisfied: the robot carries the medicines from S_3, before it is in the ICU in S_7. (d) Satisfied: the medicines are in the ICU in S_8, and $t_8 = 8$. (e) Satisfied: the last state in which the waste is carried is 1 step from S_{16} and the first is 3 steps from it. (f) Satisfied: after time 12 the robot is in the pharmacy, the lobby and the dock. The files `courier-rules-quiz-a.pddl` and `-quiz-b.pddl` hold (a) and (b), and `kairos validate` rejects the plan for each; `courier-rules-quiz-true.pddl` holds the other four together, and the plan is valid for it.

Section 15.6 — Where rules live, and rules that bend

Solution 15.6.1. "One heating cycle per shift" is a policy about energy that depends on the day: a shift with an urgent order may allow two cycles, and a problem with more parts than the oven holds would have no plan. Policies that change belong in the problem, perhaps as preferences. "Never the door open while hot" protects people: it does not depend on the order of work, nobody should be able to forget it when writing a problem, and it should never be soft.

Solution 15.6.2. `courier-rules-p11-waste.pddl`. The two wishes still exclude each other, so the planner violates the cheaper one, now `fast-meds` at 3. The best plan takes the waste to the dock first (6 steps), then the medicines to the ICU, and costs 3. The hard rule of the ICU never entered the bargain.

Section 15.7 — Two more worlds

Solution 15.7.1. `rover-two-rules.pddl`: five actions, the plan of the section without its last `drive`. The detour by the ridge is still due to `sometime-before`; without rules the rover would go by the plain. The rule `at-most-once` costs nothing here, because the short plan enters the crater only once anyway: a constraint that the best plan already satisfies is free.

Solution 15.7.2. The smallest bound is 7 (`workshop-fresh7.pddl` has a plan, `workshop-fresh6.pddl` has none). In the 11-action plan the parts are loaded at steps 2, 3 and 4 and unloaded at steps 9, 10 and 11. The deadline of each part runs from the first state in which it is in the oven, so the part loaded at step 2 must be out by step 9, the next by 10, the last by 11: the parts must come out *in the order in which they went in*. The plan of Kairos does exactly that (`lid`, `gear`, `bracket`, both times). With 6, the first part would have to be out at step 8, when the door has just been opened.

Section 15.9 — The MODEL method: rules about the whole plan

Solution 15.9.1. `rover-auto-domain.pddl` with `rover-auto-p01.pddl`. Two static predicates mark the waypoints, `(risky ?w)` and `(lookout ?w)`, and one fact is the memory of the automaton, `(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))))
```

The plan is the same five actions as with the first two rules of the constraint version. If the rover starts at the ridge, this encoding does *not* know it: `(looked)` is set only by driving *to* a lookout, so the problem must put `(looked)` in its initial state by hand. The constraint needs no such care, since S_0 is part of the trajectory. That is the weak point of every hand-made automaton: its initial state is one more thing to get right, and nothing checks it.

Solution 15.9.2. *Wash hands after raw meat, before anything else*: a rule of the kitchen (domain), hard. No single operator: "before the *next* touch" is about what happens in between. A fact `(hands-dirty)`, set by touching meat and cleared by washing, and `(always (imply (touching ?x) (or (raw-meat ?x) (not (hands-dirty)))))`, or simply a precondition of the touching actions. *Oven never on with nobody in*: domain, hard, one state: `(always (imply (oven-on) (cook-in-kitchen)))`. *Tasted before served*: domain, hard, history: `(forall (?d - dish) (sometime-before (served ?d) (tasted ?d)))`; here "at some earlier time" is what is meant. *Fridge opened as few times as possible*: a wish, not a rule. `at-most-once` as a preference prices only the difference between one opening and more; to count openings, a preference in the precondition of the opening action (Chapter 14) or an action cost. *Served by 14:00*: of today (problem), hard or soft depending on the guests: `(forall (?d - dish) (within 840 (served ?d)))` with time in minutes from midnight, in a temporal model.

C.16. Solutions for Chapter 16

The PDDL files of these solutions are in `examples/ch16/solutions`.

Section 16.1 — One robot in two rooms

Solution 16.1.1. *A robot is in one room*: `move` adds `(robot-at ?r ?to)` and deletes `(robot-at ?r ?from)`; no other action touches `robot-at`. *An item is in one room or with one robot*: `pick` deletes `(item-at ?i ?l)` and adds `(carrying ?r ?i)`; `drop` does the reverse. *Free exactly when carrying nothing*: `pick` requires and deletes `(free ?r)` as it adds `carrying`; `drop` adds it back as it deletes `carrying`. The precondition `(free ?r)` of `pick` is part of the invariant too: without it a robot would carry two items and become free after dropping one. All three also depend on the initial state, which must satisfy them to begin with.

Solution 16.1.2. Without that delete, picking an item leaves a copy of it in the room: after the pick both `(item-at meds1 pharmacy)` and `(carrying r1 meds1)` hold. Any goal that needs one item in two places exposes it: `(and (item-at meds1 pharmacy) (item-at meds1 ward-a))` has a seven-action plan (`courier-copy-domain.pddl` with `courier-copy-p01.pddl`). So does a quieter one: a goal that asks for the medicines in ward A in a problem where a rule (a constraint of Chapter 15, say) wants the pharmacy never to be without them.

Section 16.2 — A function whose value is an object

Solution 16.2.1. `courier-of-hands.pddl`, with the goal

```
(:goal (and (= (holder sample1) r2)
            (= (location-of r2) lab))))
```

`r2` fetches it: two moves to ward A, a pick, two moves back, five actions, and no `drop`. There is no action that passes an item from one robot to another, so the robot named in the goal has to pick the item itself, and `r1` is of no use. With predicates the same goal would be `(carrying r2 sample1)`; the fluent adds nothing here except that "in the lab" and "with `r2`" are visibly two values of the same thing.

Solution 16.2.2. `courier-nofree-domain.pddl`. `pick` tests that the robot holds nothing, and neither action has an effect on `free`:

```
:precondition (and (= (location-of ?r) ?l) (= (holder ?i) ?l)
                   (not (exists (?j - item)
                          (= (holder ?j) ?r))))
:effect (assign (holder ?i) ?r))
```

The condition is safe because `holder` is total: every item has a holder in the initial state and no effect makes it undefined, so the comparison inside the `exists` never meets a fluent without a value. The third invariant of Exercise 16.1.1 is gone: there is no `free` to go out of step. In `courier-nofree-p01.pddl` the robot moves two items one at a time, in nine actions.

Section 16.3 — Fluents as terms

Solution 16.3.1. `courier-of`: `move` has one instance per robot and connection, 14; `pick` and `drop` one per robot, item and room, 8 each: $14 + 8 + 8 = 30$. Their other preconditions are dynamic and prune nothing. `courier-terms`: `pick` has no room, 1 instance; `drop` keeps its room, 8; and $23 - 9$ leaves 14 for `move`. The lifted action has only 8 instances (one per destination), but its precondition contains the fluent as an argument of a static predicate, and the planner replaces it by each room it can stand for and keeps the connected pairs: the same 14 as before. The parameter was removed from the model, not from the planner's work; the real saving is in `pick`.

Solution 16.3.2. `courier-apart-domain.pddl`. `move` gains a universal precondition over the other robots:

```
                    ; no other robot is in ?to
                    (forall (?o - robot)
                      (or (= ?o ?r)
                          (not (= (location-of ?o) ?to)))))
```

In `courier-apart-p01.pddl` the plan has seven actions and all of them are done by `r2`: the robot that was in the way goes to the pharmacy and makes the delivery itself. The goal asked for the medicines in ward A, not for `r1` to take them. To see the rule at work, give `r2` something that keeps it busy, or ask for a second delivery.

Section 16.4 — No value: undefined

Solution 16.4.1. No. While the robot carries the sample, `(room-of sample1)` is undefined, and an undefined fluent is equal to no room: the validator reports `(room-of sample1) is undefined` for the plan

that stops before the drop. Only `drop` gives the fluent a value again. To ask that the robot ends with empty hands the goal needs something it can test: `(free r1)`, the predicate that the domain kept for exactly that.

Solution 16.4.2. With `holder`, where an item is and who carries it are one variable; nothing is left to the actions. `free` remains an invariant (or disappears, as in Exercise 16.2.2). With `room-of` and `carrying`, the actions must keep "`room-of` is undefined exactly when some robot carries the item": `pick` and `drop` each change both, and a forgotten `(assign (room-of ?i) undefined)` would leave the item in the room and in the gripper, the bug of Exercise 16.1.2 again. On Fidelity and Clarity the total fluent wins. The partial one has one advantage: its value type is plain `location`, with no supertype `place`.

Section 16.5 — Two more worlds

Solution 16.5.1. `blocks-of-p02.pddl`. The best plan has four actions:

```
(unstack d)
(stack c d)
(unstack b)
(stack a b)
```

In our run the first plan that Kairos printed had six: it parked a block on the other tower and had to undo it. Without a metric the search returns the first plan it reaches and then improves it; the anytime output ends with the four-action plan, which is optimal (each tower needs its top block on the table and its bottom block on top of it).

Solution 16.5.2. `coloring-australia.pddl`, with 18 `border` facts (each border in both directions). Kairos paints the seven regions in seven actions: South Australia gets a color of its own, and the ring around it alternates the other two. Tasmania has no border, so the precondition of `paint` is empty for it and any color will do; the plan paints it because the goal asks for every region to have one.

Section 16.6 — State variables: what the planner sees

Solution 16.6.1. `courier-p02`: 18 facts, 3 variables, 2 mutex groups. The groups are the eight `robot-at` facts, and the eight `item-at` facts together with `(carrying r1 meds1)`; the third variable is `(free r1)`. `courier-twin-p02`: 18 facts, 10 variables, 1 mutex group. The group that survives is the item's: `pick` and `drop` were not touched and still exchange one fact for another. The robot's group is lost, because `move` now adds a position without removing one, and the eight `robot-at` facts are eight independent variables: 1 + 8 + 1 = 10.

Section 16.7 — The MODEL method: one value or many

Solution 16.7.1. Floor of an elevator: object fluent (always one). People inside: predicate (many). State of a door: object fluent with three constant values. Rooms a door joins: predicate, static (two rooms; a relation). Battery level: numeric fluent. Cart a robot pushes: object fluent that may be undefined (at most one). Items on a cart: predicate (many); seen from the item, "the cart or room it is on" is an object fluent, like `holder`. Robot at the only charger: object fluent

of the charger that may be undefined, or a predicate `(charging ?r)` with the invariant "at most one robot".

Solution 16.7.2. `door-of-domain.pddl` and `door-of-p01.pddl`. The states are constants of a type `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))
```

The plan for one locked and one closed door has three actions: unlock, open, open. With the predicates `open` and `locked` a door could be open *and* locked, a state that the actions had to avoid and that an initial state could contain by mistake. With the fluent a door is in one of three states and the fourth combination does not exist. Adding a fourth state later (say `jammed`) is one more constant, not a new predicate to keep consistent with the other two.

Section 16.8 — PDDL 3 at a glance

Solution 16.8.1. (a) A preference in the precondition of `move`, priced in the metric: `:preferences`. (b) A trajectory constraint, `always`: `:constraints`. (c) An object fluent: `:object-fluents`. (d) Soft goals with `is-violated` in the metric: `:preferences`. (e) A soft constraint, `(preference on-time (within 600 ...))` in a temporal model with seconds: `:constraints` and `:preferences`.

C.17. Solutions for Chapter 17

The PDDL of these solutions is in `examples/ch17/solutions`. As in the chapter, the outputs of Kairos were obtained with one thread, and the note about the metric that comes with every PDDL+ plan is left out.

Section 17.2 — Processes: change while a condition holds

Solution 17.2.1. At 4 liters per second the level is $4t$. It reaches 9.5 liters at $t = 2.375$ s and the capacity, 10 liters, at $t = 2.5$ s, when the event `overflow` fires. The valve must be closed from 2.375 s on and before 2.5 s. Kairos decides at multiples of 0.1 s, and only one of them falls in that interval, so the plan must show 2.4 (with 9.6 liters in the tank). It does (`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
```

The window is 0.125 s wide, a little more than one step. With a goal of 9.7 liters it would be 0.075 s wide, from 2.425 to 2.5, with no step inside, and Kairos would find no plan.

Solution 17.2.2. The leak is a process whose condition is about the water, not about anything the agent does (`tank-leak-domain.pddl`):

```
(:process leaking
  :parameters ()
  :precondition (> (level) 0)
  :effect (decrease (level) (* #t (leak))))
```

While the valve is open two processes change the level and their rates add up: $2 - 0.5 = 1.5$ liters per second. Eight liters need $8/1.5 = 5.33$ s; the first step after that is 5.4, where the level is 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
```

The problem does not ask for the valve closed, so the plan has one action and ends when the goal holds. That is deliberate: with this domain, a goal of "8 liters and the valve closed" holds only at the instant of closing. After the valve is closed, `leaking` is the only active process and the level falls at 0.5 liters per second until the tank is empty. Then its own condition, `(> (level) 0)`, becomes false and the process stops by itself: a process needs no event to end. A goal that must *stay* true is not something a PDDL+ goal can ask for; the plan ends at `@PlanEND` and the goal is checked there.

Section 17.3 — Events: change at an instant

Solution 17.3.1. The files are `tank-wrong-loop-domain.pddl` (in the folder of the chapter) and `tank-wrong-loop-p01.pddl`. Kairos still returns the plan that closes the valve at 4 seconds: in that plan the level never reaches the capacity, the condition of the event is never true, and a wrong event that never fires does no harm. The plan that closes at 6 seconds is another matter:

```
$ 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
```

The student expected this plan to be valid now, since the goal no longer mentions the flood. But at 5 seconds the event fires, its condition is still true, and it must fire again, without end. The plan has no defined outcome. The student's reasoning ("the event is not needed for that problem") confuses two things: whether the *goal* cares about the event, and whether the *model* is well formed. An event has to switch itself off in every state the world can reach, not only in the plans we have in mind. The bug stays hidden as long as the planner avoids the event, and that is what makes it dangerous.

Solution 17.3.2. The switch is an event that deletes the fact that the process needs (`tank-switch-domain.pddl`):

```
(:event float-switch
  :parameters ()
  :precondition (and (valve-open)
                     (>= (level) (switch-level)))
```

```
    :effect (not (valve-open)))
```

Deleting `(valve-open)` makes the condition of the event false (it fires once) and stops `fill`. We also add `(< (level) (switch-level))` to the precondition of `open-valve`: otherwise the agent could reopen the valve above the switch level, and the event would close it in the same instant, which is legal but useless. The problem with a single action asks for 9 liters and the valve closed:

```
$ 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
```

The agent opens the valve and the world closes it. The problem `tank-switch-p02`, whose goal is `(flooded)`, shows that the tank cannot flood: Kairos finds no plan. The switch is the first example of something Chapter 18 develops: events as the automatic devices of a system, not only as its accidents.

Section 17.4 — Three kinds of change

Solution 17.4.1. The second bounce is at $t = 2.6$ s and leaves the ball with $+6.4$ m/s. At $t = 3.0, 0.4$ s later, the height is $6.4 \cdot 0.4 - 5 \cdot 0.4^2 = 1.76$ m and the velocity $6.4 - 4 = 2.4$ m/s. Both conjuncts of the window fail: the ball is 14 cm under the hand and too fast. The validator reports the first one it checks:

```
$ 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
```

The number 1.76 in the message is the value computed by replaying the processes and the two events up to that instant: the validator does the same replay that we did by hand.

Solution 17.4.2. Kairos answers `no plan found` for `ball-p02.pddl`. This time it is a fact about the ball. After the first bounce the ball leaves the floor at 8 m/s. At height h its speed is $\sqrt{8^2 - 2 \cdot 10 \cdot h}$, the same going up and coming down; between 1.9 and 2.1 m that is between 5.1 and 4.7 m/s, always above the 2 m/s that `catch` allows. The ball is slow only near the top of the arc, at 3.2 m, where the hand is not. No instant satisfies the precondition, on the grid of steps or off it. The exercise shows how to back a `no plan found`: the planner gives the evidence and the modeler supplies the argument.

Solution 17.4.3. *The cabin goes up*: a process, running while the cabin is `going-up` (its height increases at the speed of the cabin). *The door opens when the cabin reaches a floor*: an event; nobody decides it, and it happens at the instant the height reaches that of the floor. *The person presses a button*: an action. *The person walks in*: an action if the model treats it as instantaneous; if walking in takes time that matters, the start-process-stop pattern. *The door closes five seconds after opening*: a process (a timer that runs while the door is open) and an event (when the timer reaches 5); this pair is how PDDL+ writes "some time later". *The cabin is overloaded*: neither; it

is a condition of the state (the load is above the limit), which may appear in the precondition of an event (an alarm) or of an action.

Solution 17.4.4. At most 1 m/s: one second of acceleration reaches the limit and covers 0.5 m; braking takes another second and another 0.5 m; the 9 m left need 9 s at the limit. Eleven seconds, and the validator accepts the plan (`car-p02-hand.plan`):

```
$ kairos validate car-domain.pddl solutions/car-p02.pddl
    solutions/car-p02-hand.plan
plan valid, makespan 11
```

Kairos, with one thread and the time limit of the tests, returns the plan of 33.8 seconds that was its first plan for `p01`: 0.3 s of acceleration and half a minute at 0.3 m/s. That plan never comes near either limit, so it is valid for both problems. The comparison says what the note says: in PDDL+ the search of Kairos looks for few actions, both plans have five, and nothing pushes it towards the shorter one. Its plans are valid, not optimal. When the duration matters, read the plan, and if it dawdles, write a better one and validate it, or tighten the problem (a deadline, a limited amount of fuel) so that dawdling is not a plan.

Solution 17.4.5. Two processes burn the fuel and an event kills the engine (`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)))
```

The second process is not "the fuel burnt when accelerating" but the *extra* burnt when accelerating: the two run together and their rates add up to 3 ml per second. The event deletes `(running)`, which switches off the event, both burning processes and the process `moving`, and makes `park` impossible. With 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
```

It burns $3{\times}2.3{+}1{\times}4.4 = 11.3$ ml and covers $2.645{+}2.3{\times}2.1{+}2.645 = 10.12$ m. Without a speed limit, time is fuel, and the plan is faster than those of the chapter. With 3 ml (`car-fuel-p02.pddl`) there is no plan: the engine runs for at most 3 seconds, and no way of spending them covers ten meters and ends at rest.

Section 17.5 — The courier world in PDDL+

Solution 17.5.1. The plan lasts 38 seconds and the robot is on all the time, so the battery must last 38 seconds: the drain rate can be at most $30/38 = 0.79$ per second. (At exactly that rate the battery is empty at the instant of the `drop`, the event `shutdown` fires first, and the drop fails; the rate must be strictly lower.) The medicines are picked at $t = 10$ s, when the battery shows $30 - 0.5 \times 10 = 25$. The position of the robot, the load and the pick do not enter the computation: the process `drain` looks only at `(powered r1)`.

Solution 17.5.2. In `courier-battery-p03.pddl` the initial state has no `(powered r1)`. Every action of the robot needs it, so the plan must start with `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
```

Three actions share the time 0: they are applied in the order of the plan, with no time between them. The battery ends at $15 - 0.5 \times 28 = 1$. While the robot was off, before the plan started, it lost nothing; had the problem required waiting, staying off would have been the way to wait for free.

Solution 17.5.3. A robot whose battery ran out in a room, or that was switched off there, could still pick an item lying in that room: a dead robot that grabs things. It is a missing precondition, the classical kind of bug: an *action* is applicable when it should not be. No process runs for free. The two bugs are found by different tests: the missing precondition by a problem in which a switched-off robot is asked to do something, the free process by a problem that needs more time or energy than there is.

Solution 17.5.4. By number of actions: medicines first (10), sample first (11), the plan of Kairos (14). By duration: 57, 62 and 73 seconds. The first is invalid, so the choice is between the other two. A planner that minimized the number of actions would return "sample first". Kairos did not because its search is not optimal: it returns the first valid plan that its heuristics lead it to, and after parking the sample in the fridge the rest of the problem looks easy to them. Which plan the robot should execute depends on what the hospital cares about. If it is time, "sample first". If it is the sample, the plan of Kairos has a point: the sample spends 33 seconds out of a fridge in both, but in the plan of Kairos it is never more than 25 seconds away from one. For the planner to prefer one or the other, the preference must be in the model: a metric (`total-time`, or the final temperature) and a planner that optimizes it, or a harder limit, such as a lower `max-temperature`, that leaves only the plans we want.

Section 17.6 — How Kairos plans PDDL+

Solution 17.6.1. A classical problem has finitely many states. A search that visits all of them without reaching the goal has proved that no

plan exists, and Kairos says `unsolvable`. In `tank-p03` the decision is *when* to close the valve, and there are infinitely many instants. Kairos tries the instants on a grid of 0.1 s; having tried them all, it knows that no plan exists *on the grid*. A plan that closes the valve between two steps was never considered. So it reports what it knows: `no plan found`, with a note that calls it "evidence but not a proof". Here we can supply the proof ourselves: any level of 12 or more has passed through 10, where the event fires. The second answer is not a bug; it is a planner that does not claim more than it has shown.

Solution 17.6.2. It is valid (`tank-p01-offgrid.plan`): at 4.05 s the level is 8.1 liters, the goal holds, and the capacity is far.

```
$ kairos validate tank-domain.pddl tank-p01.pddl solutions/
    tank-p01-offgrid.plan
plan valid, makespan 4.05
```

Kairos could not have returned it: its plans only contain times that are sums of steps of 0.1 s. It does not matter here, because the set of valid closing times, from 4 to just before 5 seconds, is an interval one second wide and contains ten steps. It would matter if the interval were narrower than a step and fell between two of them. The validator has no grid: it accepts any time, and that makes it the tool for checking a plan that the planner cannot produce.

Section 17.7 — The MODEL method: who causes each change

Solution 17.7.1. The action table, with the cause of each change:

Name	Cause	When	What it changes
`switch-on`	agent	the machine is off	machine on, heater working
`brew`	agent	machine on, water at 90 or more	coffee made
`heat-water`	world, while	the heater is working	the temperature, at 5 degrees per second
`thermostat`	world, instant	heater working and water at 95	heater not working

The state needs two facts where the story seems to have one: `(machine-on)`, which the agent sets, and `(heating)`, which the thermostat clears. If the process depended on `(machine-on)`, the thermostat would have to switch the whole machine off. The domain is `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
```

From 20 to 90 degrees at 5 degrees per second: 14 seconds. The thermostat does not fire in this plan; it would at 15 seconds, and from then on the water would stay at 95, since the model has no cooling. A more faithful machine would lose heat (a second process) and switch the heater on again below some temperature (a second event), which is the thermostat of Chapter 18.

Solution 17.7.2. *Fidelity.* The process is faithful: the sample warms whenever it is out of a fridge. The effect on moves warms it only when the robot moves with it. A sample left on a bench, or held by a robot that stands still, stays cold for ever. The story that breaks it is `courier-cold-p01` itself: the sample waits on the bench while the robot delivers the medicines first. With the encoding by moves the plan "medicines first" is accepted, the sample arriving at $4 + 0.1 \times 25 = 6.5$ degrees in the model and spoiled in the world. *Size.* The moves encoding needs no process and no event, but one variant of `start-move` per sample carried, or a conditional effect. *Guidance.* The moves encoding is a numeric problem without time, far easier to search. *Clarity.* The process says in one place what happens to samples; the other encoding spreads it over every action during which time passes, and each new action that takes time must remember it. *Portability.* The moves encoding runs on any numeric planner; the process needs PDDL+. The decision: if samples only ever warm up in the hands of the robot (they start and end in fridges, and the robot never waits), the cheap encoding is acceptable and should say so in its Assumptions. As soon as something can warm up while nobody acts on it, only the process is right.

C.18. Solutions for Chapter 18

The PDDL of these solutions is in `examples/ch18/solutions`. The outputs of Kairos were obtained with one thread, and the note about the metric that comes with every PDDL+ plan is left out.

Section 18.1 — Rules that hold at all times: global constraints

Solution 18.1.1. *Fidelity.* With careful problems both encodings accept the same plans. The event depends on each problem repeating `(not (fined))` in its goal: a problem that asks only for `(parked)` at ten meters accepts the plan with the pedal down for 3.2 seconds, a speeding car that the modeler meant to forbid. The constraint has no such gap. On the other hand the event can describe a world in which the fine is a consequence and not a prohibition (an ambulance that may speed); the constraint cannot. *Size.* One ground event and one fact against one ground constraint: no difference. *Guidance.* With the constraint, a state that breaks the rule is discarded when it is generated. With the event, states with `(fined)` stay in the search, and everything reachable from them is explored although none of it leads to the goal. In our runs both versions returned the same plans; the difference grows with the problem. *Clarity.* The constraint says the rule in one place and under a name, and the validator reports that name and the instant (`constraint 'keep-the-limit' broken at 2.000001`); with the event the message is that the goal does not hold, and the reader must work out why. *Portability.* The event is PDDL+ as defined by Fox and Long; `:constraint` is an extension of some planners. Decision: the constraint when the rule is absolute, which is the case of a speed limit in a planning model; the event when breaking the rule is possible and changes what happens next.

Solution 18.1.2. The domain `tank-safe-domain.pddl` drops the predicate `flooded` and the event, and adds:

```
(:constraint never-flood
  :parameters ()
  :condition (<= (level) (capacity)))
```

The goal of the problem is now only "8 liters and the valve closed". Kairos returns the plan that closes at 4 seconds, and the validator rejects the late one by naming the rule:

```
$ 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
```

There is a small difference at the edge: the event fired when the level *reached* the capacity (`>=`), and the constraint allows a tank that is exactly full. More important is what the new model cannot say. It has no flooded tank at all: the problem whose goal was the flood (`tank-p02`) cannot be written, and nothing can be made to depend on a flood, such as an alarm or a cleaning task. A constraint removes a part of the world from the model; an event keeps it and labels it.

Section 18.2 — Change that is not linear

Solution 18.2.1. (a) Linear: the rate, 0.5, is a constant. (b) Polynomial, of degree two: the rate of the height is the velocity, which changes at the constant rate of gravity; between bounces the height is a parabola. (c) Exponential: with the heater off the only process is the loss, whose rate is proportional to the difference between the temperature and the outside, so the difference shrinks by the same fraction every second. (d) Exponential, for the same reason: the rate depends on the level that it reduces. (e) Linear, like (a). A check that does not need calculus: look at the expression multiplied by `#t`. If it contains no fluent that changes while the process runs, the change is linear. If it contains one that other effects change, polynomial. If it contains the fluent that the effect itself changes, exponential or stranger.

Solution 18.2.2. The model (`cup-domain.pddl`) has one process, whose rate depends on the temperature it lowers:

```
(:process cool-down
  :parameters ()
  :precondition (and (poured) (> (coffee-temp) (room-temp)))
  :effect (decrease (coffee-temp)
            (* #t (* (cooling) (- (coffee-temp) (room-temp)))))
   )
```

The action `drink` needs at least 55 degrees. Drinking too hot is not forbidden, it has a consequence: an event marks `(burnt-tongue)` if the coffee is above 65 degrees once drunk, and the goal refuses the mark. By hand: the difference with the room starts at 70 and shrinks as $70\,e^{-0.1t}$. It is 45 (the coffee at 65) when $t = 10\ln(70/45) = 4.42$ s, and 35 (the coffee at 55) when $t = 10\ln 2 = 6.93$ s. The window for drinking is 2.5 seconds wide, many steps.

```
$ 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
```

The plan drinks at 4.5 s, the first step after 4.42, with the coffee at 64.6 degrees. A staircase of steps of 0.1 s, each with the rate at its start, puts the coffee under 65 degrees already at 4.4 s, when it is in fact at 65.1: one more reason why the planner integrates with care and the validator has the last word.

Section 18.3 — Processes that interact

Solution 18.3.1. With the heater on, the temperature settles where the loss equals the heating: $0.1\,(T - 5) = 2$, so $T = 25$ degrees, in the middle of the growing band and far from the 30 of the constraint. The heater can now be left on. The shortest plan has two actions, `switch-on` and `harvest`. Its duration: the distance to 25 starts at 10 degrees and shrinks as $10\,e^{-0.1t}$; the band is entered at 22, when the distance is 3, at $t = 10\ln(10/3) = 12.04$ s. Twenty seconds of growth later it is 32.04 s, and the first step after that is 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
```

The weaker heater gives the longer plan and the simpler one: it takes seven seconds more and needs no switching. The pencil computation of where each quantity settles told us, before any search, which kind of plan to expect.

Solution 18.3.2. Free fall until 1.1 s: velocity $-10 \cdot 1.1 = -11$ m/s, altitude $20 - 5 \cdot 1.1^2 = 13.95$ m. Rotors on for 0.9 s, with a net acceleration of $+5$: velocity $-11 + 4.5 = -6.5$ m/s, altitude $13.95 - 11 \cdot 0.9 + 2.5 \cdot 0.9^2 = 6.08$ m, at 2.0 s. Free fall for 0.1 s: velocity -7.5 m/s, altitude $6.08 - 0.65 - 0.05 = 5.38$ m, at 2.1 s. From there, with the rotors on, the altitude is $5.38 - 7.5s + 2.5s^2$, which is zero at $s = 1.18$ s, with a velocity of $-7.5 + 5 \cdot 1.18 = -1.58$ m/s. Without rotors the drone falls for 2 seconds and arrives at -20 m/s. The condition of `crash` holds and that of `touch-down` does not, so `crash` fires: the drone is no longer airborne, no action is applicable, and `(landed)` can never become true. The planner does nothing "about" the event; it simply finds no plan along that path and must choose another. The crash is a dead end marked by the world.

Section 18.4 — Events that cascade, and events that never stop

Solution 18.4.1. `blinker-wrong`: `light-up` adds `(lit)`, which `go-dark` needs; `go-dark` deletes it, which `light-up` needs. Two arrows, one cycle, and nothing has to happen between the two events: the cycle closes in zero time. `blinker`: the same two arrows, but each event also sets the timer to zero and both need the timer at half a second. Between one event and the next the process `clock` must

run for 0.5 s. `greenhouse-auto` with 23 and 27: `too-warm` turns the heater off, one of the two things `too-cold` needs; the other is a temperature of 23 or less, and the temperature is at 27. The process `losing-heat` must bring it down, which takes about 2 seconds; the way back up, against the loss, takes about 4. With both thresholds at 25 the arrows are the same and the distance to travel is zero: after `too-warm`, the condition of `too-cold` is already true. The four graphs have the same shape. What tells the good models from the bad ones is not the graph but whether some *quantity* has to travel along each cycle.

Solution 18.4.2. Each domino gets a fact `(falling ?d)`, a timer `(tilt ?d)`, a process that advances the timer while the domino falls, and two events (`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))
```

When a domino lands, `hit` fires in the same instant and starts the fall of the next one: a cascade of two events, followed by half a second of process.

```
$ 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
```

Yes, the agent could now save the third domino. It starts to fall at 1 second, when the second lands. An action `remove` applicable to a domino that is neither falling nor fallen could be placed at any time before that, and `hit` would then find nothing to hit. In the instantaneous model no such moment existed. Whether an agent can intervene in a chain of consequences depends on whether the model lets time pass inside the chain.

Section 18.5 — Processes with durative actions and timed literals

Solution 18.5.1. The plan is `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
```

It is ten seconds shorter than the plan of Kairos (98 s). The robot plugs in with 6 units, charges for 46 seconds at a net 1.8 per second, and leaves with 88.8; it is at the door of the pharmacy at 60 seconds. The `pick` cannot be at 60.000: the timed literal makes `(is-open pharmacy)` true at that instant, and an action that needs a fact cannot share the instant with the happening that produces it. With the pick at 60.000 the validator answers `(is-open pharmacy) is false`. One epsilon later, 0.001 s by default, the fact is there. It is the rule of Chapter 10 about happenings that interfere, applied to a timed literal.

Solution 18.5.2. The pharmacy is 6 meters from the lobby, so the robot is there at 6 seconds, two before it closes: there is a plan. The problem (`courier-plus-p04.pddl`) has the fact in the initial state and a timed literal that deletes it:

```
   (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
```

The odd thing is the gap between 6 and 8 seconds. The robot picks the medicines and then stands in the pharmacy until it closes, for no reason: leaving at 6 gives a plan of 34 seconds. It is the same habit seen in the chapter, a wait that runs to the next thing that changes, here the timed literal. The two seconds cost 0.4 units of battery and nothing else, and the plan is valid. With the pharmacy closing at 5 seconds there would be no plan.

Section 18.6 — One system, three models

Solution 18.6.1. (a) Level 1. The first question is answered “no”: nothing in the story depends on time, and fuel is a budget that the trips spend. (b) Level 2. Time matters (two robots in parallel, an hour of drying), so the first answer is “yes”; but the drying can be attached to the painting, as a durative action or as a condition on the time since the first coat ended, and nothing changes outside the activities, so the second answer is “no”. (c) Level 3. The oven heats whether or not the cook is doing something, and for as long as it is on: the second answer is “yes”. If preparing the dough has a known duration it is a durative action, and the model is mixed. (d) Level 2, with timed initial literals for the visibility windows: the world changes by itself, but at times on the clock, which is what timed literals are for. The second question asks about change that depends on the state.

Solution 18.6.2. Level 2: the robot went to the dock first and arrived with $20 - 4 = 16$. The duration of `charge` is bounded by $(100 - 16)/2 = 42$ seconds, and the plan uses the bound. Level 3: the robot fetched the medicines first and arrived at the dock with 4. Plugged and on, the battery gains $2 - 0.2 = 1.8$ per second, and reaches 100 after $96/1.8 = 53.33$ s; the plan unplugs at the last step before that, 53.3. What the task needs is less. At level 3: 26 meters are left after the dock and 4 units are in the battery, so 22 more are needed, 12.2 seconds of charging. Neither planner looked there. At level 2 the engines of Kairos try a duration that is an inequality at its bounds, and the bound is a full battery. At level 3 the long wait stops when a numeric condition is about to change, and "the battery is enough for the rest of the plan" is not a condition of the model: no precondition, event or goal mentions it. The only condition in sight was that of `battery-full`. Both plans are valid. To get the shorter one from a planner, the pressure must be in the problem: a deadline for the delivery, as a timed literal.

Section 18.7 — Limits: steps and tolerances

Solution 18.7.1. At 5 liters per second the level is in the dose from $7.9/5 = 1.58$ s to $8.1/5 = 1.62$ s: a window of 40 ms, less than half a step. And Kairos finds a 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
```

The window is narrow, but a step falls inside it: at 1.6 s the level is exactly 8. The rule of the chapter, a window as wide as the change of one step, is what *guarantees* that some step lands in the window wherever the window is. A narrower window is a matter of luck: with 5 liters per second the steps happen to pass through 8, with 7 they pass through 7.7 and 8.4. A model should not depend on that luck; a small change in a rate, made months later by somebody else, would turn a solvable problem into `no plan found`.

Solution 18.7.2. In the search, the switch is examined at the steps. At 1.0 s the level is 7.0, under the target; at 1.1 s it is 7.7, over it, and the event closes the valve with 7.7 liters, outside the dose. In continuous time the switch closes when the level reaches 7.6, also outside the dose. The proposal fails twice: the search still sees no state with the right dose, and now the validator would reject the plan as well. Tuning a threshold so that the *stepped* world comes out right is aiming at the wrong world: the search is only where candidates come from, and the model must be right in continuous time, where plans are judged. The honest options are the ones of the chapter: a tolerance as wide as a step, or a slower inflow.

Section 18.8 — The MODEL method: testing and choosing hybrid models

Solution 18.8.1. *Trivial* (`drone-p04.pddl`): the drone starts 10 cm above the ground. Expected: no action at all; it falls for 0.14 s and touches

down at 1.4 m/s. Kairos returns a plan with no actions that ends at 0.2 s:

```
$ kairos drone-domain.pddl solutions/drone-p04.pddl
; cost = 0.2
; makespan = 0.200
0.200: @PlanEND
```

The empty plan is not "the goal is already true": the goal becomes true by waiting. It tests that `touch-down` fires. *No plan* (`drone-p03.pddl`): the battery is empty, `rotors-up` is never applicable, and the drone arrives at 20 m/s. Expected and obtained: `no plan found`. It tests `crash`. *Tempting* (`drone-p05.pddl`): the drone starts 5 m under the ceiling, going up at 4 m/s with the rotors on. A plan that begins by waiting breaks the constraint within a second, so the first action must come at once.

```
$ 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
```

The rotors go off at time zero in the best plan (at 0.2 s in the first one, which still stays under the ceiling). The set does not yet test `battery-empty`: a fourth problem with a battery for one second of thrust would.

Solution 18.8.2. (1) *Pencil.* Done in the text: the three sums of rates (−0.2, −1, +1.8), and from them the cost of the wait in `p03` (12 units for a battery of 10). (2) *Every event.* `shutdown` can never fire in a valid plan whose robot is needed, so it is tested by avoidance: `p03`, where the plan plugs in one step before it, and the problem without a plan below. `battery-full` does not fire in any plan printed in the chapter, because the plans unplug one step earlier: this is missing. A problem whose plan leaves the robot plugged until the battery is full would cover it. (3) *Horizon.* A problem without a plan was missing; `courier-plus-p05.pddl` has the battery at 3 and the dock at 4 meters, and ends with `no plan found` in half a second. (4) *Continuous validation.* The plan of Exercise 18.5.1, with times off the grid (60.001), is valid; the same plan with the pick at 60.000 is invalid. (5) *Replay and plot.* The figure of the battery for `p03`. (6) *Windows.* The only window that a plan must hit is the opening of the pharmacy, which lasts for ever in `p03` and 8 seconds in Exercise 18.5.2; the battery conditions are thresholds with wide margins. Nothing is narrower than a step. What the list shows is typical: the tests that come naturally exercise the plans we hope for, and the events are what remains untested.

C.19. Solutions for Chapter 19

Each solution is one reference model, in `examples/ch19/solutions`, with its test set in the `tests.txt` of the chapter. For every project we give the lines of the model card that took a decision, the patterns of the

two tables, the tests with what Kairos answered (one thread), and the alternative that step L should weigh. The domains are short and are not printed whole: read them next to these notes. A model that differs from ours and passes its own honest tests is a solution too.

Section 19.2 — The five checklists

Solution 19.2.1. *The library* (`library-domain.pddl`).

Card. Agents: one robot, so it needs no object. Scope, out: where exactly on a shelf a book goes; the order of the books in the basket. Assumptions: a book taken from the desk is not put back.

Tables. The corridor is a **static table** with **mirrored facts**, `(adjacent ?a ?b)`; the aisle of a book is another static table, `(belongs ?b ?p)`. The position of the robot is an **at-most-one invariant**. The basket is a **successor chain**: `(holding ?n)` with `(next ?n ?m)`, and the limit of three books is the end of the chain in the problem file. The desk is a constant, because the action `take` names it. `take` and `shelve` are **one-way actions**, which is what the assumption says.

Tests. Trivial (`p01`): 4 steps. Typical (`p02`), five books and three aisles: 18 steps in two trips, proved optimal after 928 states; the first trip takes the three books of the far aisles. No plan (`p03`): an aisle closed for works, with no `adjacent` fact, is `unsolvable` and the note names `(shelved saga)`. The cheats:

```
$ 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
```

A fourth book cannot even be written down: there is no fourth count. The book left in the wrong aisle is rejected because `(belongs saga history)` is false.

Step L. The rival of the chain is **slots**, three named places in the basket. Nothing in the story tells the places apart, so slots would only add symmetric plans, as they did for the tray of Chapter 7. A numeric fluent would also do and would cost the classical planners; the count is small and changes by one, the case that Chapter 8 leaves to the chain. The decision to watch is the map: with one corridor, `move` between neighbors keeps the model small, and a **merged action** that goes anywhere would hide the only cost that makes two trips different from five.

Solution 19.2.2. *The hotel floor* (`hotel-domain.pddl`).

Card. Scope, out: the route between rooms (`go` is a **merged action** from any place to any other; the cleaners know the way) and time. Assumptions: linen in the store room never runs out.

Tables. The vacuum cleaner is a **resource lock**: `(vacuum-at ?p)` or `(has-vacuum ?c)`, never both, and the goal asks for it back in the store room, as Chapter 7 advises for every lock. The signs and the departures are static tables. The goal is two rules about every room:

```
(:goal (and
  (forall (?r - room)
    (or (do-not-disturb ?r) (vacuumed ?r)))
  (forall (?r - room)
    (imply (checkout ?r) (fresh-linen ?r)))
  (vacuum-at store))))
```

Tests. Trivial: 5 steps. Typical (`p02`), four rooms: 14 steps, proved optimal. The cheat, one vacuum cleaner in two hands, is rejected with `(has-vacuum ben) is false`. The "no plan" test is the hole of the story: a guest who has left and whose door still shows the sign. The second rule asks for fresh linen and the sign forbids entering, so `p03` is `unsolvable` after one state. The model did its job here: it found a question for the owner of the problem. If the answer is "skip the room", the second rule gets the same exception as the first.
The second cleaner. In the best plan of `p02` one cleaner does everything and the other never leaves the store room. Plan length counts steps, and two people working at once do not make a plan shorter. The model is right for "what must be done"; the question "who does what, to finish early" needs the durative actions of Chapter 10, and that would be version 2 of this model.
Step L. The `forall` goal against a listed goal (Chapter 5): the list would hide the two rules and would have hidden the contradiction of `p03` with them. The `or` and the `imply` cost the planners without disjunctions; complementary predicates (`may-enter`) would buy them back.

Section 19.3 — Which feature for which need

Solution 19.3.1. *The water truck* (`watering-domain.pddl`).
Guide. Counted or measured: yes, liters, so numeric fluents. A better plan: yes, a sum of fixed costs of the moves, so action costs. Every later question: no. Time is out because nobody asked when; the card says so.
Card. The decision of the story: a bed gets all its water in one visit. Then `water` is one action with the precondition `(>= (tank) (need ?b))`, and a bed that needs more than a full tank can never be watered. That is our "no plan" test (`p03`, a lawn of 1200 liters), and a line of *Assumptions* that the town should read.
Tables. The tank is a **bounded quantity**: the precondition keeps it above zero and `refill` uses `assign`, so it cannot pass the capacity. The roads are mirrored facts with a distance each way; the meters go to `(total-cost)`.
Tests. Trivial: cost 800. Typical (`p02`): four beds that need 1900 liters. Kairos prints plans of 5000, 3600 and 2600 meters and proves the last one optimal: the truck waters the roses and the lawn, refills at the hydrant of the station, waters there and comes back through the park for the tulips. The first plan went back to the depot to refill twice. The cheat, 1100 liters from a tank of 1000, is rejected with `600 >= 700 is false`.
Step L. The alternative is a bed watered in parts, with `(need ?b)` as a dynamic fluent that each visit decreases. It loses no solutions and admits the large lawn; it needs an action for each amount, or an amount that the planner chooses, and it makes the best plan harder to find. With beds much smaller than the tank, as here, it buys nothing.

Solution 19.3.2. *The water network* (`network-domain.pddl`).
Guide. Is a fact a consequence of others? Yes: where the water reaches follows from the valves, through chains of pipes of any length. That is a recursive derived predicate (Chapter 13); no set of effects could keep it right, since closing one valve dries an unknown number of joints. Is there a rule about the whole plan? Yes, the clinic: a constraint (Chapter 15).

```
(:derived (supplied ?n - node)
  (or (source ?n)
      (exists (?p - pipe ?m - node)
        (and (joins ?p ?m ?n) (open ?p) (supplied ?m)))))
```

Card. The rule of the clinic belongs to the problem, not to the domain: another village has another clinic, or none. It is hard.
Tests. Trivial: close, repair, open. Typical (`p02`): a ring in which one valve, `p5`, starts closed.

```
$ 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)
```

The plan first opens the other side of the ring, so that the clinic keeps its water, and only then isolates the joint. The three-step repair that ignores the clinic is rejected with `the plan violates the problem :constraints`, and a repair with water in the joint with `(supplied j2) is true`. Without the fifth pipe (`p03`) the problem is `unsolvable`: the only way to dry the joint leaves the clinic dry.
Step L. The rule could be a precondition of `close-valve`, but which valves endanger the clinic depends on the state of all the others, so the precondition would need the derived predicate *after* the action: it cannot be written. This is a rule about states, and the constraint is its only direct encoding. On Portability the model pays twice: three planners of the eight read recursive derived predicates and two read constraints.

Solution 19.3.3. *The turnaround* (`turnaround-domain.pddl`).
Tables. Five durative actions with constant durations. The teams are **tokens**: `(cleaners-free)` and `(truck-free)`, deleted at start and added at end. The fuel rule is an **invariant**: `refuel` needs `(over all (cabin-empty ?p))`, `deboard` adds that fact at its end and `board` deletes it at its start, so refueling fits only between the two. Catering has no token, because the story gives every aircraft its own truck.
Tests. By hand, one aircraft: 10 minutes to get off, then the three services at once, of which refueling is the longest, then 15 to board: 45. Two aircraft: the truck refuels them one after the other, from 10 to 30 and from 30 to 50, and the second boards at 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]
```

The planner finds both makespans. The cheats are two plans by hand: boarding while the last fuel goes in (`over-all condition broken ... (cabin-empty ib31) is false`) and one truck at

two aircraft (`((truck-free) is false)`). The "no plan" test has no fuel truck, and is `unsolvable` with a note that names `(truck-free)`.
Step L. The fuel rule as an invariant against a **phase token** (deboarded, serviced, boarding). The phases are easier to read and forbid a plan that the story allows: boarding as soon as cleaning and catering end, when the fuel went in early. The model uses no negative preconditions on purpose: it runs on the temporal planners that lack them.

Solution 19.3.4. *The causeway* (`causeway-domain.pddl`).
Card. Assumptions, the schedule: the tide and the shift change at known times, whatever the driver does. They are timed initial literals of each problem. The decisions: a crossing needs low tide from its first minute to its last, `(over all (low-tide))` and also `at start`, since the interval of `over all` is open (Chapter 10); and nothing is done after the shift, `(over all (on-shift))` in both actions.
Tests. Trivial: there and back in the first tide, 55 minutes. Typical (`p02`), three parcels:

```
$ 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]
```

Forty-five minutes of work do not fit between minute 20 and the crossing back that should start by 40, so the van stays on the island and returns with the second tide. The cheat leaves at minute 50 and is rejected: `over-all condition broken after the happening at 60.000000: (low-tide) is false`.
For the "no plan" test we wrote two. A shift that ends at minute 30 (`p03`) is proved `unsolvable`: the note says that the timed literals take away what the goal needs before it can be reached, the proof of Chapter 13. A day with a single low tide (`p04`), in which the van can deliver and never come back, has no plan either, and Kairos cannot say so: with a limit of 3 seconds it answers `timeout`, exit code 11, and with 20 the same. Keep the first kind in the test set, and keep the second as a note in the card.
Step L. The end of the shift as a timed literal against `(within 480 ...)` on the goal (Chapter 15): the constraint is one line, but it lets the van cross after hours on the way to an earlier goal. The tide cannot be a constraint at all: it is a window that opens twice.

Solution 19.3.5. *The robot vacuum* (`vacuum-domain.pddl`).
Card. Required: the robot ends at its base. Wished, with weights 8, 5, 4 and 2: the kitchen, the living room, the nursery and the hall are clean. The weights only compete with one another, so their unit is free and the metric has no other term. Rules: "the hall never before the kitchen" looks at history and is `(sometime-before (cleaned hall) (cleaned kitchen))`; "the nursery once at most" is `(at-most-once (robot-at nursery))`. Both are hard, and both belong to the problem: another family has other rules, and the domain has none.
Tables. The battery is a bounded quantity with a precondition before every decrease. The domain is twenty lines: everything that makes this project interesting is in the problem file.
Tests. Typical (`p02`): 90 percent of work wished, 70 of battery. The

best plan found cleans the nursery, the kitchen and the hall, in that order, and gives up the living room: cost 5. By hand: the kitchen and the living room together need 60 and six moves, 72; no set with the living room and the kitchen fits, and no set without the kitchen is worth more. The cheats: the hall on the way in (`the plan violates the problem :constraints`) and a plan that needs 94 percent, rejected at a move with `0 >= 2 is false`. No plan (`p03`): the robot starts in the living room with 1 percent. Every wish can be given up and the way home cannot: `unsolvable`.

Step L. The rule of the hall as a precondition of `clean` with a static table `before`, as the courses of the waiter in Chapter 7: portable and just as exact, but it moves a rule of one family into the domain. The rule of the nursery would need the automaton of Chapter 15, a fact `left-nursery`. Constraints are the right place for both while the planner reads them, and two planners of the eight do.

Solution 19.3.6. *The canal lock* (`lock-domain.pddl`).

Who causes each change. The keeper opens paddles and gates and the boat sails: actions. The water rises while the upper paddles are open, whether the keeper watches or not: a process, and another for emptying. The water stops when the levels meet, at one instant: an event for each direction, which also closes the paddles in the model and sets the level to that of the reach. With that `assign` the level is an exact value again, the one safe way to compare it with equality (Chapter 18); the gates still test it with `>=` and `<=`.

Card. "The two gates are never open together" is written as preconditions of the two actions that open a gate, because only they can break it. A global constraint would say it once; we kept the model free of an extension that fewer planners read.

Tests. Pencil first: three meters at half a meter per minute are six minutes.

```
$ 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
```

The chamber that starts full is emptied and filled again: twelve minutes, twice the six of the trivial test. Both events were fired by a test, as Chapter 18 asks. The cheat opens the upper gate after four minutes and the validator, in continuous time, gives the level: `(>= (level) (upper-level)): 2 >= 3 is false`. The "no plan" test asks for both gates open and ends with `no plan found` after 1512 states: exit code 14, as every PDDL+ problem without a plan, and a short search because nothing in this model can keep changing for long.

Step L. Filling as a durative action of six minutes. By the guide of Chapter 18 it is enough here: the duration is known at the start and nothing interrupts it. The process earns its place as soon as the story grows: a chamber that starts half full, a keeper who closes the paddles half way, a leak. We chose the process because a lock is a system whose law is the water, and wrote in the card that the

durative version is the portable one.

Section 19.7 — Growing a model in versions

Solution 19.7.1. *The laundry, with coins* (`laundry-cost-domain.pddl`).
Card. Quality is new: among the plans that fold everything, the fewer coins the better. *Assumptions*: loading, unloading and folding are free. Nothing else moves.
Tables. Step O: a static function `(price ?m)` and the counter `(total-cost)`, which starts at zero. Step D: the cost column, zeros included: `wash` and `dry` add the price of their machine, the other three actions add nothing, on purpose. The domain is version 2 with one requirement, two functions and two effects more.
Tests. The tests of version 2 keep their verdicts. The typical one now has a cost that can be computed before running, two washes and two drying cycles: 10 coins.

```
$ kairos laundry-cost-domain.pddl laundry-cost-p01.pddl -v
...
kairos: solved (proven optimal), cost 10
...
```

With the second washer (`p02`) the best plan costs 9: one wash of the three loads in the large drum, 5 coins, and two drying cycles, since the dryer still holds 8 kg. That is the test with two plans of known cost that Chapter 9 asks for: the planner must prefer 5 coins to two washes of 3. The cheat saves the dryer and folds the whites wet; it is rejected as in version 1.
What next. The coins fit version 3 as a second term of the metric, a weighted sum of minutes and coins (Chapter 9), and the card must then say what a coin is worth in minutes. Expect the two measures to pull apart: the cheapest plan fills the drums and waits for them, the fastest runs more cycles. In version 4 the coins would be one more thing that a wish must be worth.

C.20. Solutions for Chapter 20

The exercises of this chapter are projects, and a project has more than one good solution. What follows are reference solutions: files of `examples/ch20/solutions` that pass their tests, and the decisions behind them. A solution of yours that takes other decisions is right if its tests pass and you can defend them on the five criteria.

Section 20.3 — Version 1: who carries what

Solution 20.3.1. The model card changes under *Assumptions*: "The ICU is restricted; a robot enters it only with a keycard; `r1` has one." In the state table that is two static rows with the names of Chapter 5, `(restricted ?l)` and `(has-keycard ?r)`, and in the action table one more condition of `move`. The reference domain (`courier-fleet-keycard-domain.pddl`) writes it next to the rule of the lab, in the same form:

```
    (robot-at ?r ?from) (connected ?from ?to)
    (or (not (restricted ?to)) (has-keycard ?r))
    (or (not (sterile ?to))
```

The key decision is that both facts are static tables in the problem and that nothing is said in the domain about which room or which robot: another floor, or a second keycard, is a line in a problem file. An action to hand the keycard over would be a different story, and the manager did not tell it.

Tests: the trivial test of the chapter still passes (ward A is not restricted). The typical test, `courier-fleet-keycard-p01`, is the round with `(restricted icu)` and `(has-keycard r1)`; Kairos returns 35, 33 and then 31 actions. The test without a plan, `p02`, takes the keycard away and is `unsolvable`, with the same note as for the closed door: `deliver` is never applicable. A cheat test is `p01` with the goal that `r2` ends in the ICU.

The best plan still has 31 actions and still uses one robot, but now it is `r1`: all 31 actions are its own. In a model that counts actions, the cheapest way to respect a rule about who may enter is to give everything to the one who may. The keycard changes *who*; only time, in version 2, makes two robots worth using.

Solution 20.3.2. The two encodings differ in a state where an unsealed sample is carried inside the lab without having come through the door carried. The problem `courier-fleet-leak-p01` builds it: the sample starts in the lab and must go to ward 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)
```

After the pick, the robot is in a sterile room carrying an unsealed sample: the constraint is violated, and since `seal` needs the sample to be carried, there is no way around it. The domain with the precondition does not notice, because the forbidden state was reached by `pick` and only `move` was guarded. It is the leak of Chapter 15, in our own model.

The repair, `courier-fleet-tight-domain.pddl`, gives `pick` the same rule: in a sterile room, a sample that is picked must be sealed.

```
                    (or (not (sterile ?l))
                        (forall (?s - sample)
                          (or (not (= ?i ?s)) (sealed ?s)))))
```

It needs `:equality`, because `?i` is an item and `sealed` speaks of samples. With it both domains answer `unsolvable` to the leak problem, and the round keeps its 31-step plan.

For the Fidelity row: "guards the door only" was a real difference, harmless for the Scope of the card (samples go to the lab) and wrong outside it. Whether to keep the simpler domain is a decision of step M: if the lab may also send samples out, the card must say so and the tight domain is the model. The exercise also shows the use of the constraint version: we found the leak by asking where the two encodings disagree, not by staring at one of them.

Section 20.4 — Version 2: time and battery

Solution 20.4.1. No change in the domain is needed because chargers are data: `has-charger` and `charger-free` take a location as a parameter, and `charge` works wherever both hold. The problem `courier-fleet-time-p05` adds one line to the round:

```
    (has-charger ward-b) (charger-free ward-b)
```

With three seconds, Kairos returns plans of 230.631, 177.224 and 137.619. The last one charges `r2` in the dock for 14 seconds, as before, and later charges `r1` in ward B for 19.6 seconds before the last delivery, which is the urgent box, at 137.

The plan is worse than the 113.017 of the chapter, although every plan of the old problem is still a plan of the new one. Nothing is wrong with the model. A second charger adds ground actions and, with them, many new ways of spending time that look useful to the heuristic (a charge is always applicable to a robot that is not full and stands at a charger); in the same three seconds the anytime search had not yet reached the plan that ignores the new charger. The answer to the question is no: more options are never worse in principle and are often worse in practice, within a time limit. That is the Guidance criterion, and it is a reason to compare the plans of a new version with those of the old one. When the new plan is worse and the old one is still valid, validate the old one and keep it:

```
$ 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
```

Solution 20.4.2. The domain `courier-fleet-time-blocks-domain.pddl` replaces the charge by a block: 5 seconds, 5 times the rate (25 units), applicable only while a whole block still fits in the battery, so that the capacity is never exceeded.

```
    :duration (= ?duration 5)
    ...
      (at start (<= (+ (battery ?r) (* 5 (charge-rate ?r)))
                    (battery-capacity ?r)))
    ...
      (at end (increase (battery ?r) (* 5 (charge-rate ?r))))
```

On the round it returns four plans in three seconds, the last with makespan 107.015: `r2` charges one block, from 30 to 55, and makes two deliveries with 46 meters; `r1` makes the other two and delivers the urgent box last, at 107.

The new column of the table: *Fidelity*, close to the chosen duration (charges of 5, 10, 15 seconds), but a robot above 75 cannot charge at all, which is a small lost solution; *Size*, 82 ground actions, as the full charge; *Guidance*, good, 107.015, the same makespan as the chosen duration with one charge action instead of four; *Clarity*, simple with one arbitrary constant; *Portability*, any temporal planner with numbers.

If a robot may leave the charger when it wants, we would choose the blocks, for this planner and this problem: they reach the plan of the chosen duration, with one dispatch to the charger instead of four, and they run on more planners. The block size decided two things: the precision of a charge (one block was enough here because $30 + 25 \geq 46$; with a block of 2 seconds the plan would need eight of them) and the dead zone near the top of the battery. It deserves the care of any other number in the model: a good choice

is the smallest charge that is worth a trip to the dock.

Section 20.5 — Version 3: a calendar and wishes

Solution 20.5.1. The action is `deliver` and the condition is `over all`: the receiver must be there during the two seconds, as for the `pick` of the chapter. In `courier-fleet-day-hours-domain.pddl`:

```
(over all (robot-at ?r ?l))
(over all (is-open ?l))
(at end (in-time ?i)))
```

The change has a consequence that is easy to forget and that the first run reveals: every destination must now be declared open in the problem. With only the lobby and ward B open, as in the files of the chapter, the urgent box cannot be delivered and the problem has no plan. The problem `courier-fleet-day-hours-p01` opens ward A, the ICU and the lab, and closes the lab at 90.

The plan of `r1`, with three seconds: cost 12, then cost 11. `r1` delivers the urgent box at 67 and goes straight home, where it arrives at 98. It loses the sample: it cannot be in the lab before $67+5+2+3+20 = 97$, and the lab closes at 90. Cost 11 is the routine box, the sample and the meal.

`r2` cannot recover the sample either, if it also takes the routine box: from ward A at 62 the sample is 24 meters away and the lab 20 more. It could do the sample *instead* of the box (after its charge: ward B at 40, pick, seal, lab at 65, delivered at 67), but the box is worth 5 and the sample 4, so the day that follows the weights forgoes the sample and the meal: total cost 6 for this order. The plan of Figure 20.4 does no better: its sample would be handed over from 97 to 99, with the reception closed since 90, so the new `deliver` rejects it; without the sample its cost would be $4 + 1 = 5$. One closing hour took the sample out of the round whoever carries it, and the weights say that it, and not the medicines, is the right loss.

Solution 20.5.2. The problem `courier-fleet-day-within` is the trivial test without `(at 80 (not (in-time meds1)))` and with two more lines:

```
(:requirements :constraints)
...
(:constraints (within 80 (delivered meds1))))
```

The fact `(in-time meds1)` stays in the initial state, because `deliver` still asks for it; it is now simply always true. Kairos returns the same plan as for `p01`, with makespan 67.005.

Of the timed literals of the round, the four deadlines can be written as `within`: the urgent one as a hard constraint, and the three at 100 too, but only *inside* the preferences, because a routine delivery is a wish: `(preference meds2-done (within 100 (delivered meds2)))` in the constraints section, which Chapter 15 showed. The two literals of the pharmacy cannot. Opening is a fact that becomes true at a time and that an action needs *throughout*; `within` speaks of the instant at which a condition is first reached, not of when an action may run. A window is a schedule of the world, and schedules are timed literals. This is the verdict of Chapter 13, found again on our own files.

Solution 20.5.3. The problem `courier-fleet-day-meal` is the day of `r2` (`p06`) with `(* 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` delivers the meal and goes home; it forgoes the routine box of medicines, cost 5. It cannot do both, as the chapter argued, and the meal is now the dearer of the two.

What was said after the card was that the weights are the manager's order turned into numbers. They no longer are: she said "medicines, then the sample, then the meal", and with the weights 5, 4 and 6 the meal comes first. The two sentences for her: *"With a 6 on the meal, on a morning when a robot can make only one more delivery, it will take the meal and leave the routine medicines for the next round. If that is not what you want, the meal must stay below 5, and we can tell the kitchen that it wins only against the sample."* The lesson of Chapter 14 holds at full size: a weight is not a volume control for one wish, it is a statement about all the others.

Section 20.6 — Version 4: the battery belongs to the world

Solution 20.6.1. The domain `courier-fleet-plus-sleep-domain.pddl` adds the two actions of Chapter 17, with one more condition: only an idle robot is switched off, so that no move is cut by its own 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))
```

Nothing else changes: the processes `drain` and `motor-drain` already have `(powered ?r)` as their condition, so a robot that is off does not drain. That is the advantage of having given each change its cause. The plan by hand, `courier-fleet-plus-p04-sleep.plan`, switches off at 0, on at 30.1, picks and delivers:

```
$ kairos validate courier-fleet-plus-sleep-domain.pddl courier-
    fleet-plus-p04.pddl courier-fleet-plus-p04-sleep.plan
plan valid, makespan 58.2
```

The robot keeps its 32 units through the wait and arrives with 4 (a little less: it is on for a tenth of a second before it picks). The planner, on the same files, returns plans of 114.9 and 94.7 that do not use the switch: they go to the dock and charge, as in the chapter.

It does not find our plan, which is 36 seconds shorter and has fewer actions (seven against ten). We can only guess why. From the

initial state, the way to our plan starts with `power-off`, an action that makes every other action of the robot inapplicable, which no heuristic likes; and the plan must then let exactly the right amount of time pass with nothing happening. To somebody who concludes that the switch is useless: the planner's answer shows that a plan without the switch exists, and nothing about the plans with it. The validator accepted a better one. A planner is one source of plans, and for a hybrid model Chapter 18 already told us to write the plan we expect and validate it. If the shorter plan matters, the model can help the search, for instance with a wish that robots be off while they wait, or by making the wait itself an action.

Section 20.7 — From plan to execution

Solution 20.7.1. The decisions, in order.

What about `r2`? At 30 it is starting to pick the urgent box. Disturbing it would risk the first hard goal of the round. We let it keep its plan and plan for `r1` alone: the decomposition of the chapter, now forced by execution.

The state of `r1`. It is in ward B, free and idle. It has driven 4 + 22 = 26 meters: battery 74.

The calendar. Time 0 of the new problem is 30 of the round. The pharmacy has opened: that timed literal is in the past and becomes the plain fact `(is-open pharmacy)`. It closes in 60, and the round ends in 70.

The goals. The new request is urgent, so it is a hard goal with the end of the round as its deadline. The routine box was a wish and stays one, with its weight; so does the way home. The planner, not we, should decide whether the box still fits.

```
    (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]
```

The sample is in the lab 27 seconds after the request, at 57 of the

round. The routine box is given up: from the lab the pharmacy is 24 meters away, the robot would pick at 53, in time, and reach ward A at 81 of the new clock, eleven seconds after the end of the round. So `r1` goes home instead, with $74 - 42 = 32$ of battery, and the cost is 5. The new request cost the round its routine medicines, and the manager should hear it in those words.

Conflicts with `r2`. Add 30 to every time and lay the plan on Figure 20.4. `r1` delivers in the lab from 55 to 57; `r2` arrives there at 97. Both use the corridor, which our model allows. The only shared resource of the domain is the charger, and neither plan uses it after 30. To validate the two plans together one needs a joint problem for the instant 30, and that problem must describe `r2` in the middle of its pick; the simplest honest way is to start the joint problem at 32, when the pick has ended, with the positions and batteries that the two plans predict.

Could `r2` have helped? After the urgent box it takes the first sample from ward B and is in the lab at 99; it has no second to spare, and the routine box is lost whoever is asked. One thing the solution takes for granted, and a real system must not: planning took time. If the planner needs three seconds, the problem should be written for 33 and not for 30; our plan has more than forty seconds of slack on its hard goal and survives.

Section 20.10 — The MODEL method: one story, four models

Solution 20.10.1. *Questions for the author.* Can a forklift carry two pallets? (No.) Can a pallet be put down anywhere else than in its place? (No: the same decision as `deliver`.) Is "in front of each rack" a place for one forklift? (Yes.) May two forklifts pass each other between the aisle and a rack? (No: that is what narrow means.) Do the forklifts have to end somewhere? (Not said: no goal about it.) Battery? (They are plugged in by day; out of scope.)

Model card. Story: two forklifts store the pallets that wait at the gate, each in its rack. Agents: the forklifts, at the same time. Success: every pallet stored. Scope, in: the places and their links, which places are narrow, where forklifts and pallets are; out: battery, the height of the racks, people. Assumptions: one pallet per forklift; a narrow place holds one forklift; the gate holds any number; in the temporal version, 8 seconds per drive and 5 per load or store.

Classical version (`warehouse-domain.pddl`). The new idea is the row "a narrow place with nobody in it", `(clear ?l)`: a fact about absence, kept by add and delete, the resource lock of Chapter 7 with one lock per place. Entering a wide place and entering a narrow one are two actions, `drive` and `squeeze`, which keeps the domain in `:strips`; one action with a conditional effect would be the other reasonable encoding.

```
  ; 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)))
```

Tests. Typical (`p01`): two pallets, one per rack; eight actions, and the plan sends one forklift all the way to its rack before the other enters the aisle. Cheat (`p02`): `f1` in front of rack A and `f2` in the aisle must swap places. The tempting plan is two actions, each driving through the other; the model returns six, in which `f2` backs out to the gate, `f1` comes out behind it, and they re-enter in the other order. No plan (`p03`): the same swap with a third forklift in front of rack B and the gate cut off; `unsolvable`, and the note explains that no narrow place can ever become clear. A trivial test is one pallet and one forklift.

What the classical version cannot say: how long the night's work takes, and what "fits" means during a drive. In a sequence, a forklift is in one place and then in the next. In time it is between them for 8 seconds.

Temporal version (`warehouse-time-domain.pddl`). The decision is in `squeeze`, and it is the question of Chapter 10 about where an effect goes: the place entered is taken *at the start* of the drive, and the place left is given back only *at the end*. During the drive the forklift holds both.

```
  ; 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))))
```

With the two effects the other way round (free the old place at start, take the new one at end), two forklifts could start into each other's place at the same instant, and a third could enter a place that somebody is about to reach. The round of two pallets takes 42.005: both load at once at the gate, `f2` goes in, and `f1` enters the aisle at 21.003, one happening after `f2` has left it for its rack. Compare with the classical plan: the same eight actions, and now the order between the two forklifts is forced by the aisle and visible in the times.

Step L. The alternative to one lock per place is a single lock for the whole narrow zone (one forklift beyond the gate at a time). It is smaller and has no deadlocks to search through, and it loses the plans in which one forklift works at rack A while the other is at rack B: with two racks that is the whole benefit of the second forklift. We keep the lock per place. The next versions would follow the chapter: a calendar if the trucks leave at fixed times, wishes if not every pallet must be stored tonight.

Bibliography

References are listed in alphabetical order.

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* (tech. rep. No. 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* (tech. rep. No. 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* (tech. rep. No. CVC TR-98-003/DCS TR-1165). Yale Center for Computational Vision and 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.

Alphabetical index

abstraction, 139
action, 4, 22
  applicable, 23
  applying, 23
  in PDDL+, 368
action costs, 188
action granularity, 150
action performer, 475
action table, 12, 26
add effect, 48
ADL, 121, 127
anti-pattern, 150
  catalog, 424
  forgotten delete, 150
  goal in the initial state, 151
  idle parameter, 152
  out-of-step twins, 151
anytime search, 58, 193
applicable action, 48
at-most-one invariant, 143
axiom, 285

basic predicate, 282
behavior tree, 475
blind search, 55
blocks world, 33, 103
  above, 287
  with object fluents, 353
bouncing ball, 369
breadth-first search, 55
briefcase domain, 120
budget, 202
busy predicate, 222

capacity, 145, 170
car domain, 371
case study, 449
charging, 244
cheat test, 376
checklist, 416
chests puzzle, 102
circuit domain, 287
clarity, 14
classical planning, 5
  assumptions, 5
closed world assumption, 22, 48
coffee machine domain, 247
combinatorial explosion, 54
comparison, 164
compilation, 106
  of conditional effects, 127
complexity, 55
concurrency, 220
conditional effect, 114, 115
  compilation, 127
  semantics, 116
  universal, 119
conditions, 96
constants, 74, 84
constraint
  on the plan, 327
continuous change, 256, 258
continuous effect, 256, 259
  planning, 270
cooking domain, 225
coordination, 236
counting, 176
counting without numbers, 144
courier world, 3, 25
  battery, 271
  cart, 114, 121
  cold chain, 377
  courier-fleet, 452
  courier-fleet-day, 461
  courier-fleet-plus, 468
  courier-fleet-time, 457
  fleet, 449
  in PDDL+, 373
  sterile laboratory, 124

dead end, 63, 148
deadline, 293
  within, 299
debugging, 152
  extended procedure, 426
decision guide, 418
delete effect, 48
delete relaxation, 56
derived predicate, 282, 283
  planning, 290
  recursive, 285
  restrictions, 284
diagnostics, 490
discrete change, 256, 258
disjunctive precondition, 100
dispatching, 472
  event-triggered, 472
  time-triggered, 472

documentation, 439
domain, 8
door, 238
drone, 393
duration
  chosen, 244
  computed, 218
durative action, 211
dynamic fact, 26, 37

effect, 4, 23, 114
  add, 23
  delete, 23
either, 83
elevator domain, 226
envelope action, 238
epsilon separation, 215, 216
equality, 74, 86
event, 363
  cascade, 395
  infinite firing, 397
  must make its condition false, 367
existential precondition, 101
expansion, 55

fact, 22, 48
Fast Downward, 57
FF, 57
fidelity, 13
finite-domain representation, 355
finite-domain variable, 52
forall
  with imply, 104
forgotten delete, 150
forward search, 55
frame rule, 37
function, 163

generator domain, 265
global constraint, 387
goal, 4, 24, 49
  negative, 99
  quantified, 105
goal in the initial state, 151
granularity, 150
greedy best-first search, 56
greenhouse domain, 262
gripper, 35
ground action, 23, 48
grounding, 50

of quantifiers, 106
size, 62
guidance, 14

happening, 215
hard limit, 202
Helmert, Malte, 179
heuristic, 56
blind spots, 58
HSP, 57
hybrid system, 363
modeling, 387

idle parameter, 152
initial state, 4, 24
International Planning Competition, 10
IPC-5, 320
invariant, 143, 217, 348
inverse pair, 148
IPC, *see* International Planning Competition

joint action, 242

Kairos, 10, 486
–debug, 492
–dump, 51, 492
–no-anytime, 60
–target-cost, 60
–timeout, 60
-v, 51, 492
--no-anytime, 195
--target-cost, 195
check, 24, 29, 490
commands, 487
engines, 240, 488
exit codes, 426, 489
installation, 486
options, 488
PDDL+, 379
Python, 492
solve, 487
validate, 29, 491
kairos check, 77
kitchen domain, 191

LAMA, 57
laundry domain, 429
Lights Out, 119
linear change, 258
linear temporal logic, 336
lock predicate, 249
logistics domain, 81

makespan, 215
map coloring, 354
match cellar, 236
merged action, 150
metric, 188
over fluents, 196
weighted, 199
mirrored facts, 143
model, 3
versions, 429, 451
model card, 12
model card, 450
template, 439
MODEL method, 12, 37
case study, 478
checklists, 416
complete, 414
conditions, 106
consolidated for classical planning, 154
constraints, 340
continuous change, 275
coordination, 249
criteria, 13
derived predicates, 297
five steps, 414
hybrid systems, 407
object fluents, 356
plan quality, 203
preferences, 320
processes and events, 381
quantities, 180
step D, 12, 129
step E, 13, 29, 65, 152, 426
step L, 13, 65
step M, 12, 449
step O, 12
time, 228
timed initial literals, 297
types, 88
versions and documentation, 442
model-based autonomy, 3
modeling, 138, 414
modeling levels, 402
monitor, 339
mutex group, 52, 144, 355
mutual exclusion, 147, 222

negative goal, 99
negative precondition, 96
net benefit, 312
no plan found, 427
non-linear change, 389
numeric condition, 164
numeric effect, 165
numeric fluent, 162
numeric planning, 162

object, 21
object fluent, 348
as a term, 351
one-way action, 148
open list, 55
optimal plan, 193
optimal planning, 58
out-of-step twins, 151
oven domain, 267
over all
numeric conditions, 260, 262
over-subscription, 306

parallel actions, 220
parameter, 23, 62
partial observability, 477
pattern, 138
at-most-one invariant, 143
catalog, 421
for control, 145
for the actions, 148
for the state, 141
inverse pair, 148
merged action, 150
mirrored facts, 143
one-way action, 148
phase token, 145
resource lock, 147
slots, 145
static table, 142
successor chain, 144
turn token, 146
PDDL, 9
:action, 22
:action-costs, 189
:adl, 127
:condition, 212
:conditional-effects, 115
:constants, 84
:constraint, 387
:constraints, 329
in the domain, 336
:continuous-effects, 258
:derived, 283
:derived-predicates, 283
:disjunctive-preconditions, 100
:duration, 212
:duration-inequalities, 244
:durative-action, 212
:durative-actions, 212

:effect, 23
:equality, 86
:event, 367
:existential-preconditions, 101
:fluents, 163
:functions, 163
  object-valued, 349
:goal, 24
:init, 24
:metric, 189, 219
:negative-preconditions, 96
:numeric-fluents, 163
:object-fluents, 349
:objects, 24
:parameters, 23
:precondition, 23
:predicates, 21
:preferences, 307
:process, 364
:quantified-preconditions, 103
:requirements, 23
:strips, 23
:time, 364
:timed-initial-fluents, 296
:timed-initial-literals, 291
:types, 75
:typing, 75
:universal-preconditions, 103
=, 86
?duration, 212, 245
#t, 258
1.2, 495
2.1, 260, 497
2.2, 282, 498
3.0, 500
3.1, 500
always, 329
always-within, 332
assign, 165
  object fluent, 349
at end, 217, 330
at start, 217
at-most-once, 330
choosing a feature, 418
decrease, 165
define, 21
either, 83
exists, 101
forall, 103
  with preferences, 313
forall (in effects), 118
  hold-after, 332
  hold-during, 332
  imply, 100
  increase, 165
  is-violated, 307
  maximize, 198
  not, 96
  object, 79
  or, 100
  over all, 217
  preference, 307
  quick reference, 494
  requirements, 494
  scale-down, 175
  scale-up, 175
  sometime, 330
  sometime-after, 331
  sometime-before, 331
  syntax, 495
  total-cost, 189
  total-time, 219
  undefined, 352
  version 3.0, 320, 357
  version 3.1, 356, 357
  versions, 9
  when, 115
  within, 332
PDDL+, 363, 501
  case study, 468
  with durative actions, 399
Pednault, Edwin, 121
phase token, 145
plan, 2, 25, 49
  format, 489
  sequential, 211
  temporal, 215
  validation, 11, 491
plan execution, 472
plan length, 188
plan quality, 188
planner, 3, 47
  PDDL support, 503
planning, 2
  applications, 7
  definition, 2
  versus learning, 6
  versus programming, 6
PlanSys2, 474
plateau, 57
portability, 14
  by feature, 420
precondition, 4, 23, 48, 96
  versus condition of an effect, 117
predicate, 21
preference, 306
  and action costs, 312
  compilation, 319
  in preconditions, 314
  on the final state, 310
  over a constraint, 337
  under forall, 313
  violated, 308
  weight, 309
    choosing, 311
  without metric, 322
problem, 8, 24
process, 363
  interaction, 391
PSPACE, 55

quantity, 162

rate, 258
reachability, 285
reachable state, 50
recursive rule, 286
redundant fact, 131, 151
relaxation, 56
relaxed plan, 56
replanning, 473
required concurrency, 236
requirements, 10
resource, 222
resource lock, 147
road trip domain, 200
ROS 2, 474
rover domain, 337
rule (derived predicate), 283

s-expression, 20
SAS+, 356
satellite domain, 295, 317
satisficing planning, 58
search, 55
semantics, 47
sensing, 477
Shakey, 8
side effect, 124
simple temporal network, 248
size (criterion), 14
sliding puzzle, 53
slots, 145
soft constraint, 337
soft goal, 306
solutions to the exercises, 505
speed, 218
start-process-stop pattern, 369
state, 4, 22, 48
  as a set of facts, 22
state table, 12

state space, 50
    infinite, 178
    size, 54
state table, 26
state variable, 348, 355
static fact, 26, 37, 52, 63
static function, 170
static table, 142
steps of time, 379
    limits, 404
STN, 248
stratification, 290
stratified rules, 290
STRIPS, 8, 20
subtype, 78
successor chain, 144
successor state, 48
supertype, 78
Sussman anomaly, 35
symmetric relation, 143
symmetry, 64
task planning, 7
temporal planning, 211
temporally expressive, 240
temporally simple, 240
test set, 13, 29
thermostat, 396
time window, 293
timed initial fluent, 296
timed initial literal, 282, 291
timeline, 212
timeout, 60
token, 145
tourist domain, 316
towers of Hanoi, 53
trajectory, 329
trajectory constraint, 327
    compilation, 339
    semantics, 334
truck domain, 172
turn taking, 146
turn token, 146
type hierarchy, 78
type predicate, 26, 74
types, 74, 75

uncertainty, 477
undecidability, 178
undefined, 352
undefined value, 177
universal effect, 114, 118
universal precondition, 103
unsolvable problem, 12, 31, 60

validation
    in continuous time, 379

waiter domain, 138
water jugs, 173
weighted sum, 199
workshop domain, 338

Zeno behavior, 397
zero-cost action, 193