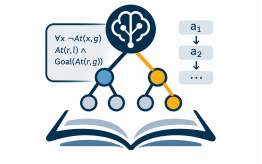
Classical planning: PDDL 1.2

Planning and PDDL, Part II (Chapters 4–7)

Francisco Martín Rico

Intelligent Robotics Lab, URJC



Outline

1. Types, constants and equality
2. Richer conditions
3. Richer effects
4. Modeling classical problems: patterns and choices
5. Summary

1 Types, constants and equality

2 Richer conditions

3 Richer effects

4 Modeling classical problems: patterns and choices

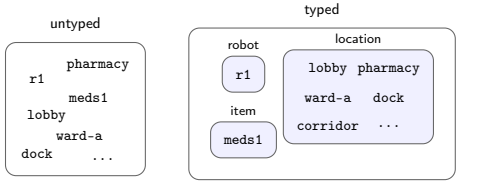
5 Summary

What type predicates cost

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

Type predicates (Chapter 2): correct, any planner. Three costs:

- **Noise**: a third of each precondition, 10 facts of `courier-p01`
- **They check nothing**: `(item-at pharmacy meds1)` passes `check`; the planner says "unsolvable"
- **They multiply candidates**: 10^3 per action, 3000 in total; 30 applicable



With **types**, each parameter draws only from the box of its type.

Types: the typed courier domain

```
(:requirements :strips :typing)

(:types robot location item)

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

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

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

A **type** is a name for a set of objects; requirement `:typing`

- Section `(:types robot location item)`
- **Typed variables**: `?r - robot`; a type applies to all the untyped variables before it: `?from ?to - location`
- No type predicates, no type facts

Same 7-step plan for `courier-p01`: types change what the *modeler* can write, not what the robot can do.

What the checker catches, what the planner gains

```
$ 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]
...
invalid
```

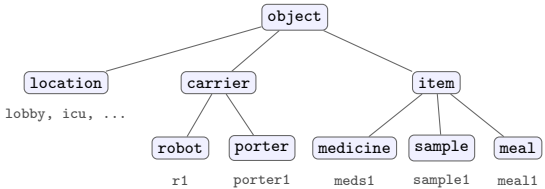
Action	Untyped	Typed
`move`	$10^3 = 1000$	$1 \times 8 \times 8 = 64$
`pick`	$10^3 = 1000$	$1 \times 1 \times 8 = 8$
`drop`	$10^3 = 1000$	$1 \times 1 \times 8 = 8$
Total	3000	80

- File, line, column, and a guess: "arguments 1 and 2 seem to be swapped"
- Candidates: the product of type sizes, not a power of the objects
- Both models: the same 30 ground actions. The gain is in grounding, not search
- `courier-p02` (2 robots, 3 items): 6591 against 224

Type hierarchies

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

- **Subtype** and **supertype**, with the hyphen
- An object of a type is also an object of *every supertype*
- `?c - carrier`: robots and porters; `?p - porter`: porters only
- Root: the predefined type `object`; untyped means `object`



Rule of thumb

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

The team: one action, two kinds of carrier

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

- Best plan, 13 steps: the same `move` and `pick` with `r1` and with `porter1`
- A plan where the robot serves: `kairos validate` says "argument 1 must be porter, 'r1' is robot"

Step 0: kinds before properties

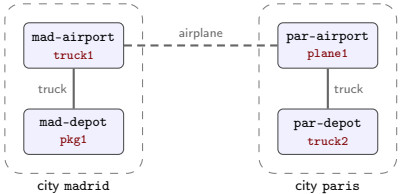
Kind	Is a	Why a kind of its own
`location`		things are *in* it
`carrier`		moves, picks, drops
`robot`, `porter`	`carrier`	only a porter serves
`item`		picked and dropped
`meal`	`item`	only a meal is served

Test for a type: *some predicate or action treats these objects differently.* Otherwise, or if it changes while the plan runs, it is a property.

A second domain: logistics

```
(:types
  city
  airport depot - place
  truck airplane - vehicle
  package vehicle - physobj)
;; A truck drives between any two places of one city.
(:action drive
  :parameters (?t - truck ?from ?to - place ?c - city)
  :precondition (and (at ?t ?from)
                     (in-city ?from ?c) (in-city ?to ?c))
  :effect (and (not (at ?t ?from)) (at ?t ?to)))

;; An airplane flies only between airports.
(:action fly
  :parameters (?a - airplane ?from ?to - airport)
  :precondition (at ?a ?from)
  :effect (and (not (at ?a ?from)) (at ?a ?to))))
```



- `truck` → `vehicle` → `physobj` → `object`
- One `load` for trucks and airplanes
- "Airplanes land only at airports": in the parameter list, no precondition
- 12 steps; a plan flying to `par-depot` is rejected by `validate`

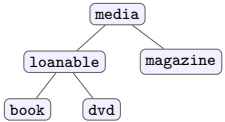
A parameter of several types: `either`

```
(: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)))
```

- `(either t1 t2 ...)`: the union of several types; part of `:typing`
- A borrowed magazine: `check` says "must be (either book dvd), 'mag1' is magazine"



the union has a name

- A repeated union is a concept with no name yet
- The hierarchy reads better and is more portable (LAMA rejects `either` in parameters, ENHSP at all)
- Keep `either` for a union that *crosses* the hierarchy

Constants

```
(:constants pharmacy - location)
;; 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))))
```

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

- A **constant**: an object declared in the domain, in every problem
- `:constants`, no requirement; usable like a variable
- Problems use it but *do not declare it*
- Plan steps carry only what was chosen

The test

Does a rule of the domain mention this object by name? The lobby is in every problem, but no rule names it: not a constant.

Equality: a robot is not its own witness

```
$ kairos
    courier-check-noeq-domain.
    pddl courier-check-p01.pddl
; cost = 9
...
(pick r1 meds1 pharmacy)
(verify r1 r1 meds1 pharmacy)
...
```

- Two parameters can always be bound to the same object if its type fits both
- `(= ?a ?b)`: requirement `:equality`; almost always negated

```
;; 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)))
```

Now p01 (one robot) is unsolvable; in p02 the best plan has `(verify r1 r2 meds1 corridor)`

Step E: the cheat test

For two parameters of overlapping types: one object of that type, a goal that needs two. If a plan comes back, read the step where the same name appears twice.

The MODEL method: kinds of things

Step	New questions
O	**Which kinds of things?** A type if some predicate or action treats them differently and no object changes kind
O	**A hierarchy?** Shared predicates: supertype; actions for a part: subtype
O	**In every problem?** Named by a rule: a constant
D	**May two parameters be the same object?** If not, and nothing prevents it: an inequality

Step E: `kairos check` every file; one wrong plan per subtype rule, rejected by `kairos validate`

Type predicates or types (`courier-p02`)

	Type predicates	Types
Fidelity	swaps accepted	swaps are errors
Size	6591 candidates	224 candidates
	124 ground actions	124 ground actions
Guidance	same search	same search
Portability	every planner	every classical one

Verdict: use types; keep type predicates for a kind that changes or crosses the tree.

A constant or a parameter (the pharmacy rule)
Constant: the object is part of the *rules*. Parameter and static predicate `dispensary`: part of the *situation*. Same 21-step plans, 64 ground actions. When in doubt, the parameter.

Common mistakes with types

A type that swallows the variable before it

`(?r ?i - item ?l - location)`: `?r` is an item. `check` reports "argument 1 of 'robot-at' must be robot, '?r' is item". Look at the hyphens of the declaration.

Types without `:typing`

Kairos warns (with `check` or `-v`); others stop at the first hyphen. The same for `:equality`.

An object declared with the supertype

`meds1 meds2 - item`: rejected where a `medicine` is expected, or silently never chosen. Use the most specific type.

A constant declared again in the problem

"'pharmacy' is already a domain constant": a warning in Kairos; a stricter planner refuses the problem.

Different variables, different objects?

Legal PDDL, no message: only the cheat test finds it. Fix: `(not (= ?r ?w))`.

1 Types, constants and equality

2 Richer conditions

3 Richer effects

4 Modeling classical problems: patterns and choices

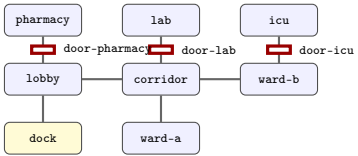
5 Summary

Conditions about absence: negative preconditions

```
;; 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)))

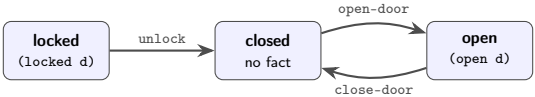
;; A closed door: one that is not open and not locked.
(:action open-door
  :parameters (?r - robot ?d - door ?a ?b - location)
  :precondition (and (robot-at ?r ?a) (joins ?d ?a ?b)
                     (not (open ?d)) (not (locked ?d)))
  :effect (open ?d))
```

- `:negative-preconditions`: `(not (p ...))` holds when the fact is *absent*
- `not` in an effect deletes; in a precondition it tests



Passages without a door: `connected`, crossed by `move`. With a door: `joins`, crossed by `move-through`.

Three states with two predicates; negative goals



- No `closed` predicate: closed is neither open nor locked
- **Closed world assumption**: what is not listed is false
- All doors closed: no door fact in the initial state

Negative goal (`courier-doors-p02`):

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

- One step more: `close-door` after leaving the pharmacy
- Often true from the start: all-negative goals may give the empty plan
- POPF and OPTIC accept no negative conditions

Exercising the doors: the cheat test earns its place

Step E on the doors: four problems

Trivial: p01, one closed door. *Typical*: p03, ICU door locked, robot with a keycard (12 steps). *No plan*: p04, without the keycard. *Cheat*: p04 on every earlier version of the domain.

v0: `open-door` with only `(not (open ?d))`

```
$ kairos courier-doors-v0-domain.pddl
    courier-doors-p04.pddl
; cost = 11
...
(open-door r1 door-icu ward-b icu)
...
```

- A locked door is not open, so the robot opened it
- Two predicates, four combinations: *open and locked* became reachable
- When absence defines a state, check which other states have the same absence

Alternatives: or and imply

```
(: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)))
```

Or, as the rule reads in the story:

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

- `:disjunctive-preconditions`
- `(or a b ...)`: some part holds
- `(imply a b)`: same as `(or (not a) b)`
- `courier-access-p01`: only `r2` has a keycard and makes the whole 9-step trip
- Nobody with a keycard (`p02`): the note has two lines about `move`, one per alternative

Some object: exists

```
(: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)))
```

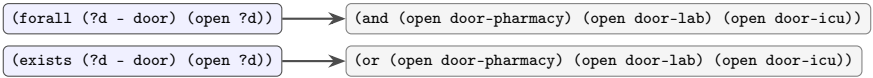
```
$ kairos courier-escort-domain.pddl
    courier-escort-p01.pddl
; cost = 8
(move r2 ward-a corridor)
...
(move r1 ward-b icu)
```

- `(exists (?x - t) c)`: `c` holds for some object of type `t`; `:existential-preconditions`
- The plan does not name the escort

Keys and chests

"You have some key that fits" is an `exists`. If a key breaks when used, the effect must delete it: the key becomes a parameter, `(open oak iron)`.

Every object: `forall`



- `:universal-preconditions`; both: `:quantified-preconditions`
- Abbreviates over the objects of the problem
- Empty type: `forall` true, `exists` false
- `not exists` = `forall not`

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

```
;; ?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)))
```

Same 6-step plan as the classic domain. The inequality was hidden there: a held block was not `clear`.

One move for all passages; quantified goals

```
(: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)))
```

`(forall (?x - t) (imply a b))`: every `x` that satisfies `a` also satisfies `b`. No door on the passage: every implication is true.

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

Independent of the objects; best plan 22 steps

Cheat test p02:

```
(forall (?i - item) (not (item-at ?i corridor)))
```

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

- An item in the gripper is in no room
- p03 adds `(forall (?r - robot) (free ?r))`: pick, move, drop
- A goal about absence must say where things may end up

What a planner does with rich conditions

- **Quantifiers are expanded** over the objects; nested ones multiply
- **Static parts are evaluated**: `(joins ?d ?from ?to)` leaves one door condition
- **Disjunctions split actions**, one per alternative; an `exists` splits into one per object
- **Negation**: handled directly, or by a second opposite fact

Encoding	Ground actions	States
`or` (access)	64	352
two actions (split)	52	352
`exists` (escort)	98	
escort as parameter	74	

The 12 extra are moves of `r2` where both parts of the `or` hold; duplicates lead to the same successor.

Warning sign

A `forall` whose body has an `or`, `imply` or `exists` over facts that change: 2^{10} for ten such conditions.

The MODEL method: absence, any, every

Step	New questions
O	**Is the property yes or no, or does it have several values?** Store one side, the one true less often; for three values or more, check that meaningless combinations cannot be reached
D	**Is the condition about absence?** `not`
D	**Is it about any object?** `exists` if the action does nothing to it; a parameter if it does, or if the plan should name it
D	**Is it about every object?** `forall`, with `imply` to select
D	**Are there several ways to satisfy it?** `or`, or one action per way

Step E: two more cheat tests

- For every state defined by absence, look for another with the same absence (the locked door that was "not open")
- For every goal about absence or "every", ask how it could be true for a silly reason (the item in the gripper; a type with no objects)

Alternatives compared

Choice	Measured	Verdict
`not` or a complementary `closed`	Same plans; 21 vs 24 facts, 42 ground actions and 102 states each	`not` for a yes/no property: nothing to keep in step. `closed` for POPF and OPTIC. A forgotten fact: a closed door, or a door that is nothing
`or` or two actions	64 vs 52 ground actions (12 duplicates); 352 states each	`or` when the alternatives are a detail of one rule with the same effects; two actions when they will differ, or the plan should tell them apart
`forall` goal or a list	46 vs 45 facts; 23679 states each	`forall` for a rule ("everything delivered"), problems by hand; the list for generated problems or planners without quantifiers (LPG-td, Metric-FF)

- Same plans in every pair: the choice is about Fidelity, Clarity and Portability

Complementary predicates

Out of step if one effect forgets one: Chapter 7 names it *out-of-step twins*.

Common mistakes with conditions

`and` under a `forall` where `imply` was meant

```
(forall (?d - door)
  (and (joins ?d ?from ?to) (open ?d)))
```

Every door must join these rooms: no robot ever moves, and Kairos says only "every state reachable ... was searched (1 states)". Under `forall` select with `imply`; under `exists`, with `and`.

A `forall` true of nothing

Medicines declared `- item`: the goal `(forall (?m - medicine) ...)` holds at the start, `; cost = 0`. Count the objects of the type.

A quantified variable without a type

`(forall (?m) (item-at ?m icu))` asks rooms and robots to be in the ICU: no plan. Always type it.

Requirements forgotten; `not` in the initial state

Each construct has its flag. A negative initial fact adds nothing: what is not listed is false.

1 Types, constants and equality

2 Richer conditions

3 Richer effects

4 Modeling classical problems: patterns and choices

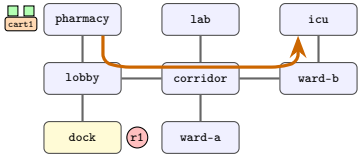
5 Summary

Effects that depend on the situation: the cart

```
; Exactly ?i and ?j on the cart.
(:action push-cart-2
  :parameters (?r - robot ?c - cart ?from ?to - location
               ?i ?j - item)
  :precondition (and (pushing ?r ?c) (robot-at ?r ?from)
                     (connected ?from ?to)
                     (on-cart ?i ?c) (on-cart ?j ?c)
                     (not (= ?i ?j))
                     (forall (?k - item)
                       (imply (on-cart ?k ?c)
                              (or (= ?k ?i) (= ?k ?j)))))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (cart-at ?c ?from)) (cart-at ?c ?to)
               (not (item-at ?i ?from)) (item-at ?i ?to)
               (not (item-at ?j ?from)) (item-at ?j ?to)))
```

One action per number of items, `push-cart-0` to `push-cart-3`:

- **A limit nobody asked for**: four items, the cart is stuck
- **Large**: $14\,n^k$ candidates; every pair counted twice
- **Repetitive**: a new rule, four places



- An item on the cart keeps its `item-at`: the push must update it
- What we want: *the robot and the cart change room, and so does every item that is on the cart*

Conditional effects: `when`

```
(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))))

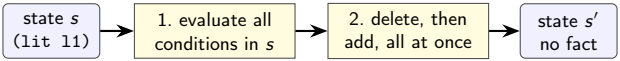
  ; The master switch: every lamp goes off, whatever it was.
  (:action all-off
    :parameters ()
    :effect (forall (?l - lamp) (not (lit ?l)))))
```

- `toggle` has no precondition: always applicable
- `lights-p01`: `(toggle l1)`, `(toggle l2)`: the same action lights one lamp and puts out the other. Read a plan with the state in mind

`(when <condition> <effect>)`: if the condition holds, the effect happens too; requirement `:conditional-effects`

1. The conditions **do not decide** whether the action applies: only the precondition does
2. Every condition is evaluated in the **state before** the action
3. Effects that fire apply **at once**: deletes, then adds

A condition in the effect or in the precondition



`(when (lit l1) ...)` fires `delete (lit l1)`
`(when (not (lit l1)) ...)` does not `add nothing`

	In the precondition	In a `when`
It decides	whether the action applies	which facts change
When false	the action cannot be used	used, without that effect
Evaluated in	the state before	the state before

- Ask the world: when it is false, can the agent still do it?
- Lit lamp: still pressable (effect). Closed door: no walking through (precondition)
- A button on three lamps: 2^3 STRIPS actions, one with `when`

Universal effects: `forall`

```
; The master switch: every lamp goes off, whatever it was.
(:action all-off
  :parameters ()
  :effect (forall (?l - lamp) (not (lit ?l)))))
```

- `(forall (<vars>) <effect>)`: once for every object
- The mirror of the universal precondition: that one *asks*, this one *does*
- No new requirement: part of `:conditional-effects`
- Expanded at grounding; deleting an absent fact changes nothing

```
$ kairos lights-domain.pddl
    lights-p02.pddl
; cost = 6
...
; better plan 2
; cost = 2
(all-off)
(toggle l2)
```

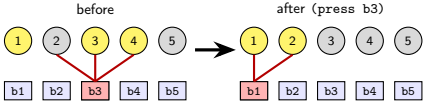
A blunt tool: *every* lamp. Most of the time the effect is for *some* objects, and the `forall` carries a `when` inside.

Both together: Lights Out

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

For every `?x` **such that**: the `forall` brings every object in, the `when` keeps those concerned. Unwired lamps are left alone (the frame rule).

```
$ kairos lightsout-domain.pddl
    lightsout-p01.pddl
...
; cost = 2
(press b3)
(press b1)
```



- `press`: one parameter, any number of lamps
- p02: unsolvable after 16 states of 32; the answer is a proof

The cart, in one sentence

```
; The robot, the cart and every item on the cart change room.
(: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 (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 `when` contains a delete *and* an add
- `pick`:
  `(forall (?c - cart) (not (on-cart ?i ?c)))`
- `move`:
  `(not (exists (?c - cart) (pushing ?r ?c)))`

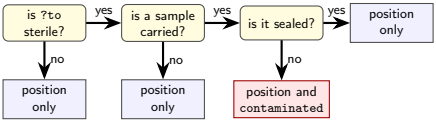
```
$ kairos courier-cart-domain.
    pddl courier-cart-p01.pddl
...
; better plan 3
; cost = 11
...
(push-cart r1 cart1 ward-b icu)
```

- The goal becomes true by a push whose parameters name no box
- p02: the robot unloads before the cart returns

Effects to avoid: the sterile laboratory

```
; The move is always possible. What it changes, besides the
; position, depends on the room entered and on what is carried.
(: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 (?i - sample)
                 (when (and (sterile ?to) (carrying ?r ?i)
                            (not (sealed ?i)))
                   (contaminated ?i)))))
; Put the carried sample in a sealed bag.
(:action seal
  :parameters (?r - robot ?i - sample)
  :precondition (and (carrying ?r ?i) (not (sealed ?i)))
  :effect (sealed ?i)))
```

- The airlock stops nobody: nothing in the precondition
- `forall`: `move` has no item parameter; the robot may carry nothing



- Three conditions, all read before the move
- An airlock that *refuses*: a precondition, another world; a question for step M

Exercising a conditional effect

Step E: a test set for a conditional effect

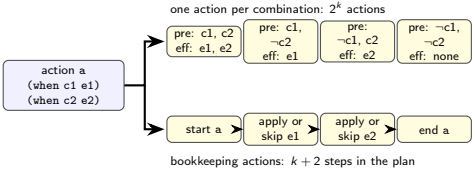
Trivial	medicines to the lab: the condition never holds (7 steps)
Typical	sample to the lab, goal `(not (contaminated sample1))`: seal first (8 steps)
No plan	open *and* usable: unsolvable after 71 states
Cheat	only the sample in the lab

```
$ kairos courier-sterile-domain.pddl
    courier-sterile-p04.pddl
; cost = 7
...
(move r1 ward-a corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

One step shorter, the sample arrives contaminated. `kairos validate` accepts this plan for p04 and rejects it for p02: "(contaminated sample1) is true".

A conditional effect can be a danger: the planner avoids it only if a goal or a later precondition needs the opposite. Every harm that matters must be in the goal.

What planners do with conditional effects



No third way avoids both costs (Nebel, 2000): conditional effects are a real gain of expressive power.

- **Universal effects are expanded**; static conditions disappear
- **Conditional effects kept, or compiled away**: the model grows, or the plans do
- **Heuristic**: each conditional effect as a small action (FF tradition)
- Kairos keeps them: 3 lamps, 4 ground actions; 6 in the STRIPS model
- Valid plans only from Kairos, ENHSP and LAMA; not from OPTIC, POPF, TFD, LPG-td, Metric-FF

The MODEL method: effects that depend on the situation

Step D: two new questions

1. **Does what it changes depend on the situation?** An "if" that does not stop the agent: `when`
2. **Many objects at once?** "Every", "whatever is on": `forall`

Action table: *always changes* and *also changes, when*. Step E: condition holds, does not, and a cheat test.

p02	A. Conditional effect	B. Action per case	C. Bookkeeping
Fidelity	exact	4 items: stuck	needs `(not (unsettled))` in goals
Size	109 ground actions	343, grows as n^k	189
Guidance	25 steps, with the cart	25 steps	best plan (28) avoids the cart

D. Do not store what can be deduced: `load` deletes `item-at`; a plain STRIPS push; 133 ground actions, 13 state variables (37 for A). But `p01` needs 16 steps: the boxes must be dropped.

Verdict: A when the outcome depends on the state; B for few fixed cases; C only without support; first ask whether the fact must be stored (D).

Common mistakes with effects

Reading the effects as a program

```
:effect (and (when (lit ?l) (not (lit ?l)))
             (lit ?l))))
```

Deleted and added: adds win, the lamp stays lit (unsolvable, 1 state).

The forgotten `when`

A push without `(on-cart ?i ?c)`: 7 steps, nobody loads anything, every item of the hospital follows the cart.

A `when` in a precondition

Read as an undeclared predicate, "never true". Use `imply`.

Requirement left out; `when` inside `when`

Declare `:conditional-effects`. `(when a (when b e))` is `(when (and a b) e)`.

A harm that the goal does not watch

The contaminated sample, the boxes left behind: right models, incomplete goals.

1 Types, constants and equality

2 Richer conditions

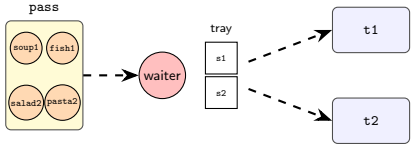
3 Richer effects

4 Modeling classical problems: patterns and choices

5 Summary

A problem from a story: the waiter

The cook leaves the finished dishes on the pass. The waiter takes them to the tables on a tray with two places. At each table the courses come in order. The evening has gone well when every dish that was ordered is on its table.



Step M: the model card

Agents	one waiter; cook and guests are not agents
Success	every ordered dish on its table
Scope	*in*: position, tray, served, course order. *Out*: cooking, the route, time, eating, payment
Assumptions	a served dish stays served; a dish does not go back to the pass

- No route: an **abstraction**, right if navigation solves it
- "Does not go back" looks harmless: it will create dead ends

Steps O and D: state table and domain

Property	Kind	Pattern
`(waiter-at ?p)`	dynamic	at-most-one
`(ready ?d)`	dynamic	
`(on-tray ?d ?s)`	dynamic	slots
`(empty ?s)`	dynamic	slots
`(served ?d)`	dynamic	
`(ordered ?d ?t)`	static	static table
`(before ?a ?b)`	static	static table

- One waiter: no object, a one-argument predicate
- pass: a constant
- Course order is *data*: a dessert needs no change in the domain

```
; 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))))
```

A constant and an inequality (Ch. 4); `forall` with `imply` (Ch. 5)

Step E: exercise the model

Trivial one dish: load, move, serve

Typical four dishes: 16, 13, then 11 steps, optimal

No plan the soup is not `ready`, the goal asks for the fish: unsolvable

Cheat only the main courses: 11 steps, all four served

Two halves of a cheat test

The planner does not take the shortcut, and the shortcut written by hand is rejected, for the reason we expect.

```
$ 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)
...
```

The cheat plan, validated: "step 4 (serve fish1 s1 t1) ... precondition false: for ?e = soup1: (imply (before ?e ?d) (served ?e))"

Eleven patterns: the catalog

A **pattern**: a named answer to a question of step O or D, with a **rival** and a **cost**.

	Pattern	Use it when	It costs
State	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
Control	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
Actions	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

Patterns for the state: static table, mirrored facts

Static table: a relation no action changes

```
; 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)))
```

- Wolf, goat, cabbage: who eats whom is `(eats ?a ?b)`, data
- 7 crossings; one more fact `(eats wolf cabbage)`: no plan
- If another instance means editing an action, the knowledge is misplaced
- Free at search time; a missing fact is a different world

Mirrored facts: a symmetric relation, both directions

```
(connected lobby dock) (connected dock lobby)
```

Rival: 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)))
```

- 7 facts instead of 14; same 30 ground actions
- The `or` loses one-way doors. Mirrored facts are plain STRIPS: the default

Patterns for the state: at-most-one invariant

```
(: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)))
```

- An **invariant** holds in every reachable state; PDDL cannot declare it
- **At-most-one**: every action that adds a value deletes the old one; exactly one in the initial state

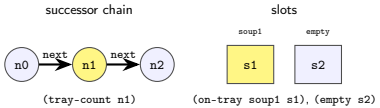
- For "*the* position", "*its* state", "*the* holder"
- A **mutex group**: at most one of its facts is true; the planner finds them
- Eight `robot-at` facts become one variable: 18 facts, 3 variables
- Free, and fragile: one forgotten delete breaks it

```
$ 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
```

Counting without numbers: successor chain or slots

```
; One more dish: the count goes one step up the chain. With
; the tray full there is no next count, and no load.
(:action load
  :parameters (?d - dish ?n ?m - count)
  :precondition (and (waiter-at pass) (ready ?d)
                     (tray-count ?n) (next ?n ?m))
  :effect (and (not (ready ?d)) (on-tray ?d)
               (not (tray-count ?n)) (tray-count ?m)))
```

- **Successor chain**: numbers as objects, `(next ?n ?m)` static, one fact holds the count
- The capacity is the length of the chain in the problem: the limit is the *absence* of data
- Only $+1$ and -1 for free; beyond a few tens, numbers (Ch. 8)



- **Slots**: one object per unit of capacity, and a fact that says it is free
- For distinct places: a rack, a parking lot, two hands
- Cost: symmetry. Waiter p02: 220 states with slots, 123 with the chain

Chain or slots

Count when only the amount matters; slots when the places differ.

Patterns for control: phase token

```
(: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
...
(advance cutting joining)
(join seat)
...
```

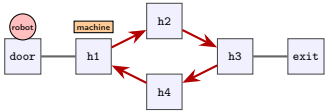
- A **token**: a fact that says who or what has permission now
- **Phase token**: each action requires its phase; `advance` moves the token forward, never back
- For stages that must not mix: all cutting before any gluing
- Preconditions order the work *of one part*; the token, *of the workshop*
- Cost: an action per phase change; dead ends. Gain: a smaller search

Patterns for control: turn token

```
; 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))))
```



- **Turn token**: actions need their turn and pass it on
- 9 steps: `wait`, then follow the machine one cell behind
- A world step that cannot happen means "forbidden"
- Cost: plans twice as long

Resource lock and merged action

Resource lock: a fact says that the resource is available

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

- Cheat test, the card in two hands: `(has-keycard r1)` and `(has-keycard r2)`: unsolvable
- Never released unless needed: put it in the goal

Merged action: steps with no decision in between become one

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

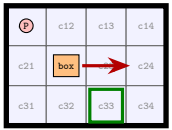
- The controller batches the tray: 4 ground actions instead of 22, a 4-step plan
- **Granularity**: model at the level of the decisions the planner must take

Patterns for the actions: inverse pair, one-way action

- **Inverse pair**: each restores what the other changed (pick and drop)
- **One-way action**: permanent (`seal`, `serve`, `advance`)
- Follow the world: a missing inverse creates **dead ends** the world does not have; an invented one lets the planner escape final mistakes

Waiter p05, tray full of main courses: no plan. The inverse of `load`:

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



- Sokoban: `walk` is its own inverse, `push` has none. Two pushes east: the box is on the wall for ever
- Inverses: no false dead ends, a larger space. Delete-relaxation heuristics miss most dead ends

Anti-patterns: valid files, wrong models

Anti-pattern: a mistake common enough to deserve a name. `kairos check` is satisfied: only the symptom gives it away.

Anti-pattern	Symptom
Forgotten delete	a plan that is too short
Goal in the initial state	an empty plan, cost 0
Out-of-step twins	a small test passes, a larger one has no plan
Idle parameter	too many ground actions

Goal in the initial state: `waiter-done-p01` gives `; cost = 0`. Worse: a goal *partly* true at the start.

Forgotten delete

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

```
$ kairos courier-nodelete-domain.pddl
    courier-nodelete-p03.pddl
; cost = 6
...
(pick r1 meds1 pharmacy)
(move r1 lobby corridor)
...
```

6 steps, not 10: it leaves *from the lobby*. The [sas] line: 10 variables, not 3.

Anti-patterns: out-of-step twins, idle parameter

Out-of-step twins

```
:effect (and (not (on-tray ?d ?s))
             (served ?d))))
```

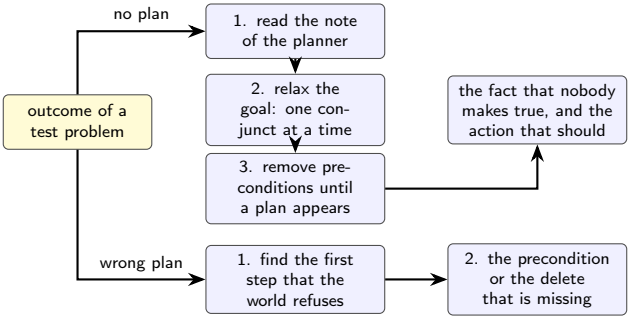
- Twins: `empty` and `on-tray`, `free` and `carrying`; legitimate
- `serve` forgets `(empty ?s)`: after two dishes the tray holds nothing and has no free place
- Symptom: the trivial test passes, the typical has no plan
- Prevention: write the invariant of each pair; check every action

Idle parameter

```
(:action load
  :parameters (?d - dish ?s - slot ?t - place)
  :precondition (and (waiter-at pass) (ready ?d) (empty ?s))
```

- `?t`: no precondition constrains it, no effect uses it
- `(load fish1 s2 pass)`, `(load fish1 s2 t1)`: the same
- 38 ground actions instead of 22
- Test each parameter: constrained, and used in an effect? (`?t` of `serve` is not idle)

Debugging a model: the procedure



No plan: the smallest goal that fails, then the precondition whose removal brings a plan back, then: which action should make that fact true?

Wrong plan: a **missing precondition**, or an **earlier action left a wrong fact**. Then a regression test: the shortcut as a plan file that `validate` rejects.

Debugging the out-of-step twins

2. Relax the goal: `p06`, only `(served soup1)`: 3 steps. `p07`, the two starters: 6 steps. Then:

```
$ 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)
```

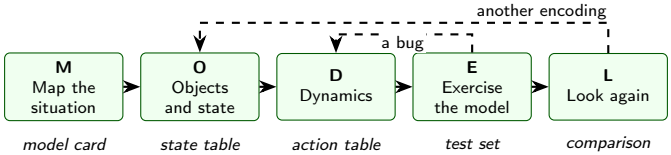
Never *the third* dish: look at the tray.

3. Remove preconditions: `load` without `(empty ?s)`

```
$ kairos
    waiter-bug-relaxed-domain.
    pddl waiter-bug-relaxed-p02
    .pddl
; cost = 10
(load soup1 s1)
(load salad2 s1)
...
```

- *Who adds `(empty ?s)`?* None: `serve` lost a line

The classical MODEL checklist



- **M** Who acts? What is solved? What is left out, and who handles it? Which assumptions make it classical?
- **O** Kinds, hierarchy, constants; static or dynamic; absence, any, every; *one value at a time? an amount? deducible from another?*
- **D** When it applies, what it changes; the same object twice; depends on the situation; *can it be undone? one decision or several? every parameter needed?*
- **E** Estimate the size; trivial, typical, no-plan, cheat; validate; on failure, the procedure
- **L** Every pattern has a rival: compare on Fidelity, Size, Guidance, Clarity, Portability

Step L for the waiter

p02	Slots	Successor chain	Merged action
Fidelity	exact; knows which dish is where	exact for the story	capacity and trips not represented
Size	22 ground actions	22 ground actions	4 ground actions
Guidance	220 states; every plan has a twin	123 states; no symmetry	9 states
Clarity	`(load soup1 s2)`	`(load soup1 n0 n1)`	shortest domain and plans
Portability	universal precondition	the same	the same

Verdict: a question of step M, *who decides the trips?*

- The planner: the **successor chain**; interchangeable places, half the states
- Places that differ: **slots**
- The robot's controller batches dishes: the **merged action**

The method is a loop

The model written first was not the best for the story as told: the normal outcome of step L.

Common mistakes: a pattern with a piece missing

Half of the mirrored facts

Only `(connected lobby dock)`: the note walks back from the goal to a `move` that "is never applicable". One missing direction: a robot that enters a room and cannot leave.

Two tokens

`(now cutting)`, `(now joining)`, `(now painting)`: 9 steps, no advance; the phases order nothing. Exactly one token in `:init`.

A chain with a broken link

Only `(next n0 n1)`: the tray holds one dish, the plan grows from 11 to 15 steps. Test a problem that needs the full capacity.

A lock that nobody returns

`r1` ends with the keycard on board. If the next shift expects it on its hook: `(keycard-at lobby)` in the goal.

1 Types, constants and equality

2 Richer conditions

3 Richer effects

4 Modeling classical problems: patterns and choices

5 Summary

What to take away

- **Types** turn silent mistakes into errors; a hierarchy states a rule once; **constants** for the objects the rules name; **equality**: the planner never assumes that two parameters differ
- **Conditions**: `not`, `or`, `imply`, `exists`, `forall`; distrust goals true because something is missing
- **Effects**: `when` and `forall`; every harm that matters goes in the goal
- **Patterns** and **anti-patterns**: names for the choices of every classical model; debugging is a procedure

`:adl` stands for	Chapter
`:strips`	2
`:typing`, `:equality`	4
`:negative-preconditions`	5
`:disjunctive-preconditions`	5
`:quantified-preconditions`	5
`:conditional-effects`	6

List what the file really uses, not `:adl`.

Classical planning ends here

ADL is a better language for the same problems. Numbers, time and preferences change the problem itself.

Exercises to try

- **4.3.2**: the zoo; draw the hierarchy, validate a plan where a keeper feeds the lion
- **4.6.1**: `swap` with and without an inequality; count ground actions
- **5.1.3**: `lock` a door; which negative preconditions?
- **5.5.1**: `close-floor` over the ward domain, every robot in the dock
- **6.4.3**: two robots and one cart; find the flaw and repair it
- **6.5.2**: the airlock that refuses; compare the test sets
- **7.5.1**: three anti-patterns in `gripper-buggy-domain.pddl`
- **7.7.1**: the whole method on a ferry for two cars, five artifacts