Time: PDDL 2.1 and 2.2

Planning and PDDL, Part IV (Chapters 10–13)

Francisco Martín Rico

Intelligent Robotics Lab, URJC



Outline

1. Durative actions
2. Coordination in time
3. Continuous change
4. Derived predicates and a world with a calendar
5. Summary

1 Durative actions

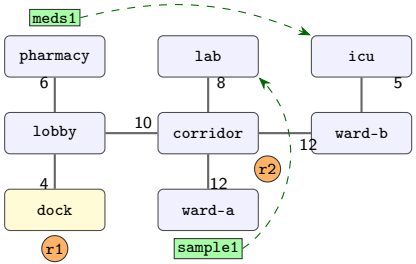
2 Coordination in time

3 Continuous change

4 Derived predicates and a world with a calendar

5 Summary

What a sequence cannot say

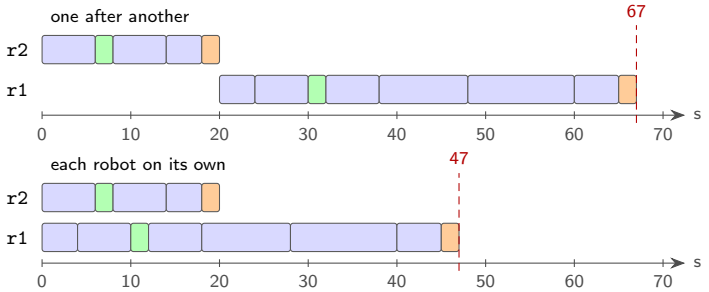


Two robots, two items, distances in meters.

A sequential planner returns **13 actions**, one per line. Missing:

- **How long** each action takes (12 m of corridor vs. lifting a box)
- **What can happen at the same time** (the robots do not need each other)
- **When the goal is reached** (what the nurse wants to know)

Two readings of the same thirteen actions

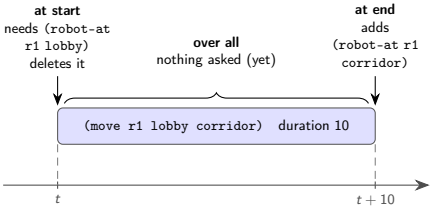


- A **timeline** (Gantt chart): time to the right, one row per thing that works. Blue moves, green picks, orange drops.
- A cost (Chapter 9) adds durations up: 67 in both drawings. It cannot see parallel work.

A first durative action

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

- `:duration`: a constraint on `?duration`
- `:condition`: each one `at start`, `over all` or `at end`
- `:effect`: `at start` or `at end` only
- Requirement `:durative-actions`



During the interval the robot is in **no room**.
PDDL has no unit of time: we read seconds.

Reading a temporal plan

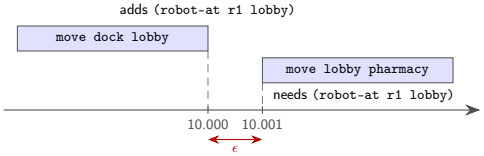
```
$ kairos courier-fixed-domain.pddl courier-fixed-p01.pddl
; cost = 52.006
; makespan = 52.006
0.000: (move r1 dock lobby) [10.000]
10.001: (move r1 lobby pharmacy) [10.000]
20.002: (pick r1 meds1 pharmacy) [2.000]
20.003: (move r1 pharmacy lobby) [10.000]
30.004: (move r1 lobby corridor) [10.000]
40.005: (move r1 corridor ward-a) [10.000]
50.006: (drop r1 meds1 ward-a) [2.000]
```

- Lines `t: (action) [d]`: start, duration; sorted by start, *nothing more*
- **Makespan**: end of the last action, $50.006 + 2$

1. 10.001, not 10: correct (next slide)
2. pick at 20.002, leaves at 20.003: a defect of *our model*

What happens at each instant

- The world changes only at starts and ends of actions: the **happenings**
- *No moving targets*: happenings **interfere** when one changes a fact the other needs or changes
- Interfering happenings are ϵ apart (Kairos: 0.001, `-epsilon`); others may share an instant. Without it, rejected:




```
$ 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]
```

Conditions in time: invariants

What must stay true *while* the action runs is an **invariant**: `over all`.

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

Step D: three questions per condition

1. Needed to *begin*? `at start`
2. Would the action fail if it stopped halfway? Also `over all`
3. Must hold when it *completes*? Also `at end`

Specifier	Checked	Typical use
`at start`	at t	what the action needs to begin: where the robot is, a free gripper
`over all`	in the **open** interval $(t, t+d)$	what must not change: the robot stays, the door stays open
`at end`	at $t+d$	what must be true to complete: the lift is still at the floor

Durations that depend on the action

```
(:durative-action move
  :parameters (?r - robot ?from ?to - location)
  :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
```

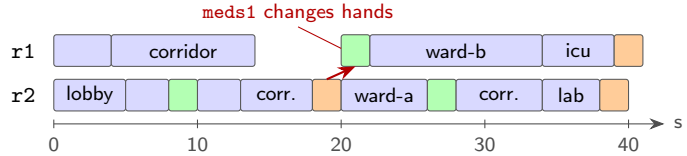
- Any numeric expression: distance over speed, evaluated **when the action starts**
- Static functions speed, handling-time; requirement :numeric-fluents
- (:metric minimize (total-time)): the makespan, in every problem
- Honest now: pick ends at 12.002, the move starts at 12.002; the over all interval is open

```
$ 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]
```

Working in parallel

Nothing new is needed: actions overlap unless the model keeps them apart.

- `r1` alone: 95 s. Two robots, one delivery each: 47.006. Searching on: 41.007



- A relay nobody asked for: the fast `r2` hands `meds1` over in the corridor.
- **The makespan is not the work**: 67 robot-seconds in the plan of 47, 75 in the plan of 41.
- No proof of optimality for the makespan. Pauses are part of plans: no `wait` action.

One thing at a time

Add a UV lamp, `disinfect` (20 s), with only its invariant:

```
$ 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]
```

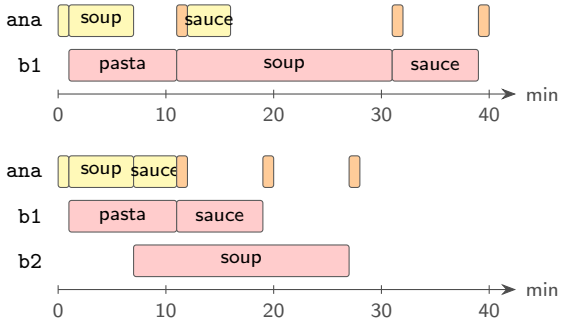
The robot picks while it disinfects. Pattern *resource lock* (Chapter 7), now for a single action. Invariants say what the action relies on; the token says the agent is one.

The robot is a **resource**: a token `(idle ?r)`, taken at start, returned 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)))))
```

With `idle` in every action: 30.003, one action after another.

Other domains: a kitchen and an elevator



One burner: 40 min. Two burners: 28. Yellow prepare, red cook, orange serve.

- `cook` has no cook: the burner works alone (token `burner-free`)
- The row **without gaps** is the bottleneck: with one burner, a second cook would not help
- Elevator: `board` asks `(lift-at ?e ?f)` at start, over all and at end. An invariant lets passengers **share** the lift; a token keeps users apart

The MODEL method: time

What Chapter 10 adds

Step M. *Success*: does it matter *when* the goal is reached? Then time and (`total-time`).
Agents: who can act at the same time as whom?
Step D. Four new columns in the action table:

- Does the duration matter, and what does it depend on?
- What must hold at the start, throughout and at the end?
- When does each effect happen? Taken at the start, produced at the end
- What may run in parallel with it? What must not needs a token

Step E. The cheat test becomes a family: an agent in two places at once; a goal whose shortest honest timetable you know; every pair of overlapping lines checked against the world.

Alternatives compared

Instantaneous actions with a cost, or durative actions (two robots, two deliveries)

	Cost (`courier-cost`)	Durative (`courier-time`)
Fidelity	Total work, 52.5 s, proved optimal; `r2` does everything	Timetable: makespan 41.007; every overlap left open is used
Guidance	3336 states, optimal in 13 ms	first plan in 2 ms, best after 0.9 s; no proof
Portability	any numeric planner	temporal planners only

Where an effect goes (`courier-walk`; in bold, plans the world does not allow)

	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

Verdict. A cost for least total work; durative actions when things overlap and the finishing time matters. **Take at start, give at end.**

Common mistakes

Renaming the parts of an action

```
$ 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]
...
```

Each specifier is a decision; the checker will not take it for you.

At start for what must hold all the way

An agent starts something and leaves. Add `over all`, validate the suspicious plan.

A token forgotten in one action

That action overlaps with all others; forget to return it and the agent never acts again. No checker warns.

Reading the lines as a sequence

Draw the timeline, one row per agent or resource.

1 Durative actions

2 Coordination in time

3 Continuous change

4 Derived predicates and a world with a calendar

5 Summary

Plans that exist only if actions overlap

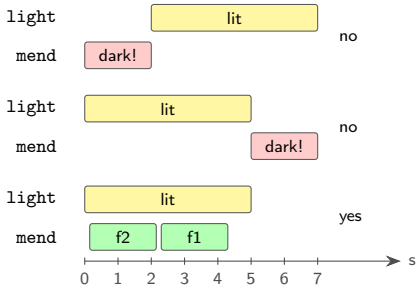
```
;; There is light from the start of the action to its end.
(:durative-action light
  :parameters (?m - match)
  :duration (= ?duration 5)
  :condition (at start (unused ?m))
  :effect (and (at start (not (unused ?m)))
               (at start (lit ?m))
               (at end (not (lit ?m)))))

;; Mending takes 2 seconds, all of them with light.
(:durative-action mend
  :parameters (?f - fuse ?m - match)
  :duration (= ?duration 2)
  :condition (and (at start (hand-free))
                  (at start (lit ?m))
                  (over all (lit ?m)))
  :effect (and (at start (not (hand-free)))
               (at end (mended ?f))
               (at end (hand-free)))))
```

- A match burns 5 s; mending a fuse takes 2 s and needs light
- `light` adds `(lit ?m)` at start and deletes it at end, on purpose: true **exactly while it runs**
- `mend` needs it at start and over all

```
$ 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]
```

Required concurrency



- No sequence of whole actions works: before or after the light, the mend is in the dark
- **Required concurrency**: every plan contains actions that overlap
- One match serves two fuses (4 s fit in 5), not three. Exit 14 is weaker than `unsolvable` (10): no proof

```
$ kairos cellar-domain.pddl
    cellar-p03.pddl
...
kairos: no plan found (exhausted)
$ echo $?
14
```

The envelope: a door held open

An **envelope** adds a fact at its start and deletes it at its end; its **contents** need the fact at start and over all.

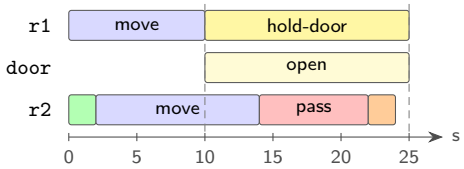
```
(:durative-action hold-door
  :parameters (?r - robot ?d - door ?l ?other - location)
  :duration (= ?duration 15)
  :condition (and (at start (idle ?r))
                  (at start (robot-at ?r ?l))
                  (over all (robot-at ?r ?l))
                  (at start (joins ?d ?l ?other)))
  :effect (and (at start (not (idle ?r)))
               (at start (open ?d))
               (at end (not (open ?d)))
               (at end (idle ?r))))
```

```
(:durative-action pass
  :parameters (?r - robot ?d - door ?from ?to - location)
  ; ...
  :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))))
```

Step D: an action that provides something while it runs

Ask what each action makes true *only while it lasts*. Then: who performs the envelope, who the contents, and can they be the same agent? Here `idle` answers: no.

The door plan



- The middle row is the fact `(open lab-door)`; pass lies inside
- Makespan 25.001: the end of the envelope, although the sample is in the lab at 24.003
- The hold starts 4 s early: nothing says “as late as possible”
- A robot alone: `unsolvable`, with a proof

```
$ 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)
```

Temporally simple, temporally expressive

- **Temporally expressive** language: it can state required concurrency; PDDL 2.1 is
- **Sequence first, schedule later**: fast, classical search; cannot solve the cellar
- **Search over starts and ends**: solves it, larger space (POPF, OPTIC)

What Kairos does

Engine **M2**: compresses each durative action and schedules. Engine **M3**: searches over happenings, times in a temporal network. `-v` shows the choice; `-engine m2` on the cellar: 2 states, no plan found.

```
$ 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
...
```

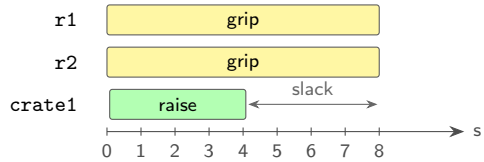
Working together: two robots and a crate

Brittle: each `lift` needs the partner's grip over `all`. Only plans: both start at *exactly* the same instant.

- validate same start: `plan valid, makespan 4`
- search: `no plan found`; one start after another, the other invariant fails

With slack: a grip is an envelope of 8 s; `raise` (4 s) runs inside both.

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



One robot playing both parts

Without `(at start (not (= ?r1 ?r2)))`: `(raise crate1 r1 r1 dock)`. Whenever a model says "another", there must be a token or an inequality.

Durations that the planner chooses

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

- Bounds instead of a value: `:duration-inequalities`
- `?duration` in an effect: the choice has a consequence
- 20 units, 31 m to the ICU: 5.5 s needed. By hand: valid, 36.504

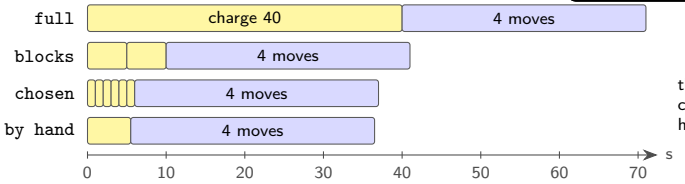
```
$ kairos courier-charge-domain.
    pddl courier-charge-p01.pddl
; cost = 71.00399999999999
; makespan = 71.004
0.000: (charge r1 dock) [40.000]
...
; better plan 2
; cost = 37.00899999999999
; makespan = 37.009
...
5.005: (charge r1 dock) [1.000]
6.006: (move r1 dock lobby)
    [4.000]
```

How a planner chooses a duration

- Kairos tries each duration at its **bounds**: full charge, minimum repeated
- If that fails, **steps of time** (0.1 s, or the instant at which ending makes a condition true)
- The validator accepts any duration within the bounds

A coffee machine: 25 ml/s, an espresso is 40 to 50 ml, bounds 1 to 10 s.

```
$ kairos coffee-domain.pddl
    coffee-p01.pddl
; cost = 1.6
; makespan = 1.600
0.000: (pour cup1) [1.600]
```



three encodings of the charge, and the plan by hand

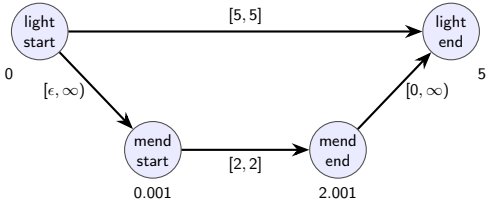
How planners order time

Simple temporal networks

A variable per time point (each start, each end); constraints $l \leq t_j - t_i \leq u$:

- duration 5: $t_{end} - t_{start} = 5$
- chosen duration: $1 \leq t_{end} - t_{start} \leq 40$
- after another happening: $t_j - t_i \geq \epsilon$

Consistent iff no cycle is a contradiction: shortest paths, no search.



The STN of `cellar-p01`: earliest times 0, 0.001, 2.001, 5. A mend of 5 s would need $\epsilon + 5$ where the direct arrow allows 5: no plan.

The MODEL method: coordination

What Chapter 11 adds

Step D. *Must it overlap with another action?* (envelope, content, joint work: which contains which, how much slack). *Is its duration a decision?* Then bounds, and an effect or condition that depends on `?duration`. **Step E.** A content that does not fit; the envelope's agent as its own content; one object in two roles. "No plan found" is not a proof.

	Envelope (`hold-door`)	Lock (`open-door`, `release-door`)
Fidelity	door always closes, after exactly 15 s (25.001)	open as long as needed (24.003); stays open unless the goal says otherwise
Guidance	required concurrency, 1919 states; no proof	temporally simple: engine M2
Portability	planners with required concurrency	any temporal planner

Verdict. Envelope when it ends by itself (a match, a pass of a satellite); lock when the agent decides when to stop; both: an envelope with `(<= ?duration 15)`.

Common mistakes

An envelope by accident

A "busy" flag added at start and deleted at end, read somewhere else. Use a token deleted at start and added at end, as `idle`.

Expecting the best duration

A planner that tries the ends gives plans at the ends.

"No plan found" read as "unsolvable"

```
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]
```

A chosen duration nothing depends on

The quantity must grow with `?duration`, and the upper bound stop it.

1 Durative actions

2 Coordination in time

3 Continuous change

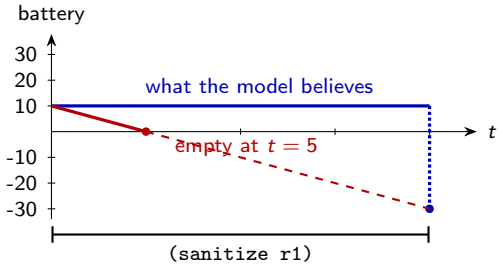
4 Derived predicates and a world with a calendar

5 Summary

A staircase where the world has a ramp

- Battery 10 of 30; a UV lamp of 20 s takes 2/s (40 in all); a charger gives 5/s
- **Careful** discrete model (pay at start, charge credited at end): `no plan found`
- **Careless** one (lamp paid at end): the lamp alone. Worse than no plan

```
$ kairos lamp-careless-domain.pddl
    lamp-careless-p01.pddl
; cost = 20
; makespan = 20.000
0.000: (sanitize r1) [20.000]
```



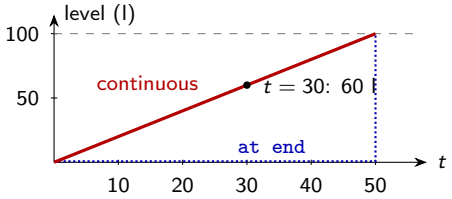
Discrete and continuous change

A **discrete change** happens at an instant; a **continuous change** during an interval, at a **rate**. Constant rate: **linear** (all of this chapter).

Continuous effects

```
(:durative-action fill
  :parameters (?t - tank)
  :duration (and
    (>= ?duration 1)
    (<= ?duration (/ (- (capacity ?t) (level ?t)) (flow ?t))))
  :condition (at start (tap-closed ?t))
  :effect (and
    (at start (not (tap-closed ?t)))
    (at end (tap-closed ?t))
    (increase (level ?t) (* #t (flow ?t))))))
```

- **No time label**: at work during the whole action
- `#t`: time since the action started; its factor is the **rate**
- Only `increase` or `decrease`; requirement `:continuous-effects`
- Same plan as an effect at end, but the level has a value at every second

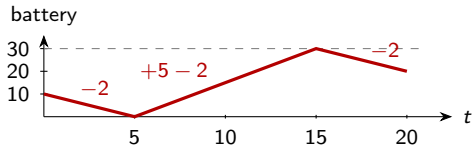



```
$ kairos tank-domain.pddl
    tank-p01.pddl
; cost = 50
; makespan = 50.000
0.000: (fill t1) [50.000]
```

Reading a number while it changes

```
(:durative-action charge
  :parameters (?r - robot ?l - location)
  :condition (and
    (at start (idle ?r))
    (at start (has-charger ?l))
    (over all (robot-at ?r ?l))
    (over all (<= (battery ?r) (battery-capacity ?r))))
  :effect (and
    (at start (not (idle ?r)))
    (increase (battery ?r) (* #t (charge-rate ?r)))
    (at end (idle ?r))))
(:durative-action sanitize
  :parameters (?r - robot)
  :duration (= ?duration 20)
  :condition (and
    (at start (lamp-off ?r))
    (over all (>= (battery ?r) 0)))
  :effect (and
    (at start (not (lamp-off ?r)))
    (decrease (battery ?r) (* #t (lamp-rate ?r)))
    (at end (lamp-off ?r))
    (at end (sanitized ?r)))))
```

```
$ 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]
...
```



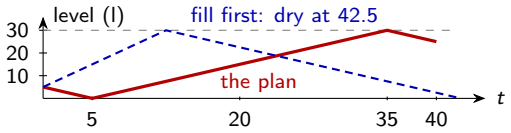
Water that comes and goes: the greenhouse

```
(:durative-action fill
  :parameters (?t - tank)
  :duration (and (>= ?duration 1) (<= ?duration 120))
  :condition (and
    (at start (tap-closed ?t))
    (over all (<= (level ?t) (capacity ?t))))
  :effect (and
    (at start (not (tap-closed ?t)))
    (increase (level ?t) (* #t (flow ?t)))
    (at end (tap-closed ?t))))
```

A general rule

A bound computed at the start of an action is only right if nothing else changes the number meanwhile. Otherwise: a constant bound and an `over all` limit.

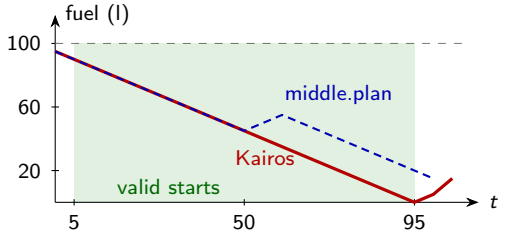
```
$ kairos greenhouse-domain.pddl
    greenhouse-p01.pddl
; cost = 40
; makespan = 40.000
0.000: (irrigate b1 t1) [40.000]
5.000: (fill t1) [30.000]
...
```



The generator: when to start

- `generate`: 100 s at 1 l/s, never empty (`over all`)
- `refuel`: 10 s at 2 l/s, never overflowing (`over all`)
- 95 of 100 l, one jerrycan: poured before 5 it overflows; after 95 the tank is dry
- Durations fixed: the planner must find an **instant**

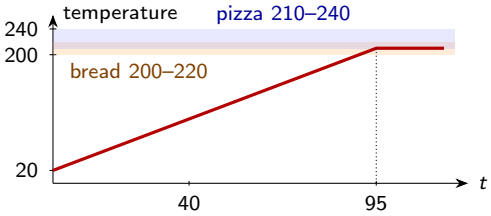
```
$ kairos generator-domain.pddl
    generator-p01.pddl
; cost = 105
; makespan = 105.000
0.000: (generate gen) [100.000]
95.000: (refuel gen j1) [10.000]
```



Kairos pours at the last instant. With 60 l: `no plan found`, exit 14; the proof is one subtraction.

The oven: when to stop

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




```
$ kairos oven-domain.pddl oven-p02
    .pddl
...
; 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]
```

- Heater 2 deg/s for 1 to 300 s. Bread alone: exactly 90 s
- Two dishes: stop at 210, in both ranges; bread goes in at 94.9, still rising

What the planner must find

How Kairos plans continuous change

As discrete changes: a number read only as "large enough" or only as "small enough": one jump at the safest instant; durations at their bounds. **In steps of time**: 0.1 s, or a jump to where a condition is about to change. **Validation**: every plan checked in continuous time.

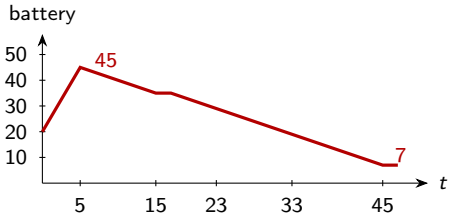
- Durations at their bounds, or where a condition fixes them
- Plans wait until the last moment
- Without a plan: often `no plan found` (14), not `unsolvable` (10)
- Habit: write the plan you believe in and `kairos validate` it (exact for linear change)

```
$ 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+
```

The courier world: a battery that drains and charges

```
(:durative-action move
  :parameters (?r - robot ?from ?to - location)
  :duration (= ?duration (/ (distance ?from ?to) (speed ?r)))
  :condition (and
    (at start (robot-at ?r ?from))
    (at start (idle ?r))
    (over all (connected ?from ?to))
    (over all (>= (battery ?r) 0)))
  :effect (and
    (at start (not (robot-at ?r ?from)))
    (at start (not (idle ?r)))
    (decrease (battery ?r) (* #t (drain-rate ?r)))
    (at end (robot-at ?r ?to))
    (at end (idle ?r))))
```

`charge`: from 5 s to the time to a full battery, +5/s. A bound computed at start is right here: `idle` keeps moves out.



`courier-battery-p02`: 20 units, 38 needed.

Charge 5 s (the minimum), arrive with 7.

- Cheat test (p05): a move started with 8 units for 12 m is rejected, `-4 >= 0 is false`
- Round of 90 m (p03): best plan charges 8 s, to full; 6 s would do

The MODEL method: continuous change

New questions for steps O and D

O: does the quantity change *while* an action runs? At what rate? **D**: does anybody read it meanwhile? Can another action change it at the same time? What bounds a free duration? *If nobody reads it meanwhile and nobody else changes it, a discrete change is enough*: at start if consumed, at end if produced.

	Careless discrete	Careful discrete	Continuous
Fidelity	impossible plans: battery at -30	loses plans that use energy as it arrives	exact for constant rates
Guidance	easy, and wrong	ordinary temporal search	instants and durations; often no proof
Clarity	hides the rule	totals by hand	rates and limits as in the world
Portability	any temporal planner with numbers	Kairos, OPTIC, POPF	Kairos and POPF

Verdict. Never careless. Continuous when producers and consumers overlap (lamp and charger, tap and beds); processes (Chapter 17) when it changes without the agent.

Common mistakes

#t under a time label

```
$ 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]
...
```

A limit checked only at the start

The number moves and nobody looks:
`lamp-alone.plan` is valid for the careless model.
Limits during an action are `over all`.

`no plan found` read as "no plan"

Exit 14 is not a proof; exit 10 is.

A requirement left out

Declare `:continuous-effects`; other planners stop at it.

1 Durative actions

2 Coordination in time

3 Continuous change

4 Derived predicates and a world with a calendar

5 Summary

Facts that follow from other facts

- `(free ?r)` is redundant with `carrying`, kept in step by hand
- "The ICU can be reached from the dock": keeping it by effects is hopeless
- **Basic predicate**: stated (initial state, effects). **Derived predicate**: computed by a rule

```
(:derived (free ?r - robot)
  (not (exists (?i - item) (carrying ?r ?i))))
```

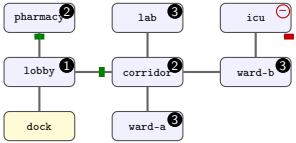
Head and body; `:derived-predicates`. `pick` and `drop` no longer touch `free`.

- **Read** anywhere: preconditions, conditions, goals, other rules
- **Never written**: not in effects, not in the initial state
- Several rules for one head: an `or`

```
$ 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]
...
```

Rules that call themselves

```
(:derived (passable ?a ?b - location)
  (and (connected ?a ?b)
       (forall (?d - door)
         (imply (or (joins ?d ?a ?b) (joins ?d ?b ?a))
                (open ?d)))))
(:derived (reachable ?a ?b - location)
  (or (passable ?a ?b)
      (exists (?c - location)
        (and (passable ?a ?c) (reachable ?c ?b)))))
(: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)))
```



`(reachable dock ?b)`, round by round; `d-icu` closed (red).

- A **recursive rule**: paths of any length, impossible with `exists` alone
- Built from nothing, round by round, until a round adds nothing
- go crosses any number of rooms: navigation is not the planner's business

Doors as the means, reachability as the consequence

All doors closed (`courier-reach-p02`):

```
$ 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)
```

No action adds `reachable`; each `open-door` changes which go is applicable.

A derived predicate in a goal, negated (p03):

```
(:goal (and (item-at meds1 icu)
            (robot-at r1 dock)
            (not (reachable lobby icu)))))
```

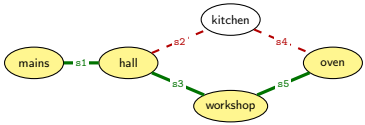
- Two plans close different doors: the corridor's, or the ICU's
- The goal says *what*, the planner finds *how*

Towers and circuits

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

- Oven powered, kitchen not (an electrician works there)
- s4 must open too: `linked` is symmetric, the oven would feed the kitchen. Found in the rules, not in any action
- Blocks: `above` as a recursive rule; the planner picks the arrangement
- Rules must be **stratified**: no predicate depends negatively on itself




```
$ kairos circuit-domain.pddl
      circuit-p01.pddl
; cost = 5
(open-switch s2)
(open-switch s4)
(close-switch s3)
(close-switch s1)
(close-switch s5)
```

What a planner does with rules

A state stores basic facts only; in every new state the rules run until nothing new comes out.

```
$ 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)
...
```

- Two rules, 518 ground rules (*axioms*): three variables over n objects, in the order of n^3
- Heuristics must reason through the rules
- Fewer planners (Kairos, LPG-td, LAMA in the book's check)

Why not compile the rules away?

For `free`, write the `exists` in `pick`. For a recursive rule there is nothing to substitute: removing axioms makes the domain or the plans exponentially larger in the worst case (Thiébaux, Hoffmann and Nebel).

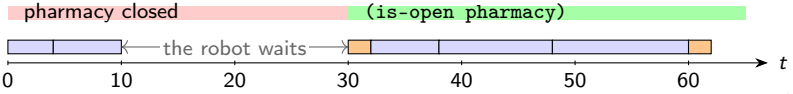
A world with a calendar: timed initial literals

In the `:init` of the problem:

```
(at 30 (is-open pharmacy))
```

- At time 30 the literal becomes true, whatever the plan does; `(at 50 (not ...))` makes it false
- Requirement `:timed-initial-literals`
- `pick` and `deliver` ask `(over all (is-open ?l))`

```
$ 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]
...
60.004: (deliver r1 meds1 ward-a) [2.000]
```



Windows and deadlines

- **Window**: one literal opens, a later one closes; actions need the fact `over all`
- **Deadline**: a fact "still in time", removed by a timed literal, needed `(at end (in-time ?i))` by `deliver`

```
(in-time meds1) (in-time meal1)
(at 50 (not (in-time meds1))))
```

Serving the meal first has the same length, and breaks the deadline:

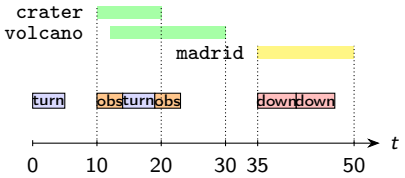
```
$ kairos validate courier-hours-domain.
    pddl courier-hours-p03.pddl
    courier-hours-mealfirst.plan
courier-hours-mealfirst.plan:14: error:
    step 13 (deliver r1 meds1 ward-a) at
    88.012: at end condition false: (
    in-time meds1) is false [validate.
    precondition_failed]
...
```

- A deadline of 20 (p04): `unsolvable`, a proof. Even ignoring deletes and with the shortest durations, `(delivered meds1)` comes after `(in-time meds1)` disappears

A satellite and its windows

```
$ 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]
```

```
(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)))
```

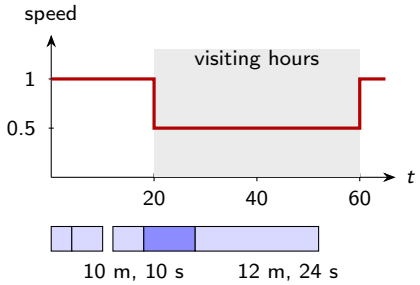


- It turns first to the crater, whose window closes sooner
- Crater visible 3 s, exposure 4 s (p02): `unsolvable`

Numbers on the calendar

```
(= (speed r1) 1)
(at 20 (= (speed r1) 0.5))    ; a timed initial fluent
(at 60 (= (speed r1) 1))
```

- A **timed initial fluent**: not PDDL 2.2, an extension with `:timed-initial-fluents`
- The last move, 12 m, takes 24 s
- The move started at 18.004 keeps its 10 s: a duration is computed **when the action starts**



The MODEL method: consequences and the calendar

New questions for steps M and O

M: does the world change at known times, whatever the agent does? A schedule under *Assumptions*; timed literals in the problem; `over all` or `at end` on the actions. **O**: is a property a consequence of others? A derived predicate: by a fixed number of facts (`free`) or a chain of unknown length (`reachable`).

`(free ?r)`	Kept by effects	Derived	Written out
Fidelity	right while every action keeps it in step	by construction	by construction
Guidance	best: heuristics see it	through the rule `:derived-predicates`	through a negated `exists`
Portability	any planner		negative, existential preconditions

- `free`: keeping it by effects is still a good choice. `reachable`: it cannot be kept by effects
- A deadline: `(within 50 (delivered meds1))` (PDDL 3) is clearer; timed literals are more portable and express windows

Common mistakes

A rule that only looks one step ahead

```
(:derived (reachable ?a ?b - location)
  (or (passable ?a ?b)
      (exists (?c - location)
        (and (passable ?a ?c) (passable ?c ?b)))))
```

Legal, and means something else. Test deeper than the rule.

Writing a derived predicate

An old effect or initial fact left in place: delete it.

The fact is already true

`(is-open pharmacy)` and `(at 30 (is-open pharmacy))`: the robot picks at 10. Write the first timed literal as the first *change*.

A window checked only at the start

An action begins before closing and ends in a closed room. Decide with the three questions; test with a window shorter than the action.

A requirement left out

`:timed-initial-literals` belongs with the others; Kairos warns.

1 Durative actions

2 Coordination in time

3 Continuous change

4 Derived predicates and a world with a calendar

5 Summary

What to take away

- A **durative action**: conditions `at start`, `over all`, `at end`; effects at start and end. A plan is a timetable; its makespan is `(total-time)`.
- Actions overlap unless the model keeps them apart: invariants, tokens (`idle`), ϵ between interfering happenings. Take at start, give at end.
- **Required concurrency**: envelopes. Kairos M2 vs. M3; `no plan found` (14) is not `unsolvable` (10). **Chosen durations**: bounds and `?duration` in effects; an STN turns orders into times.
- **Continuous effects**: rates with `#t`, they add up; limits `over all`. Use them when somebody reads the number in the middle.
- **Derived predicates**: facts computed by rules, recursive for paths. **Timed initial literals**: a calendar the world follows; windows and deadlines.

Exercises to try

- **10.1.1** Sequence vs. each robot on its own: draw the timeline
- **10.7.1** The token forgotten in one action: find it with a test
- **11.1.1** A mend as long as the match: which exit code, and why not 10?
- **11.7.2** The match cellar with a lock: what goes wrong
- **12.4.1** The generator starts with 90 liters: the window of valid starts
- **12.4.3** No pouring into a running generator: which problems keep a plan?
- **12.6.2** Remove the lower bound of `charge`: what plans for `p02`?
- **13.3.4** "On if not off" and "off if not on" as derived predicates
- **13.6.3** The satellite, volcano first: which condition fails, and when?