Hybrid systems: PDDL+

Planning and PDDL, Part VI (Chapters 17–18)

Francisco Martín Rico

Intelligent Robotics Lab, URJC



Outline

1. Processes and events
2. Modeling hybrid systems
3. Summary

1 Processes and events

2 Modeling hybrid systems

3 Summary

A world that does not wait

- So far the world changed only because the **agent** acted
- Chapter 12: the battery drains only during a `move`; an idle robot keeps it
- Real world: a sample warms up, a tank keeps filling, a ball falls
- Timed literals (ch. 13): fixed times. Here: *while* the robot is on, *when* the tank is full

Who owns the change?

Durative action: the agent; it cannot outlive the action. World: water flows *while the valve is open*; the agent only decides when to open and close.

PDDL+ (Fox and Long, 2006)

For **hybrid systems**: discrete state (valve) and continuous state (level). All of PDDL 2.1, plus:

- **process**: changes numbers continuously while its condition holds
- **event**: changes the state at the first instant its condition is true

Actions remain: the only decisions of the agent.

The tank: two actions, one process, one event

```
;; 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))
```

- Read it in three blocks:
  - what the **agent** decides: `open-valve`, `close-valve`
  - what the world does **while** a condition holds: `fill`
  - what the world does **at an instant**: `overflow`
- Actions are **instantaneous**: they are the only decisions
- A plan lists those decisions with their times; what the world does in between is not in the plan

Processes: change while a condition holds

```
(:process fill
  :parameters ()
  :precondition (valve-open)
  :effect (increase (level) (* #t (inflow))))
```

- `:precondition`: not something to achieve, but when the process **is running**
- `:effect`: only `increase`/`decrease` with `#t`; no facts, no `assign`
- `#t`: time running; `(inflow)`: the **rate**
- Never in a plan. Requirement `:time`

Indirect control

The agent does not control `fill`: it controls the fact `(valve-open)`.

Actions change facts, facts enable processes, processes change numbers.

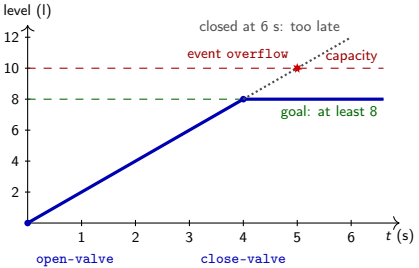
In one line

A process is an action that nobody chooses: its precondition says when it runs, its effect, always with `#t`, how fast.

The first plan

Goal of `tank-p01`: at least 8 l, valve closed, no flood

```
$ 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
```



The slope is the rate of `fill`. Dotted: the valve closed two seconds later.

- **The gap**: 0 to 4 s, `fill` runs: $2 \times 4 = 8$ l
- `@PlanEND`: time may pass after the last action
- Cost = duration, **not** minimized: few actions

The times are the plan

```
$ 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
```

The validator of Kairos replays any plan in continuous time.

- Any closing time in $[4, 5)$ works
- Classical: *which* actions; durative: also *order and overlap*
- PDDL+: the **times themselves**: valid at 4.5 s, invalid at 3 or 6
- Search over instants: expensive, so problems stay small

Events: change at an instant

```
(:event overflow
  :parameters ()
  :precondition (and (>= (level) (capacity)) (not (flooded)))
  :effect (flooded))
```

- Effects of an action: facts, `assign`, jumps
- **Fires** at the first instant its precondition holds: never earlier, later or skipped

- Avoided only by keeping its condition false
- It leaves a mark, the goal refuses it
- **It must make its own precondition false**: `(not (flooded))`

```
$ 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
```

An event as the goal

`tank-p02`: the goal is 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, then five seconds in which the agent does nothing
- The plan ends at `@PlanEND`, when the event has fired
- Here the end matters: `open-valve` alone, ending at time zero, does not reach the goal

Without `(not (flooded))`

At 5 s the event fires and its condition still holds (the level is at the capacity): it would fire again and again in the same instant. PDDL+ forbids it.

A problem without a plan

`tank-p03`: 12 l 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]
...
kairos: no plan found (exhausted)
```

- Not the `unsolvable` of the classical chapters: exit code 14, not 10
- Kairos tried every timing it considers, not every instant: no proof claimed
- Step E: "must have no plan" becomes "Kairos finds none", *plus* our own reasoning

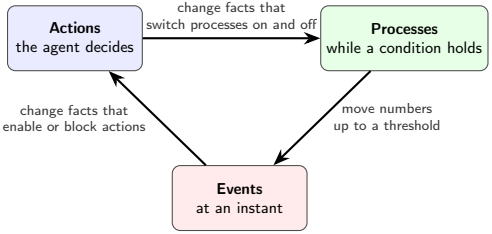
Three kinds of change

	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

The start-process-stop pattern

To give the agent an activity it can start and end: an **action** that adds a fact, a **process** that runs while the fact holds, an **action or event** that deletes it. Tank: `open-valve`, `fill`, `close-valve`. In every model of the chapter.

How the three feed each other



Only the box on the left is under the control of the planner.

What happens between two actions

1. No event condition true: **time flows**, every active process changes its numbers at its rate
2. An event condition becomes true: time stops, the event fires, others may follow in the same instant
3. Then time flows again

Two processes on the same number: **rates add up**. Events firing for ever in one instant: the model is wrong.

The bouncing ball

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

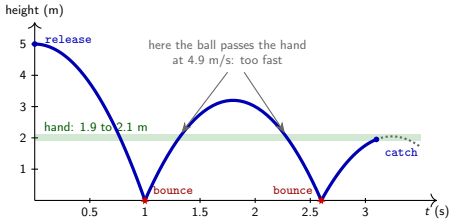
- **One process, two effects**: the rate of the height is the velocity, which is itself changing: a parabola
- **An event that assigns**: `bounce` keeps 80% of the speed, upwards; the velocity becomes positive, so it switches itself off
- **A narrow window**: `catch` only within 10 cm of the hand and slower than 2 m/s

The ball: we write the laws, the planner finds when

`ball-p01`: from 5 m, gravity 10, catch after 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
```

- Floor at $t = 1$ s, -10 m/s; bounce: $+8$
- Floor again at $t = 2.6$, -8; bounce: $+6.4$
- At $t = 3.1$: 1.95 m, 1.4 m/s: inside the window



Each arc is the process `fall`; the stars are `bounce`. Only near the top of the third arc is the ball slow enough.

The car: the driver only changes the acceleration

```
;; Parking needs the car at rest and the pedals released
(:action park
  :parameters ()
  :precondition (and (running) (= (acceleration) 0)
                     (>= (velocity) -0.05) (<= (velocity) 0.05))
  :effect (and (not (running)) (parked)
               (assign (velocity) 0)))

(:process moving
  :parameters ()
  :precondition (running)
  :effect (and
    (increase (distance-run) (* #t (velocity)))
    (increase (velocity) (* #t (acceleration)))))

(:event radar
  :parameters ()
  :precondition (and (> (velocity) (speed-limit))
                     (not (fined)))
  :effect (fined)))
```

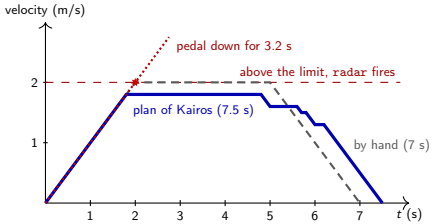
- `accelerate`, `brake`: ± 1 of acceleration, in $\{-1, 0, 1\}$
- `moving`: acceleration $\rightarrow$ velocity $\rightarrow$ distance
- `radar`: marks `fined` above the limit

Step D: who causes each change

Agent: pedals, parking (3 actions). *World, while*: velocity and distance (1 process). *World, at an instant*: the radar (1 event). `drive-ten-meters` would give the agent a power it does not have.

The car: valid is not good

```
$ kairos car-domain.pddl car-p01.
    pddl
; cost = 33.8
0.000: (accelerate) [0.000]
0.300: (brake) [0.000]
...
; better plan 2
; cost = 7.5
0.000: (accelerate) [0.000]
1.800: (brake) [0.000]
...
7.500: (park) [0.000]
```



- Plan 1: five actions, the least; crawls at 0.3 m/s for half a minute
- Plan 2: valid; an instructor would not sign it
- By hand (dashed), 2 + 3 + 2 s: valid, makespan 7
- Pedal down 3.2 s: `(fined) is true`

The courier world: a move is an action, a process and an event

```
;; --- 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)))
```

- The battery now drains **whenever the robot is on**
- `start-move`: the decision; resets the odometer; on the way, in no room
- `travel`: while moving *and switched on*
- `arrive`: at the distance; deletes `moving`, stopping both
- An event on purpose: at the distance, it *is* there

The battery: a process and an event

```
;; 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)))
```

- `drain` does not care about moving
- `shutdown` deletes `(powered ?r)`: stops itself, `drain` and `travel`
- Empty between two rooms: stuck for good
- No precondition "enough battery for the trip": the laws do that job

Step 0: the state of a hybrid model

Two new kinds of state:

- facts that only switch processes: `(powered ?r)`, `(moving ?r ?f ?t)`
- numbers that measure a process: `(traveled ?r)`

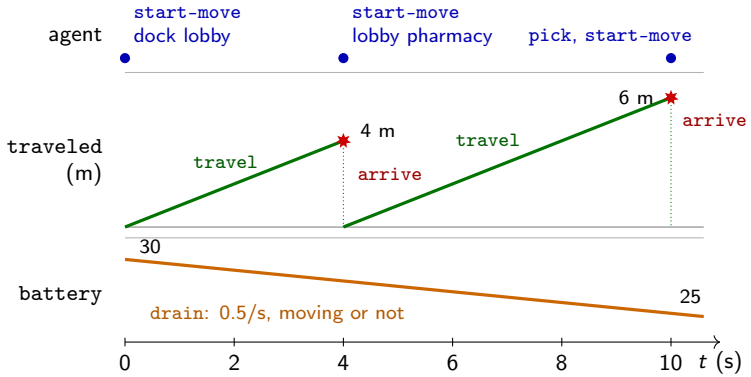
For every process: which fact starts and stops it, and which number tells how far it has gone?

Exercising the model

```
$ kairos courier-battery-domain.pddl
    courier-battery-p01.pddl
; cost = 38
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
```

- Battery 30, drain 0.5/s: 60 s on; trip 38 m at 1 m/s
- Only the **starts** of the moves: arrivals are events, read off the distances
- Each `start-move` at the instant of the previous arrival
- End: $30 - 0.5 \times 38 = 11$
- p02, battery 8 (16 s): `no plan found` after 221 664 states

The first ten seconds, in three lanes



Top: decisions. Middle: `travel` raises the odometer until `arrive` fires; the next `start-move` resets it. Bottom: `drain` does not care.

The cheat test: a robot that coasts

```
$ kairos courier-battery-coast-domain.pddl
    courier-battery-coast-p02.pddl
; cost = 38
...
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]
...
38.000: (drop r1 meds1 ward-a) [0.000]
```

- Bug: `travel` with `(moving ?r ?from ?to)` alone, without `(powered ?r)`
- Solves `p01` with the same plan
- On `p02` (battery 8, must have no plan): start a move, switch off, the wrong process keeps pushing
- Battery 0.3 at the first `power-off`: enough for the day
- A perpetual motion machine, because the model contained one

The correct model rejects the free ride

```
$ 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
```

Switched off, the robot stopped 60 cm short of the lobby. Classical models: the cheat was a missing precondition; hybrid models: a missing **condition of a process**.

The planner looks for free rides

A process that runs when it should not, or fails to run when it should. Test with a problem whose only plans would need one.

A sample that warms up

```
;; 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)))
```

- `warm-up` runs for every sample not `(chilled ?i)`: no action "starts" it
- The agent controls it through the facts of its condition: `pick` takes the sample out, `store` puts it in a fridge
- `courier-cold-p01`: robot in the corridor, meds for ward A; sample in the lab at 4°, 0.1°/s, spoiled at 8: 40 s out of a fridge; fridges in lab and ICU

Medicines first fails

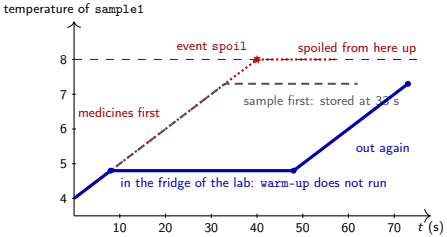
A planner that counts actions is tempted to deliver the medicines first: ten actions. We write that plan by hand and validate it.

```
$ 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 is applicable; it fails because of something **no action did**
- The robot reaches the lab at 32 s and the ICU at 57; the sample was spoiled at 40

The planner parks the sample

```
$ kairos courier-cold-domain.pddl
    courier-cold-p01.pddl
...
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]
...
48.000: (pick r1 sample1 lab)
    [0.000]
...
73.000: (store r1 sample1 icu)
    [0.000]
```



- In the fridge *of the lab* from 8 to 48 s; ICU at 7.3°: what a sensible nurse would do
- Sample first: 62 s, eleven actions, also valid

How Kairos plans PDDL+

Search in steps of time

Kairos adds actions of its own that only let time pass:

- a **short wait**: 0.1 s
- a **long wait**: steps of 0.1 s until a numeric condition is about to change

Processes are integrated with **Runge-Kutta**; events fire between steps. Waits are the gaps between the lines of the plan.

Validation in continuous time

A stepped plan is only a candidate. Before printing it, Kairos integrates exactly and fires each event at the instant its condition is crossed. Rejected candidates are discarded. `kairos validate` applies the same check to any plan.

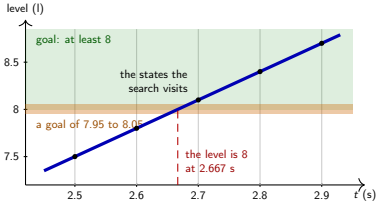
- Times are multiples of 0.1: valve at 4.0, not 4.5
- Coasting robot off at 15.4: last step before empty
- Car plan 1: few search steps, a lot of time

What it can miss

`tank-p04`: inflow 3 l/s; 8 l at 2.667 s

```
$ 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
```

- "At least 8": a wide target, a step lands on it (8.1 l)
- A goal of 7.95–8.05 l: `no plan found`, although a plan exists



Other ways to plan PDDL+

Fixed step (ENHSP), or equations for a constraint solver; all check against the continuous semantics (VAL). Of the eight planners of Appendix B only Kairos planned the test domain with processes: the least portable models of the book.

The MODEL method: who causes each change

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 step D: the agent falls asleep

Whatever still happens is a process or an event; whatever stops was an action. The water keeps flowing (process), the tank overflows (event), the valve stays as it was (actions).

Alternatives compared: the end of a move

	Arrival as an event	**Arrival as an action**	**Durative move**
Fidelity	Good; a robot that dies on the way stays there	Robot may linger between rooms (44 s plan)	Good; a shutdown breaks `over all`: plan invalid
Size	32 actions, 15 processes, 15 events	46 actions, 15 processes, 1 event	18 actions, 14 durative, 1 process, 1 event
Guidance	76 139 states, 0.4 s	290 833 states, 1.4 s	279 states, 0.01 s
Clarity	Three pieces; plan shows only starts	Starts and stops, like a log	One piece; plan shows durations
Portability	PDDL+ only	PDDL+ only	PDDL+ planners that also accept durative actions

Verdict

An activity that ends by itself: do not make its end a decision. Prefer the **durative action** when the duration is known at the start and nothing interrupts it (search two orders of magnitude smaller). Keep the **event** when the end depends on how the world evolves.

Common mistakes (1)

An event that does not switch itself off

`overflow` with `(>= (level) (capacity))` alone: `kairos check` accepts it; the validator says `events keep firing at 5.000000 (overflow): an event must make its own condition false`. Fix: a fact it adds and negates, a fact it deletes, or a number it moves back across the threshold.

A process without #t, or #t outside a process

`(increase (level) (inflow))` in a process: `processes can only have continuous effects (with #t)`. `#t` in `open-valve`: `#t can only appear in increase/decrease effects of durative actions and processes`. Fix: the action adds a fact, a process does the filling.

A process that runs when it should not

The coasting robot: nothing rejects the domain, ordinary problems give ordinary plans. Fix: write in the precondition everything that must hold for the change to go on; run the cheat test.

Common mistakes (2)

Reading "no plan found" as "unsolvable"

`kairos: no plan found (exhausted)`, exit code 14: all the timings on its grid, not all the timings. Write the expected plan by hand and give it to `kairos validate`, which has no grid.

Forgetting `:time`, or the keywords of a durative action

Without `:time`: accepted with a warning (`uses processes but :requirements does not include :time`); other planners may refuse the file. Processes and events take `:precondition` and `:effect`; `:condition` is an error.

1 Processes and events

2 Modeling hybrid systems

3 Summary

Rules that hold at all times: global constraints

```
(:constraint keep-the-limit
  :parameters ()
  :condition (<= (velocity) (speed-limit)))
```

Replaces radar event + goal (`not (fined)`)

- Holds at **every instant**; no effect, it only forbids
- From ENHSP, not original PDDL+; not PDDL 3 (`:constraints ...`)

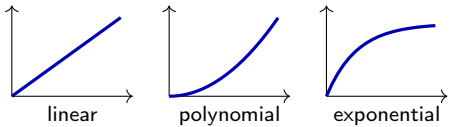
```
$ kairos validate car-limit-domain.pddl car-limit-p01.pddl car-limit-p01-fast.
    plan
car-limit-p01-fast.plan: error: constraint 'keep-the-limit' broken at 2.000001
    : (<= (velocity) (speed-limit)): 2 <= 2 is false [validate.
    global_constraint]
plan invalid
```

Pedal down for 3.2 s: which rule broke, and when. **Constraint**: what must not happen at all. **Event**: a bad thing with consequences (a fine, a spoiled sample).

Change that is not linear

Effect (* #t *rate*): the rate decides the shape

- **constant**: a straight line (**linear**, chapter 12)
- **another changing quantity**: distance at rate velocity: a parabola
- **the quantity itself**: cooling, air drag: approaches a limit



Why a staircase is not enough

Rate fixed during each step: exact for linear change, wrong otherwise. The ball from 5 m in steps of 0.1 s: 0.5 m after one second, when it is on the floor. Kairos integrates with **Runge-Kutta**: planned and validated exactly; the heuristics linearize, which costs search effort but never validity.

Portability: planners for chapter 12 usually expect constant rates.

A car with air drag

```
(:process moving
  :parameters ()
  :precondition (running)
  :effect (and
    (increase (distance-run) (* #t (velocity)))
    (increase (velocity) (* #t (acceleration)))))

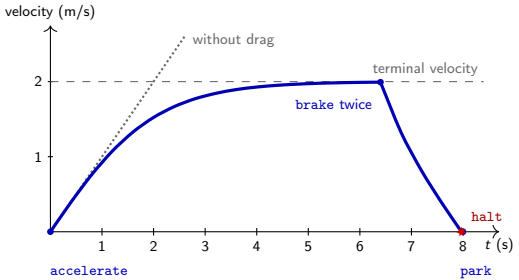
;; A second process on the same quantity: the two rates add up
(:process air-drag
  :parameters ()
  :precondition (and (running) (> (velocity) 0))
  :effect (decrease (velocity)
            (* #t (* (drag) (* (velocity) (velocity))))))

;; 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))))
```

- Two processes on the velocity: rates add up, $a - kv^2$
- **Terminal velocity** $\sqrt{a/k}$: with $a = 1$, $k = 0.25$, 2 m/s. The physics gives the limit
- `halt` sets velocity and acceleration to exactly 0: `park` can ask for `(= (velocity) 0)` at any later time
- An event at a crossing turns a **narrow window into a stable state**

The drag plan

```
$ 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
```



With drag the velocity bends towards the terminal velocity.

- Two `brake`: release, then brake
- 6.4 s: first step at 10.03 m; `halt` at 7.97 s

Processes that interact: a greenhouse in winter

```
;; The plants die of frost below 10 degrees and burn above 30
(:constraint plants-alive
  :parameters ()
  :condition (and (>= (temperature) 10) (<= (temperature) 30)))
(: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))))))

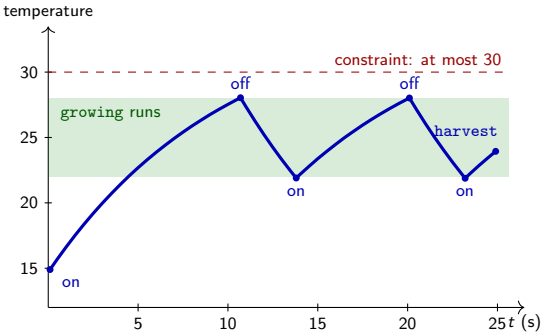
(:process growing
  :parameters ()
  :precondition (and (>= (temperature) 22)
                     (<= (temperature) 28))
  :effect (increase (growth) (* #t 1))))
```

- **Two processes, one quantity**: `heating` $+3$/s, `losing-heat` $-0.1(T-5)$/s; rates add up
- **A process changes its own rate**: the loss shrinks as T falls
- **Switched by a quantity**: `growing` runs only for $22 \leq T \leq 28$; the agent can only steer
- Constraint: $10 \leq T \leq 30$

Pencil first, then the plan: bang-bang

Heater always on: $0.1(T - 5) = 3$, $T = 35$.
Always off: towards 5. Both kill the plants.

```
$ kairos greenhouse-domain.pddl
    greenhouse-p01.pddl
; cost = 24.9
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]
```



Nobody told the planner to keep T in the band: growth happens only there.

A drone that lands: exclusive events

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

Rule

Events that can fire at the same instant and touch the same facts: make their conditions **exclusive**.

```
$ 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
```

- Free fall 1.1 s to -11 m/s; `touch-down` at 3.28 s, 1.58 m/s
- Pause at 2.0: the 0.1 s step is the search's only resolution

Events that cascade: dominoes

```
(:action push
  :parameters (?d - domino)
  :precondition (and (within-reach ?d) (not (fallen ?d)))
  :effect (fallen ?d))

;; A fallen domino knocks down the next one. The condition
;; (not (fallen ?b)) is what stops the event: it fires once
(:event topple
  :parameters (?a ?b - domino)
  :precondition (and (fallen ?a) (next ?a ?b)
                     (not (fallen ?b)))
  :effect (fallen ?b)))
```

```
$ kairos dominoes-domain.pddl
    dominoes-p01.pddl
; cost = 0
; makespan = 0.000
0.000: (push d1) [0.000]
0.000: @PlanEND
```

- A chain of events at one instant: a **cascade**
- Four events, no time passing; it stops because each event switches itself off and the row ends
- `dominoes-p02` (d2 down, d3 up): no plan. No instant for an action inside a cascade

In one line

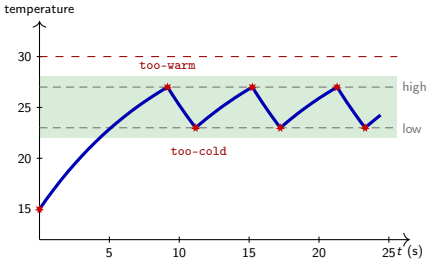
A cascade must end: each event switches itself off, and no event switches another back on.

Events as devices: the thermostat

```
;; The thermostat: what the agent decided by hand, now as events
(:event too-cold
  :parameters ()
  :precondition (and (thermostat-on) (not (heater-on))
                     (<= (temperature) (low)))
  :effect (heater-on))

(:event too-warm
  :parameters ()
  :precondition (and (thermostat-on) (heater-on)
                     (>= (temperature) (high)))
  :effect (not (heater-on))))
```

```
$ kairos greenhouse-auto-domain.pddl
    greenhouse-auto-p01.pddl
...
; better plan 2
0.000: (enable-thermostat) [0.000]
24.400: (harvest) [0.000]
```



- Seven events, off any grid
- 8 030 states instead of 432 899
- **Where the intelligence is**: a thermostat in the system? Model it

Events that never stop: the blinker

Wrong on purpose

```
(:event light-up
  :parameters ()
  :precondition (and (blinking) (not (lit)))
  :effect (lit))

(:event go-dark
  :parameters ()
  :precondition (and (blinking) (lit))
  :effect (not (lit))))
```

- Each event alone is correct; together they fire for ever at time 0
- `kairos check`: valid; planner: no plan; validator: `events keep firing at 0.000000 (go-dark)`

Repaired: time between the events

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

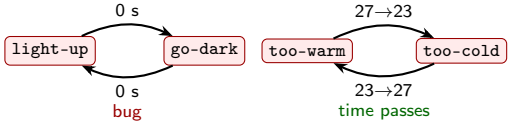
Kairos: `(switch-on)` at 0, `@PlanEND` at 2.5; the lamp lights at 0.5, 1.5, 2.5 s. Timer process + event that resets it: the idiom for anything periodic or delayed.

One threshold: Zeno behavior

- Thresholds 23 and 27: **hysteresis**, time must pass between events
- `greenhouse-auto-p02`: both at 25: infinitely many events around one instant
- Planner: `kairos: timeout (timeout)`, with 3 s or with 20
- Validator: accepts "enable and harvest at 40 s" (finite precision)
- No clear message: **slow or disagreeing tools**

Step D: the questions for a set of events

Draw an arrow from A to B if an effect of A can make the condition of B true. A chain that ends: a cascade, fine. A cycle: fine only if *time must pass* somewhere (hysteresis, a timer). A cycle in zero time is a bug.



Processes with durative actions and timed literals

Each construct for the change it owns:

- **durative action**: an activity of the agent that runs its course
- **process**: change that depends on the state
- **event**: a consequence the world draws
- **timed literal**: a change at a time on the clock

`courier-plus`: durative `move` marks `(driving ?r)`; the agent plugs and unplugs, the charger does the rest. On: -0.2/s; driving: -1; plugged: $+1.8$.

```
;; 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)))
```

The robot recharges

```
$ kairos courier-plus-domain.pddl
    courier-plus-p02.pddl
; cost = 95.3
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]
...
95.300: (drop r1 meds1 ward-a) [0.000]
```

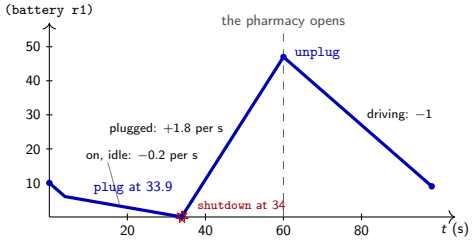
- p02: battery 20; 34 m to go
- At the dock with 4 units; plugged 53.3 s; unplugs at 99.9, one step before `battery-full`
- Needed 26 for the rest: valid and wasteful (the long wait)
- `(over all (powered ?r))`: a shutdown during a move breaks the plan
- 736 states: durations and timed literals at exact times

Waiting is not free

p03: battery 10, `(at 60 (is-open pharmacy))`

```
$ kairos courier-plus-domain.pddl
    courier-plus-p03.pddl
; cost = 98
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]
...
98.000: (drop r1 meds1 ward-a)
    [0.000]
```

A minute of waiting costs 12; the robot has 10. In Part IV it would just wait in the lobby.



- Plugs in with 0.02 left, one step before the shutdown; charges until 60 exactly
- Plugging on arrival is 10 s shorter; both spend the wait at the charger

One system, three models

Robot in the lobby, battery 20, medicines in the pharmacy, ward A 34 m away, a charger at the dock.

1. Numbers (ch. 8)

`courier-discrete`: a move subtracts distance × consumption; `charge` fills the battery.

Nine actions, found in a millisecond, proved optimal. No durations, least of all the charge.

2. Durative (ch. 12)

`courier-durative`: battery falls during a move, rises during a `charge` of chosen duration.

86.008 s; charges 42 s, the bound. A robot standing still spends nothing.

3. PDDL+

`courier-plus`: drain whenever on, charging is a state, empty battery is an event.

95.3 s; plugged 53.3 s.

What each level cannot say

Pharmacy opens in a minute, battery 10: level 1 cannot state the hour; level 2 waits in the lobby and delivers; only level 3 knows the wait empties the battery. Forty deliveries in a day: level 1 plans it, the other two, on most planners, will not.

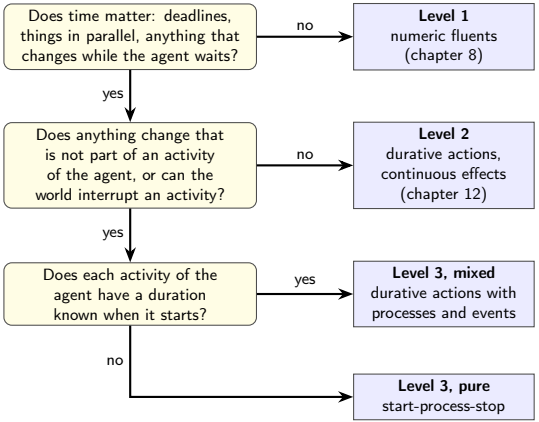
Alternatives compared: three levels

	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
Clarity	The simplest to read and to explain	One action per activity; plans are timelines	Laws separate from decisions; plans must be replayed
Portability	Any numeric planner: five of the eight of Appendix B	Continuous effects: two of the eight	Processes: one of the eight

Verdict

Choose the **lowest level that does not lie** about what the problem turns on. Within level 3, keep durative actions wherever level 2 had them.

Choosing the level of a model



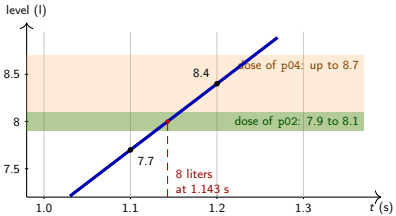
- **Level 1**: which actions and in which order; quantities are budgets
- **Level 2**: time is part of the question, every change blamed on an activity
- **Level 3**: something changes while the agent does nothing, or the world interrupts
- Each "yes" buys fidelity and pays in search effort and in planners that read the model

Limits: a window narrower than a step

doser: 8 l $\pm$ 0.1; p02 pumps 7 l/s

```
$ kairos doser-domain.pddl doser-p02.
    pddl
...
kairos: no plan found (exhausted)
$ kairos validate doser-domain.pddl
    doser-p02.pddl doser-p02-hand.plan
plan valid, makespan 1.143
```

- In the dose from 1.129 to 1.157 s: 29 ms; the search sees 7.7 l at 1.1 s, 8.4 l at 1.2
- Same symptom as a wrong model



The level is in the narrow dose between two states of the search (dots), 0.7 l apart.

An event does not rescue the search

- p03 arms the float switch at 8 l: valid in continuous time, yet `no plan found`
- The search fires events *between steps*: the switch closes at 1.2 s with 8.4 l
- The event makes the **world** precise, not the search
- p04 accepts up to 8.7 l (one step of inflow): the validator replays it, the switch closes at 1.143 s with 8 l

```
$ kairos doser-domain.pddl
    doser-p04.pddl
; cost = 1.2
; makespan = 1.200
0.000: (open-valve) [0.000]
1.200: @PlanEND
```

Rule for the modeler

Every window a plan must hit should be at least **the rate times 0.1 s** wide. Otherwise: does the story need it? Slow the process, or plan another way and validate.

The MODEL method: testing and choosing hybrid models

Step E for a hybrid model

1. **Pencil first**: where each quantity goes with each switch position
2. **Check every event**: a problem whose only plan passes through it; draw the arrows
3. **Bound the horizon**: small margins for "no plan" problems
4. **Validate in continuous time**: the plan you expect, off the 0.1 grid, and the plan you fear
5. **Replay and plot**: a free ride shows in a plot
6. **Measure the windows**: width vs rate $\times$ 0.1 s

Step	New questions
M	Must something never happen? Is a bad outcome forbidden, or does it only have consequences?
O	Each rate: a constant, another changing quantity, or the quantity itself?
D	Which processes touch the same quantity? Which events trigger which, and where must time pass? Which devices are events?
E	The six points on the left
L	The lowest level that does not lie

Common mistakes in hybrid models (1)

Two events that switch each other on

`blinker-wrong`: `kairos check` says valid, the planner says no plan, only the validator names it (`events keep firing`). The cause is the pair. Fix: time inside every cycle of events, a timer or two thresholds.

A thermostat with one threshold

A Zeno model; nothing reports it. Symptoms: a slow planner and tools that disagree. Fix: hysteresis, thresholds several steps apart.

A window narrower than a step

`no plan found (exhausted)` after 100 states although a plan exists. A missed window takes few states, a too strict model many. Confirm with a hand plan; widen to rate $\times$ 0.1 s.

Common mistakes in hybrid models (2)

Equality on a quantity that flows

`(= (level) 8)`: a window of width zero. The validator is exact: `(= (level) 8)`: `8.001 = 8 is false`. Fix: two inequalities, or an event that *assigns* the exact value (as `halt`).

Expecting the shortest plan

`courier-plus` charges to 99.9% and waits by the charger. In PDDL+ Kairos minimizes actions; long waits stop where something changes. Fix: pressure in the model (a deadline, a smaller battery) or improve by hand and validate.

A constraint where an event was needed, and the reverse

"Battery never empty" forbids rescue plans; a speed limit by event and goal fails when a problem forgets the goal. Impossible: a law of the world; forbidden: a constraint; costly: an event with consequences.

1 Processes and events

2 Modeling hybrid systems

3 Summary

What to take away

- PDDL+ separates what the agent **decides** (instantaneous actions) from what the world does: **processes** (continuous, while a condition holds) and **events** (instantaneous, when a condition becomes true)
- Actions control processes through facts: **start-process-stop**. An event must switch itself off; no cycle of events in zero time
- Plans list decisions and `@PlanEND`: replay them to read them
- Global constraints for what is forbidden; events for what has consequences
- Processes on one quantity add rates; change can be non-linear; mix with durative actions and timed literals
- Kairos: steps of 0.1 s, Runge-Kutta, validation in continuous time; `no plan found` (14) is not `unsolvable`. Windows $\geq$ rate $\times$ 0.1 s, no equality on flows
- The **lowest level that does not lie**; test for free rides

Exercises to try

Chapter 17

- 17.2.2: a tank with a leak
- 17.3.2: a float switch for the tank
- 17.4.3: actions, processes or events in an elevator
- 17.5.4: the three cold-chain plans
- 17.6.2: closing at 4.05 s
- 17.7.1: a coffee machine

Chapter 18

- 18.1.1: radar event vs constraint, five criteria
- 18.2.2: when to drink the coffee
- 18.4.1: arrows between events in four models
- 18.5.1: plug in on arrival
- 18.6.1: choose a level for four stories
- 18.7.1: the doser at 5 l/s