PDDL+

Dealing with Continuous Change and Exogenous Events

Matteo Cardellini

Università degli Studi di Genova

Motivating Example

lvl3

With the introduction of PDDL2.1, we saw how to manage temporal and lvl2 numerical planning in order to model domains much closer to the real world. However, **in real life, some actions produce a continuous change of the world and this change is most of the time non-linear**. These behaviours cannot be modelled with PDDL2.1 which is discrete and linear.

Physical events, like heating, charging a battery or accelerating, are classical examples of continuous changes which act upon a variable (i.e., temperature, charge and position) in a continuous and non-linear way.

Moreover, in PDDL2.1, all the changes of the world were under the strict control of the planner, which decided which actions to take at the right time. **Sometimes, instead, we would like to model phenomenons which are not controllable and must be simply dealt with** (i.e., boiling of water, the arrival of a new train at a station, the breaking of a component, etc).

Extending the robots domain

In the PDDL2.1 slides, we saw an example of the two robots which need to cooperate to move two balls from one room to another. Let's change the problem a little to model more realistic behaviours:

> *Wally, alone, has moved outside, and it is now simply tasked to move two balls from Garden A to Garden B. The gardens are kilometres away from each other, so now the battery has to be taken under consideration: Wally starts with a battery of 100%, and it decreases linearly while moving between the gardens. If the battery reaches 10% Wally can no longer move. He can use its solar panel to charge (non-linearly) whenever he needs, but cannot move while charging.*

this is an event

More Expressivity

We saw in the previous lections that is possible to model more and more expressive fragments of planning using the PDDL language

$$\textit{STRIPS} \subseteq \textit{PDDL} \equiv \text{PDDL2.1 LVL 1} \subseteq \text{PDDL2.1 LVL 2} \subseteq \text{PDDL2.1 LVL 3}$$

tutto quello che modello in strips posso modellarlo anche in pdll

We will now see an even more expressive language, called PDDL+ with

$$\text{PDDL 2.1 LVL3} \subseteq \text{PDDL+}$$

having

$$\begin{aligned} \text{PDDL+} &= \text{PDDL2.1 } \mathbf{\text{LVL 2}} + \text{Events} + \text{Processes} \\ &= \text{PDDL} + \text{Numbers} + \text{Events} + \text{Processes} \end{aligned}$$

PDDL+ Events

Events model a *happening* which is not under the control of the agent. The syntax of an event is the following:

```
(:event <event_name>
    :parameters (<arguments>)
    :precondition (<prop_or_numerical_expression>)
    :effect (<prop_or_numerical_expression>)
)
```

We saw that actions (and durative-actions) become *applicable* when their precondition are met, and then it is responsibility of the planner to chose *which* action to take next. Even if an action is always applicable, it is possible that this action would never appear in the final plan since it does not concur in reaching the goal. Events instead are applied as soon as its preconditions are respected, and their effects immediately change the state of the world.

PDDL+ Processes

Remember how with durative actions effects couldn't be expressed using `over all`.

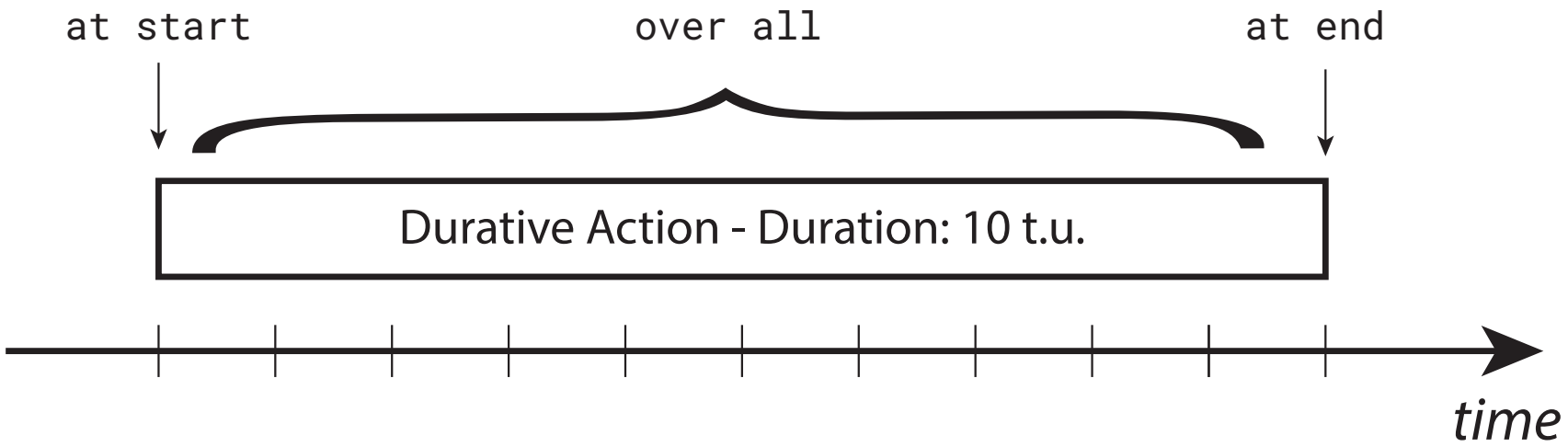


Figure: Representation of a durative action during 10 time units

PDDL+ Processes

Processes model a *continuos* change of a numerical variable. Like events, processes are not under the control of the planner. The syntax of a process is the following:

```
(:process <process_name>
    :parameters (<arguments>)
    :precondition (<prop_or_numerical_expression>)
    :effect (<numerical_expression>)
)
```

assegnazione, aumento , diminuzione

As it can be seen, effects of a process should only deal with numerical expressions. This is because, since processes model a continuous change, boolean predicates can not change more than once and would be useless to include them in the effect of processes.

PDDL+ Processes to model differential equations (I)

Processes are most used to express the amount of change of a quantity (i.e., its derivate). In our example, we will see that the battery of the robot discharges linearly and charges non-linearly with these equations:

$$\text{Discharge} \quad : \quad \frac{d\mathcal{B}(t)}{dt} = -2 \tag{1}$$

$$\text{Charge} \quad : \quad \frac{d\mathcal{B}(t)}{dt} = 0.2 \cdot (100 - \mathcal{B}(t)) \tag{2}$$

Where $\mathcal{B}(t)$ represents the amount of battery.

When discharging (1) the battery decrease linearly with a slope of -2. Instead, when charging (2), the battery increases with an amount which is proportional to the amount of battery left to charge, thus producing a non-linear effect in which the battery is charged more rapidly if the battery is low.

PDDL+ Processes to model differential equations (II)

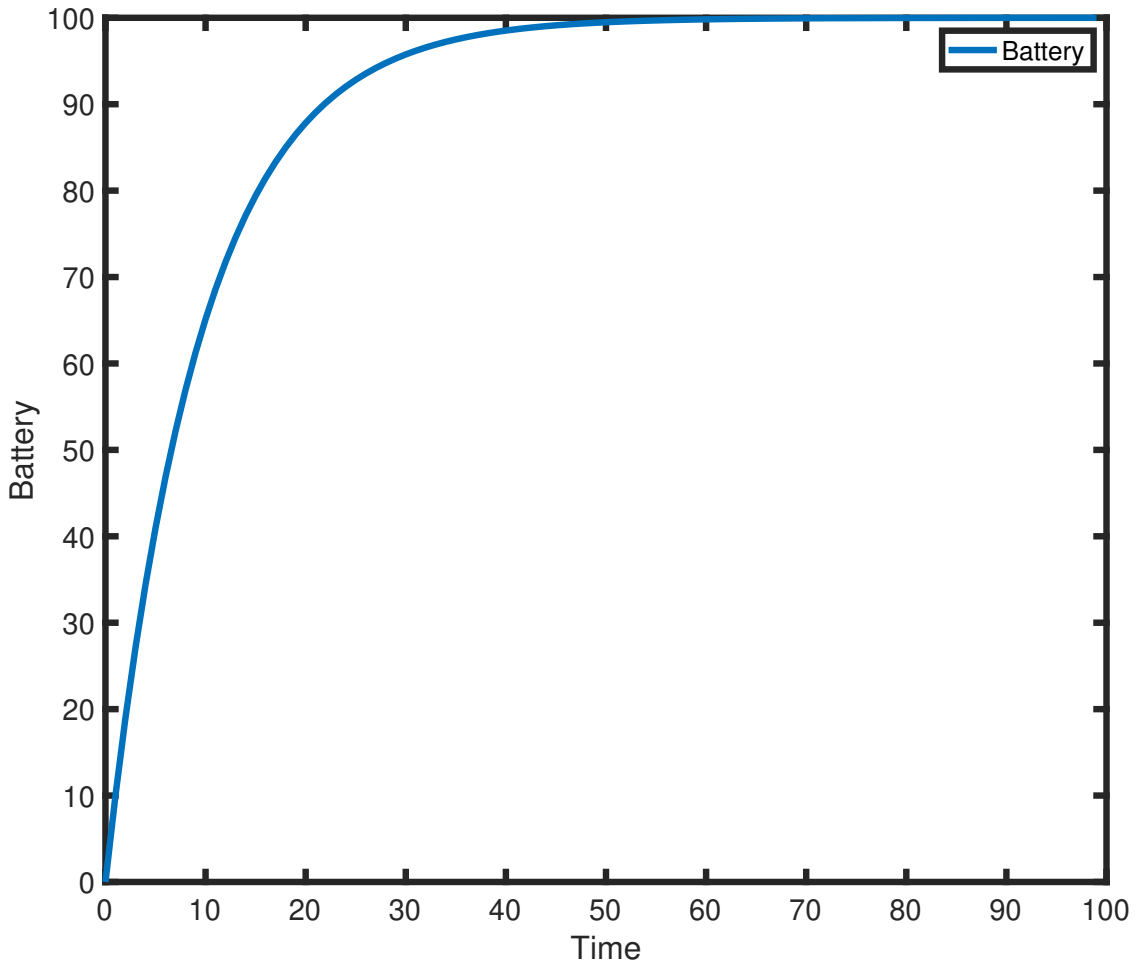


Figure: Nonlinear recharge of a battery over time, following Equation (2)

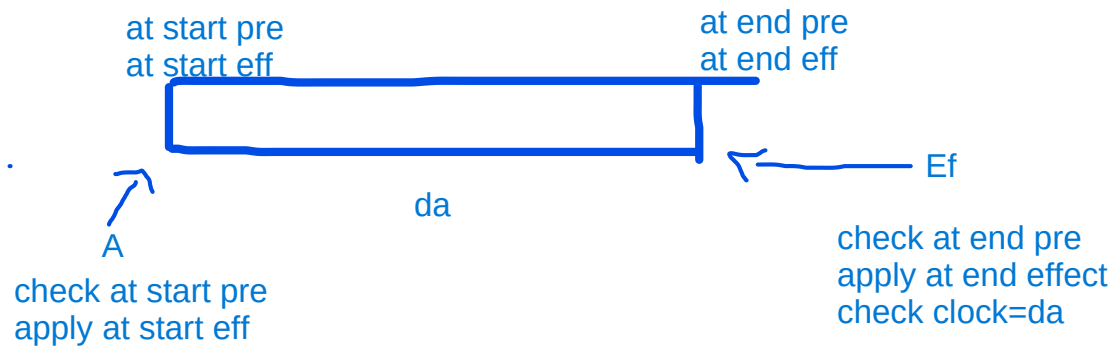
PDDL+ Processes to model differential equations (III)

These two differential equations can be modelled in PDDL+ with the following numerical expressions:

Discharge : `(decrease (battery ?r) (* 2 #t))`

Charge : `(increase (battery ?r) (*(*(- 100 (battery ?r)) 0.2) #t))`

Where `#t` is a reserved word and a syntactic device that marks the update as time-dependent. It can be considered the equivalent of *dt* in the differential Equations (1) and (2).



From PDDL2.1 durative-actions to PDDL+ processes and events

We saw in PDDL2.1 that actions with a duration could be modelled with the help of durative-actions. However, in our new domain, these type of actions would not allow us to model the possibility to pause the movement if the robot needs to charge mid-trip.

A durative-action can be replaced in PDDL+ by four components:

1. An action which gives the planner the possibility to **begin** the durative-action.
2. A process which models the **elapsed time** of the durative-action.
3. An event which signals the **end** of the durative-action when the elapsed time has reached the duration.
4. An event which signals the **failure** of the durative-action if the overall conditions have not been respected.

Transformation of durative-action

```
(:durative-action name
      :parameters (p⃗)
      :duration (= ?duration Dur[p⃗])
      :condition (and (at start Pre_S) (at end Pre_E) (over all Inv))
      :effect (and (at start Post_S)(at end Post_E[?duration]))
)
```

```
(:action name-start
     :parameters (p⃗)
     :precondition (and Pre_S (not (name_clock_started p⃗)))
     :effect (and Post_S
                  (name_clock_started p⃗)
                  (assign (name_clock p⃗) 0)
                  (assign (name_duration p⃗) Dur[p⃗])
                                          )

(:process name-process
     :parameters (p⃗)
     :precondition (name_clock_started p⃗)
     :effect (increase (name_clock p⃗) (* #t 1))
```

```
(:event name-failure
     :parameters (p⃗)
     :precondition (and (name_clock_started p⃗)
                  (not (= (name_clock p⃗) (name_duration p⃗)))
                  (not Inv))
     :effect (assign (name_clock p⃗) (+ (name_duration p⃗) 1)))

event
(:[redacted] name-end
     :parameters (p⃗)
     :precondition (and Pre_E (name_clock_started p⃗)
                  (= (name_clock p⃗) (name_duration p⃗)))
     :effect (and Post_E[(name_duration p⃗)]
                  (not (name_clock_started p⃗))
                                          )
```

Be careful of not *Inv* ! If *Inv* is $\wedge$ it has to be transformed in a $\vee$ (De Morgan's Rule).

Transformation of `move` durative-action

How can we model the durative-action move using processes and events?

```
(:durative-action move
    :parameters (?r - robot ?a - room ?b - room)
    :duration (= ?duration (move-time ?r))
    :condition (and
        (over all (allowed ?r ?b))
        (at start (at-robot ?r ?a))
        (at start (> (battery ?r) 20)))
    :effect (and
        (at start (not (at-robot ?r ?a)))
        (at end (at-robot ?r ?b))
        (at end (decrease (battery ?r) 20))))
```

Transformation of `move` `durative-action`: Action

Let's see now how the `durative-action` `move` can be transformed with the newly introduced PDDL+ constructs. First, let's create an action to begin the movement:

```
(:action startMove
    :parameters (?r - robot ?a - garden ?b - garden)
    :precondition (and  (at-robot ?r ?a)
                        (not (moving ?r)) (not (charging ?r)))
    :effect (and        (not (at-robot ?r ?a))
                        (moving ?r) (moving-path ?r ?a ?b)
                        (assign (moved-distance ?r) 0)))
```

The preconditions are the same ones that were specified in the DA with the option `at start` or `overall`. The predicate `path ?a ?b` now signals that there exists a path between the two gardens (to avoid autocycles). In the effects, instead, it can be noted the introduction of three predicates: (i) `moving ?r` signals that the robot has begun the movement, (ii) `moving-path ?r ?a ?b` states that the robot is moving between gardens `a` and `b`, and (iii) `moved-distance ?r` is initialized to 0 to allow counting the space run by the robot.

Transformation of `move durative-action`: Process

Now let's declare a process which will keep track of the progress of the movement:

```
(:process movingProcess
    :parameters(?r - robot ?a - garden ?b - garden)
    :precondition (and (moving ?r)(moving-path ?r ?a ?b)(allowed ?r ?b))
    :effect (and
        (increase (moved-distance ?r) (* (speed ?r) #t))
        (decrease (battery ?r) (* 2 #t))))
```

This process, as soon the predicate `moving ?r` is activated (by the action `startMove`), starts increasing the fluent `moved-distance` by the value $v \cdot dt$ where v is the speed of the robot. The battery instead starts to discharge with the rate expressed in Equation (1).

Transformation of `move durative-action`: Event

Finally, an event will signal that the robot as reached its destination.

```
(:event arrived
    :parameters(?r - robot ?a - garden ?b - garden)
    :precondition (and  (moving ?r) (moving-path ?r ?a ?b)
                        (= (moved-distance ?r) (distance ?a ?b)))
    :effect(and         (at-robot ?r ?b) (not (moving ?r))
                        (not (moving-path ?r ?a ?b))))
```

In the preconditions, it is stated that the event should be triggered when the moved distance of the robot is greater or equal than the distance between the two gardens. When this happens, the predicates which model the movement are shut off, implicitly stopping the process `movingProcess`.

Charging the robots

This is how in our domain we give the possibility to the robots to charge themselves:

```
(:event batteryDead
  :parameters(?r - robot)
  :precondition (and
    (<= (battery ?r) 10))
  :effect (and
    (not (moving ?r))))

(:action startCharge
  :parameters (?r - robot)
  :precondition (and
     (moving ?r)
     (<= (battery ?r) 50))
  :effect (and
    (not (moving ?r))
    (charging ?r)))

(:action stopCharge
  :parameters (?r - robot)
  :precondition (and
    (charging ?r))
  :effect (and
    (not (charging ?r))))
```

```
(:process charging
  :parameters(?r - robot)
  :precondition (and
    (charging ?r)
    (<= (battery ?r) 100))
  :effect (and
    (increase
      (battery ?r)
      (* (* (- 100 (battery ?r)) 0.2) #t))))


(:action repriseMovement
  :parameters (?r - robot ?a - garden ?b - garden)
  :precondition (and
    (moving-path ?r ?a ?b)
    (not (moving ?r))
    (not (charging ?r)))
  :effect (and
    (moving ?r)))
```

Problem file

This is the problem file for the PDDL+ robot files. Nothing here has changed w.r.t. the PDDL2.1 model:

```
(define (problem pb1)
  (:domain robot)
  (:objects
    gardenA - garden gardenB - garden
    ball1 - obj ball2 - obj
    wally - robot)
  (:init
    (at-robot wally gardenA)
    (free wally)
    (at-obj ball1 gardenA) (at-obj ball2 gardenA)
    (allowed wally gardenA) (allowed wally gardenB)
    (= (speed wally) 1)
    (= (battery wally) 100)
    (= (distance gardenA gardenB) 30)
    (= (distance gardenB gardenA) 30)
    (path gardenA gardenB) (path gardenB gardenA))

  (:goal (and (at-obj ball1 gardenB) (at-obj ball2 gardenB)))
)
```

Planning I

We are now able to run the planner to find a solution to the problem. We will use ENHSP which is the SoTA planner for PDDL+.

```
enhsp -o domain.pddl -f problem.pddl -delta 5

0,00000: (pick ball2 gardenA wally )
0,00000: (startMove wally gardenA gardenB )
(0,00000,30,00000)------>waiting
30,00000: (arrived wally gardenA gardenB )
30,00000: (drop ball2 gardenB wally )
30,00000: (startMove wally gardenB gardenA )
(30,00000,55,00000)------>waiting
55,00000: (startCharge wally )
(55,00000,60,00000)------>waiting
60,00000: (stopCharge wally )
60,00000: (repriseMovement wally gardenB gardenA )
(60,00000,65,00000)------>waiting
65,00000: (arrived wally gardenB gardenA )
65,00000: (pick ball1 gardenA wally )
65,00000: (startMove wally gardenA gardenB )
(65,00000,95,00000)------>waiting
95,00000: (arrived wally gardenA gardenB )
95,00000: (drop ball1 gardenB wally )
```

The command has been run with the option `--delta 5` to speed up the solving time and, as it can be seen in the plan, all the actions are performed only in timestamps multiple of 5.

Planning II

As it can be seen from the produced plan, we are able to correctly model the possibility for a robot to stop its movement in order to charge the battery. In fact, at timestamp 30, Wally starts its movement from `gardenB` back to `gardenA` but has to stop mid-trip in order to charge its batteries.

Since the plan represents a *recipe* of actions that the agents must take in order to move from the initial condition to the goal state, only actions are outputted in the plan. Events and processes are omitted (except for the special process `waiting`) since it is outside the control of the agents.

Optimize the plan I

In PDDL2.1 we saw the possibility to specify a metric in order to produce plans of increasing quality over time. In PDDL+ the `:metric` directive is still present, but many planners do not consider it. However, since in PDDL+ it is possible to deal directly with flowing variables, we can include an *optimisation constraint* directly in the goal state.

Let's suppose, for example, that we would like to search for a plan with a smaller make-span. We could then introduce another process like this:

```
(:process timePassing
  :parameters()
  :precondition ()
  :effect (increase  (time) (* #t 1)))
```

This process simply increase a fluent called `time` (which is not a reserved word and can be chosen freely) during the whole plan.

Optimize the plan II

After finding the first plan (with a make-span of 95) we could then rerun the planning problem with the following goal, asking the planner to find a plan of better quality.

```
(:goal (and (at-obj ball1 gardenB) (at-obj ball2 gardenB) (< time 95)))
```

Running it a second time, we notice that no other plan is found with a make-span less than 95, thus informing us that the plan is optimal.

The plan is indeed optimal with the option `-delta 5` but a better solution with a make span of 93 can be found specifying `-delta 3`. This shows us how the discretisation step can condition also the optimality of the problem

In this example we used the time as an example, but any other fluent can be used as a metric.

Additional infos and material

- The main paper about PDDL+ can be found here `https://www.jair.org/index.php/jair/article/view/10471`. In this paper, Fox and Long demonstrated how every durative-action can be rewritten as a PDDL+ action-process-event flow thus showing that PDDL+ is strictly more expressive than PDDL2.1.
- More infos about the ENHSP planner can be found here `https://sites.google.com/view/enhsp/home`.
- A compiled version of the planner, together with the PDDL+ instance files of the problem covered in these slides, can be found here: `https://github.com/matteocarde/unige-aai`.