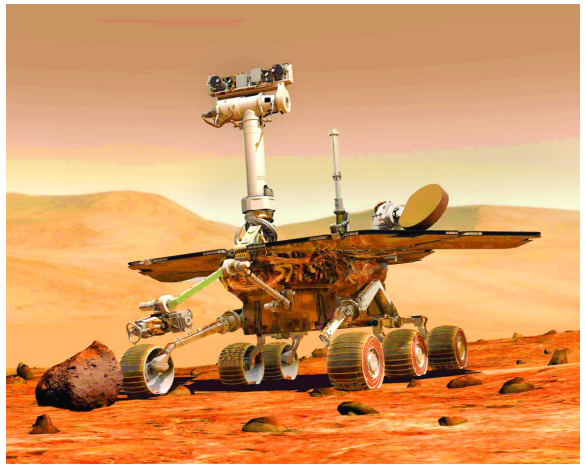
PDDL+

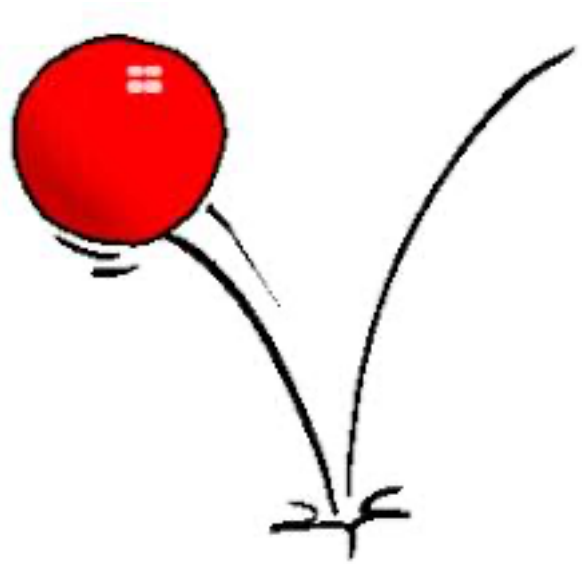
Why, what, how?

Some examples: how to in PDDL?



Rover (it is gone now, but we still remember it): battery recharges as soon as the sun hits the Rover

A ball that bounces on the floor



What is missing in PDDL?

- The ability to easily describe mixed discrete-continuous changes in the world (hybrid)
- Support for exogenous events
- (PDDL faces also issues in synchronisation of operations)

Answer: PDDL+

What's in PDDL+?

- The agent performs actions that change the world state in some way (the planner decides what actions to take)
- **Processes** execute continuously under the direction of the world–The world decides whether a process is active or not
- **Events** are the world's version of actions: the world performs them when conditions are met

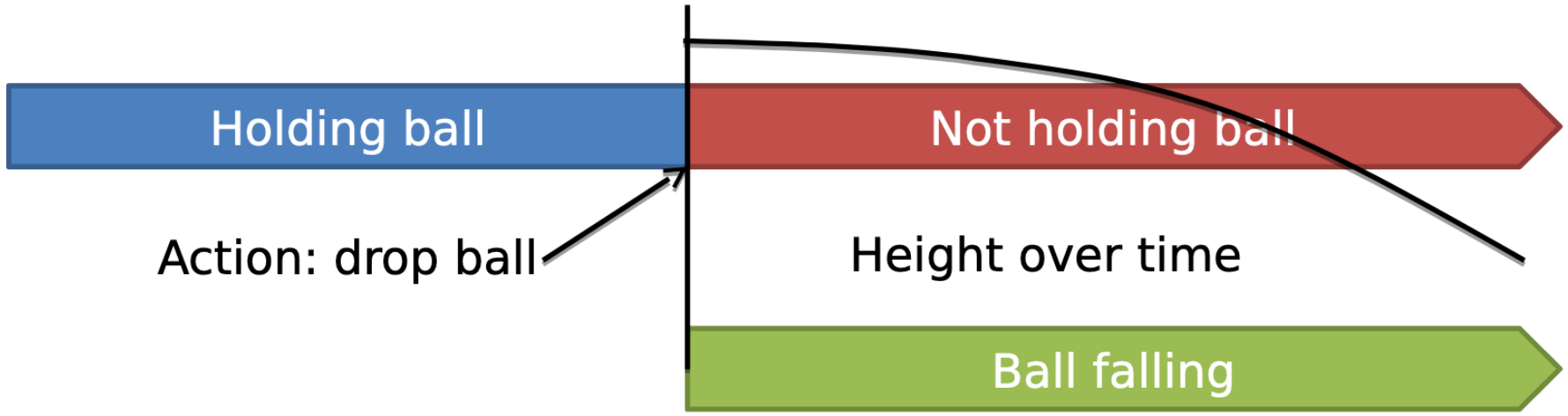
- Planning is about deciding which actions the agent should perform by anticipating the effects of actions before they are executed–Requires a way to predict what will happen when actions are performed (and also when they are not)

Processes

- Many of our interactions with the world involve us performing actions that initiate, terminate, control or simply avoid continuous processes
- Processes can represent flows: heat, energy, liquids, gases, traffic, money, information (continuous change), motion, growth and contraction

Is it really hybrid?

- When actions or events are performed they cause instantaneous changes in the world
  - These are discrete changes to the world state
  - When an action or an event has happened it is over
- Processes are continuous changes
  - Once they start they generate continuous updates in the world state
  - A process will run over time, changing the world at every instant



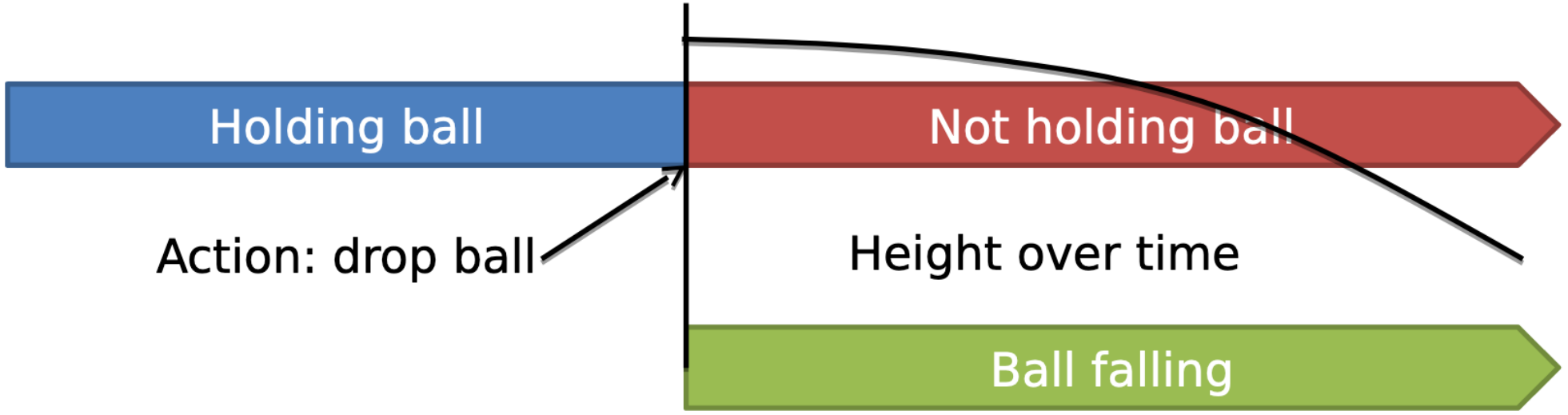
The ball example

- First drop it..
  - (:action release
    :parameters (?b –ball)
    :precondition (and (holding ?b) (= (velocity ?b) 0))
    :effect (and (not (holding ?b))))

- Then watch it fall…
  - (:process fall
    :parameters (?b –ball)
    :precondition (and (not (holding ?b)) (>= (height ?b) 0)))
    :effect (and (increase (velocity ?b) (* #t (gravity)))(decrease (height ?b) (* #t (velocity ?b)))))

Processes

```
(:process fall
:parameters (?b – ball)
:precondition (and (not (holding ?b)) (>= (height ?b) 0)))
:effect (and (increase (velocity ?b) (* #t (gravity)))
        (decrease (height ?b) (* #t (velocity ?b)))))
```



More on Processes

- Preconditions are to be considered as *over all* conditions
- More than one process can be active at any time
- The continuous effects of a process occour at all times the condition is satisfied.
- Processes cannot be used to model discrete changes!

- Bouncing
  - (:event bounce
    :parameters (?b -ball)
    :precondition (and (>= (velocity ?b) 0) (<= (height ?b) 0))
    :effect (and (assign (height ?b) (* -1 (height ?b)))(assign (velocity ?b) (* -1 (velocity ?b)))))

- Try to catch it!
  - (:action catch
    :parameters (?b -ball)
    :precondition (and (>= (height ?b) 5) (<= (height ?b) 5.01))
    :effect (and (holding ?b) (assign (velocity ?b) 0)))

Events

- Events are the world's version of actions: the world performs them when conditions are met

```
(:event bounce
:parameters (?b - ball)
:precondition (and (>= (velocity ?b) 0) (<= (height ?b) 0))
:effect (and (assign (height ?b) (* -1 (height ?b)))
             (assign (velocity ?b) (* -1 (velocity ?b)))))
```

More on Events

- Preconditions are like classical planning actions preconditions.
- Effects must be discrete
  - Effects are fired as soon as preconditions are met
- Effects must delete its own preconditions
  - This is to avoid multiple executions of the event
  - Usually better to avoid back to back events, as they are instantaneous. (Semantically, this can be tricky)

1.51421: **Event triggered!**
Triggered event (bounce b1)
Unactivated process (fall b1)
Updating **(height b1)** (-2.22045e-15) by 2.22045e-15 assignment.
Updating **(velocity b1)** (14.1421) by -14.1421 assignment.

1.51421: **Event triggered!**
Activated process (fall b1)

4.34264: Checking Happening... ...OK!
(height b1)$(t) = -5t^2 + 14.1421t + 2.22045e - 15$
(velocity b1)$(t) = 10t - 14.1421$
Updating **(height b1)** (2.22045e-15) by -2.44943e-15 for continuous update.
Updating **(velocity b1)** (-14.1421) by 14.1421 for continuous update.

4.34264: **Event triggered!**
Triggered event (bounce b1)
Unactivated process (fall b1)
Updating **(height b1)** (-2.44943e-15) by 2.44943e-15 assignment.
Updating **(velocity b1)** (14.1421) by -14.1421 assignment.

4.34264: **Event triggered!**
Activated process (fall b1)

4.757: Checking Happening... ...OK!
(height b1)$(t) = -5t^2 + 14.1421t + 2.44943e - 15$

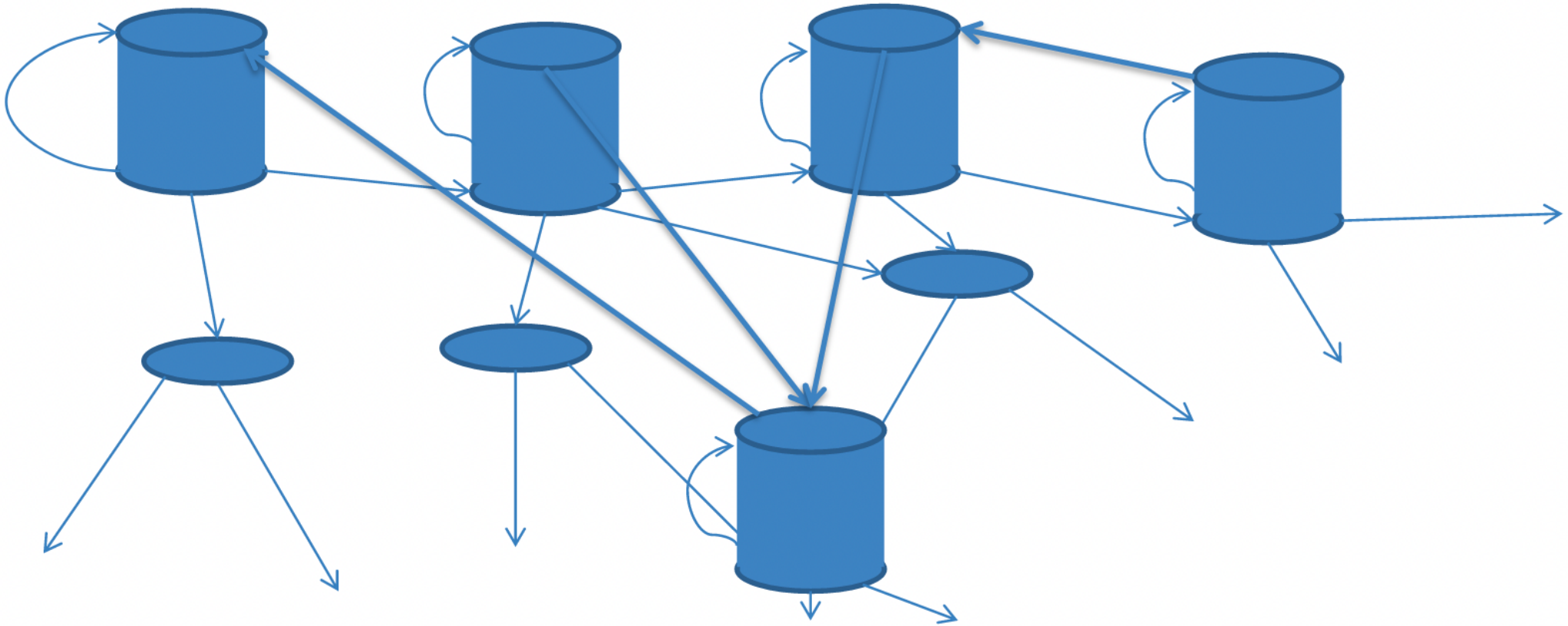
Updating **(height b1)** (2.44943e-15) by 5.00146 for continuous update.
Updating **(velocity b1)** (-14.1421) by -9.99854 for continuous update.

4.757: Checking Happening... ...OK!
Adding (holding b1)
Updating **(velocity b1)** (-9.99854) by 0 assignment.

4.757: **Event triggered!**
Unactivated process (fall b1)

Example Validation

- Processes mean time spent in states matters



Another Example

- Vertical Take-Off Domain
- The V-22 Osprey aircraft takes off vertically and needs to reach a location where stable fixed-wind flight can be achieved. The aircraft has fans/rotors which generate lift and which can be tilted by 90 degrees to achieve the right velocity both vertically and horizontally.

- (:action start_engines
  :parameters ()
  :precondition (and (not (ascending)) (not (crashed)) (= (altitude) 0))
  :effect (ascending))
- (:process ascent
  :parameters ()
  :precondition (and (not (crashed)) (ascending))
  :effect (and (increase (altitude) (* #t (-(* (v_fan) (-1 (/ (* (* (angle) 0.0174533) (* (angle) 0.0174533)) 2))) (g))))(increase (distance) (* #t (* (v_fan) (/ (* (* 4 (angle)) (-180 (angle))) (-40500 (* (angle) (-180 (angle))))))))))

- (:durative-action increase_angle
  :parameters ()
  :duration (<= ?duration (-90 (angle)))
  :condition (and (over all (ascending)) (over all (<= (angle) 90)) (over all (>= (angle) 0)))
  :effect (and (increase (angle) (* #t 1))))
- (:event crash
  :parameters ()
  :precondition (and (< (altitude) 0))
  :effect ((crashed)))
- (:process wind
  :parameters ()
  :precondition (and (not (crashed)) (ascending))
  :effect (and (increase (altitude) (* #t (wind_y) 1) (increase (distance) (* #t (wind_x) 1))

How to model in PDDL+?

- The main focus is usually on processes; they drive the changes of the world.
- Actions are under the control of the planner, that can use them to start processes.
- Events are used to define “limits”, and to automatically start/stop processes.

- We move away from a more traditional agent-based planning, to a control theory approach

Planning engines

What about planners?

PDDL+ is the most expressive language of the PDDL family, and the modelled problems can be extremely challenging.

- Not many planners available

- Two main approaches
  - Search on the PDDL+ problem
  - Translate the PDDL+ problem into a different problem

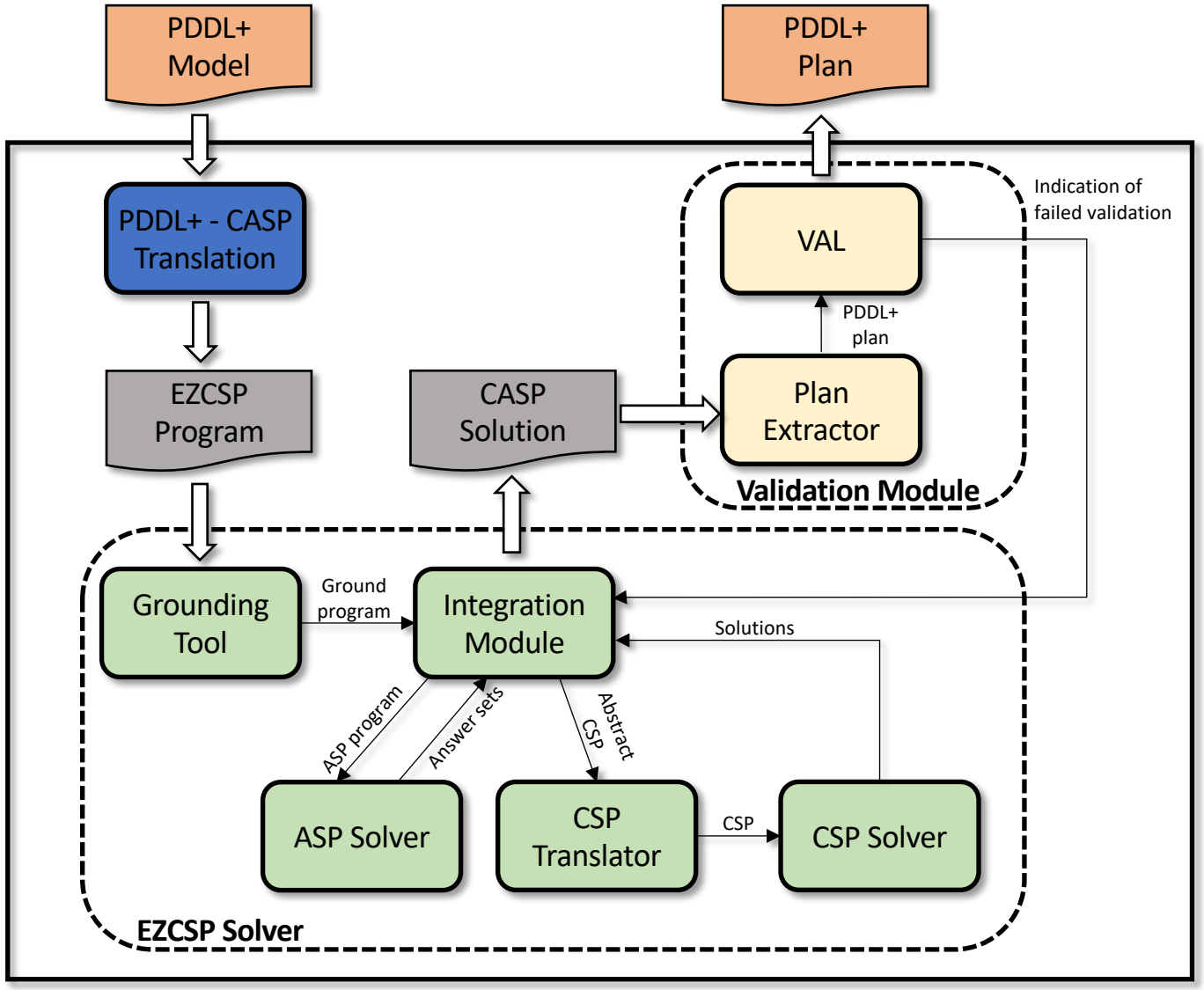
Planners

- ENHSP (https://sites.google.com/view/enhsp)
- SMTPlan (http://kcl-planning.github.io/SMTPlan/)
- DiNo (https://github.com/KCL-Planning/DiNo)
- CASP
- SMTplan and DiNo are a bit cumbersome to compile and run

DiNo and UPMurphi

- Based on model checking
- They generate a state system, that can be navigated using the original concepts introduced in the PDDL+ model
- Upmurphi then uses model checking, to blindly explore the state system and identify a trajectory
- DiNo relies on a heuristic based on the notion of staged relaxed plannning graph

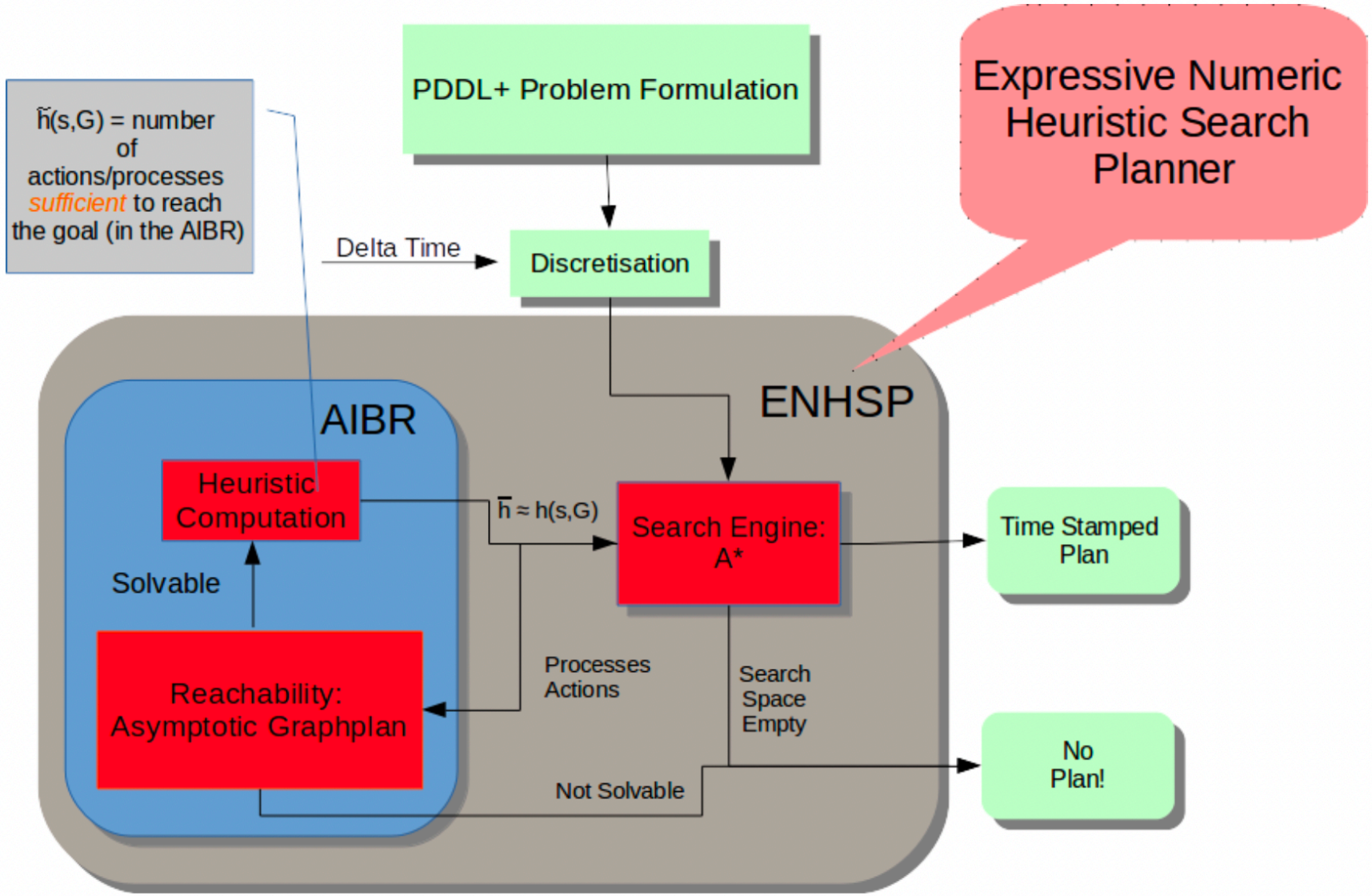
CASP


PDDL+
Model
PDDL+
Plan
PDDL+ - CASP
Translation
VAL
Indication of
failed validation
PDDL+
plan
EZCSP
Program
CASP
Solution
Plan
Extractor
Validation Module
Grounding
Tool
Ground
program
Integration
Module
Solutions
ASP program
Answer sets
Abstract
CSP
ASP Solver
CSP
Translator
CSP
CSP Solver
EZCSP Solver

ENHSP

The ENHSP Planning system

- Planning with autonomous processes
  - Classical Planning
  - Numeric Planning
  - Autonomous Processes (with a fixed delta_t, easy integrator)
  - Global Constraints over numeric and propositional variables
- Technologies used
  - Search engines: GBFS, WASTAR, EHC,
  - Heuristics: AIBR, Subgoaling based heuristic (extension of h^1 for the numeric case)..others…

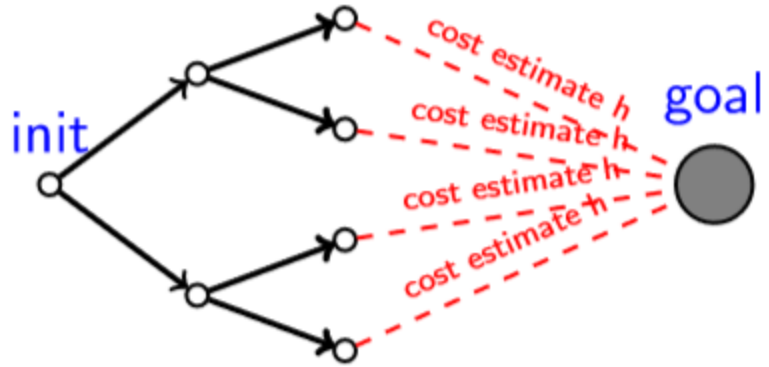

PDDL+ Problem Formulation
Expressive Numeric Heuristic Search Planner
h̃(s,G) = number of actions/processes sufficient to reach the goal (in the AIBR)
Delta Time
Discretisation
ENHSP
AIBR
Heuristic Computation
h̄ ≈ h(s,G)
Search Engine: A*
Time Stamped Plan
Solvable
Processes Actions
Search Space Empty
Reachability: Asymptotic Graphplan
No Plan!
Not Solvable

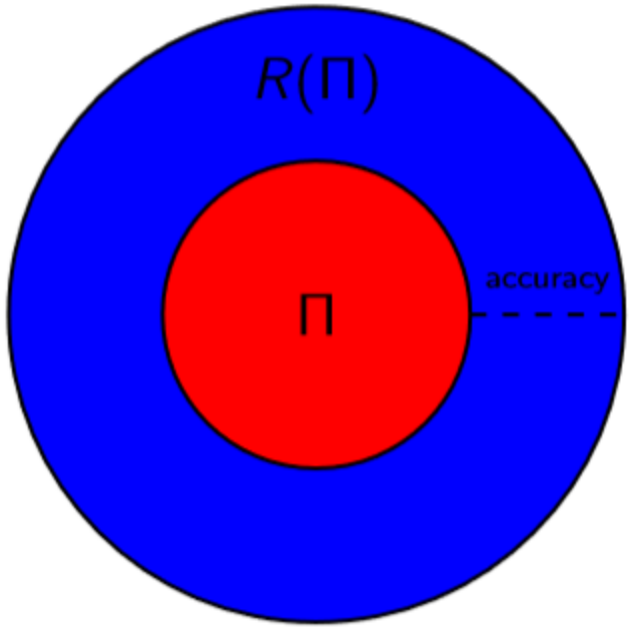
Search Space: an example

Issues…

- Number of plans is infinite as the state space is infinite
- You need guidance! And a good one..

Heuristic based on relaxation/abstraction





- **Basic Idea**: You have a problem Π, define a relaxation of Π, called Π', solve Π' to guide the resolution of Π
- State of the art classical planner uses this as a fundamental principle to devise heuristics (and not only)

Additive Interval Based Relaxation*: key ideas (details in Scala et al. ECAI 2016 paper)

1. Processes as Actions Relaxation
2. Redefine semantics of the problem using Interval Analysis
   1. State is not punctual but a set of possible values; each variable is an interval
   2. Action changes variable using convex union
3. Relaxed satisfiability. Goals evaluated w.r.t. intervals. Conditions evaluated in separation
4. Reachability Plus Action Counting
   1. Reachability analysis using asymptotic relaxed planning graph
   2. Estimate extracted counting number of actions

Results

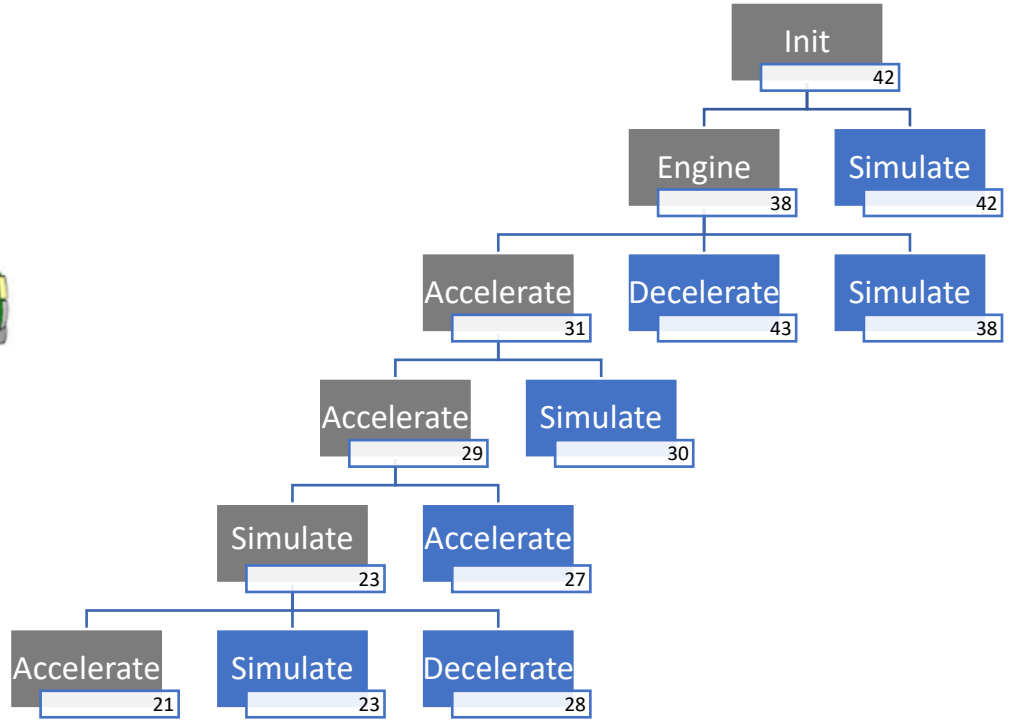
Computationally
1. Decidable Proper Relaxation
2. Polynomial Algorithm

Class of Problems
1) All numeric planning with computable functions
2) Sequential numeric planning, Processes, classical planning…

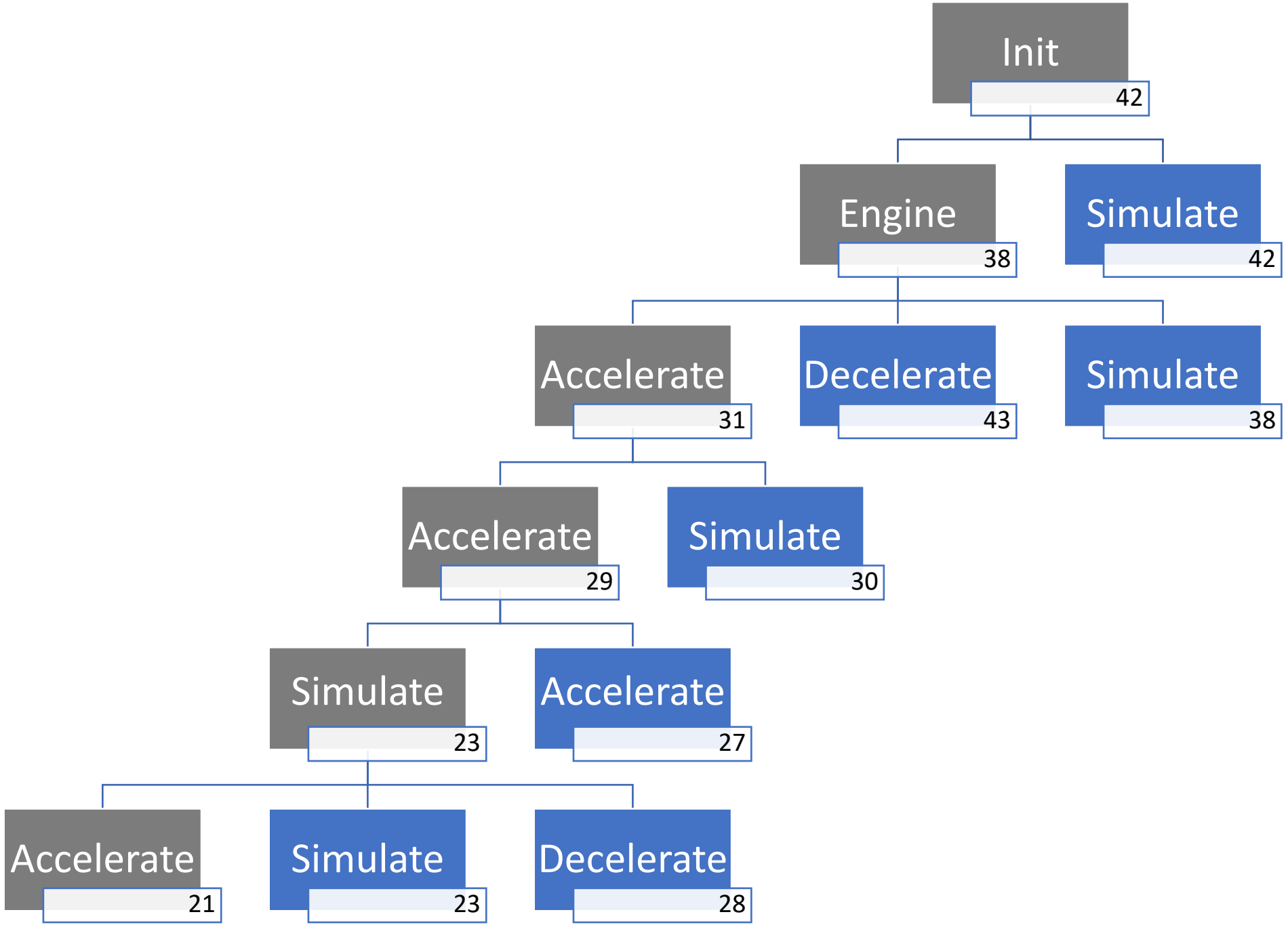
*Interval Based Relaxation was introduced in Metric-FF (Hoffman, 2003), then revisited for acyclic tasks by Aldinger et al. 2015.
In our paper we extend it to unrestricted numeric planning problem, plus processes

Car Example. The Search Space Explored

```
(:action accelerate
    :precondition (running)
    :effect (increase (a) 1)
)
(:action decelerate
    :precondition (running)
    :effect (decrease (a) 1)
)
(:process moving
   :precondition (running)
   :effect (and (increase (d) (* #t (v)))
                (increase (v) (* #t (a))))
)
```



Start in position x = 0, arrive in position x = [30,31], with speed = 0, motor stopped, domain of possible acceleration from -8,8


Init
42
Engine
38
Simulate
42
Accelerate
31
Decelerate
43
Simulate
38
Accelerate
29
Simulate
30
Simulate
23
Accelerate
27
Accelerate
21
Simulate
23
Decelerate
28

Something more…

The issue of delta (2)

- Discretisation is done in current PDDL+ planners.

- 2 types of discretisation (delta) to take into account
  - Planning delta
  - World evolution delta

The deltas

- Planning delta:
  - How often the planner can decide what to do
  - As large as possible, to reduce the complexity
- World evolution delta:
  - How often the exogenous processes and events are checked, and their effects applied to the world.
  - As small as possible, to ensure validity.
- Both of them have to be carefully selected, and on a domain-by-domain basis.

Expressivity

- PDDL+ is strictly more expressive than PDDL2.1

- Validation, in absence of discretisation, can be particularly challenging (worst case, undecidable) when cascading events are involved – there comes the issue of ordering
  - How to order events that should happen concurrently?
  - What if one event de-activate another one?

Do we really need PDDL+?

- In some works it has been shown that PDDL2.1 can represent what is done in PDDL+ with processes and events.
- PDDL+ Planning with Events and Linear Processes, *Amanda Jane Coles, Andrew Ian Coles, ICAPS 2014*
- Translations from Discretised PDDL+ to Numeric Planning, *Francesco Percassi, Enrico Scala, Mauro Vallati, ICAPS 2021*

We introduce a simple illustrative example problem based on the use of a mobile phone. The scenario is as follows: a person initially in the countryside with his phone switched off must go to the city and make a call from there (i.e. the goal is *called*[2]). The domain has three durative actions:

- **travel**: dur = 15; $\text{pre}_{\vdash}$ = {at country}; $\text{eff}^{-}_{\vdash}$ = {at country}; $\text{eff}^{+}_{\dashv}$ = {at city}; $\text{eff}_{\leftrightarrow} = \{d(signal)/dt = 0.5\}$;
- **turn_on**: dur > 0; $\text{pre}_{\vdash}$ = {$\neg$on}; $\text{pre}_{\leftrightarrow}$ = {battery>0}; $\text{eff}^{+}_{\vdash}$ = {on}; $\text{eff}^{-}_{\dashv}$ = {on}; $\text{eff}_{\leftrightarrow} = d(battery)/dt = -1$
- **call**: dur=1; $\text{pre}_{\vdash}$={at city $\wedge$ battery > 1}; $\text{eff}^{+}_{\dashv}$={called};

There is also a process, which models the transfer of data over the network at a fixed rate, if certain conditions are met:

- **transfer**: $\text{pre}_{\leftrightarrow} = \{\text{on} \wedge \text{battery} > 10 \wedge \text{signal} > 5\}$; $\text{eff}_{\leftrightarrow} = d(data)/dt = 1$.

Finally, an event models a low-battery warning:

- **warning**: pre={$\neg$warned$\wedge$battery< 8}; eff^{+}={warned}.

Processes and events

actions. First, we observe that, at any time, each process p_i (with precondition C_i and effects $\text{eff}_{\leftrightarrow}p_i$) is either executing, or not; i.e. either C_i or $\neg C_i$. We might therefore consider two durative-actions for p_i, rather than one:

- run_p_i, with $\text{pre}_{\leftrightarrow}\text{run\_}p_i = C_i$, and $\text{eff}_{\leftrightarrow}\text{run\_}p_i = \text{eff}_{\leftrightarrow}p_i$;
- not-run_p_i, with $\text{pre}_{\leftrightarrow}\text{not-run\_}p_i = \neg C_i$, and no effects;
- in both cases, the duration of the action is in $[\epsilon, \infty]$.

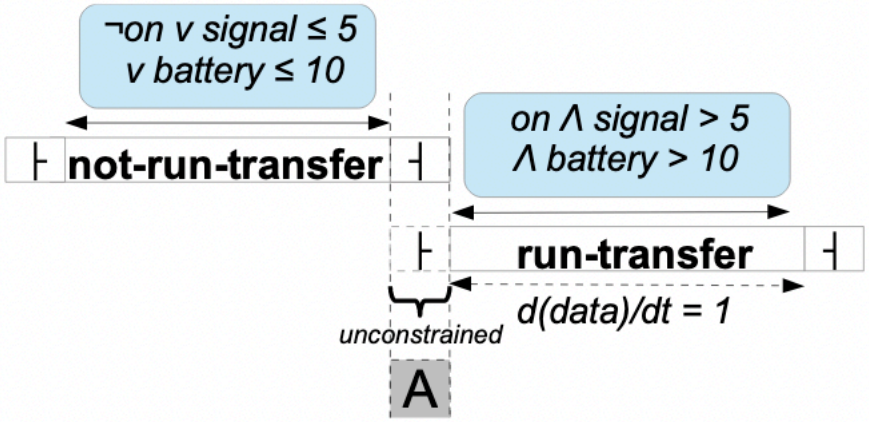
If we could ensure that we only ever apply run_$p_{i\dashv}$ (the end of run_p_i) if we simultaneously apply not-run_$p_{i\vdash}$ – and vice-versa – then the behaviour of the process has been simulated with actions. In other words, the start of one action must be always synchronised with the end of the other. Whenever the truth value of C_i changes (which may be many times) we simply switch which of these actions is executing. Ensuring this switch happens simultaneously is crucial: if time passed between e.g. not-run_$p_{i\dashv}$ and run_$p_{i\vdash}$ then there would be time when C_i might be true, but the effect of p_i is not being captured by any executing action.

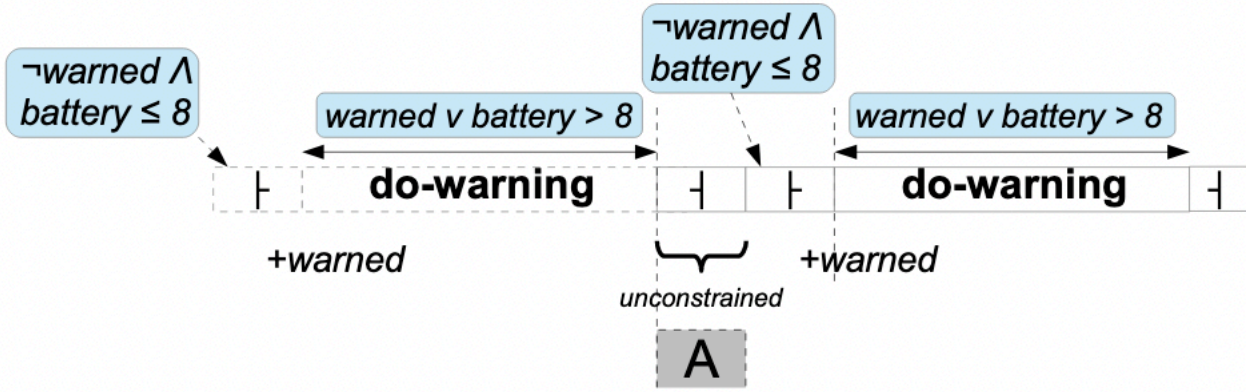
We also observe, that at any point each event e_j with precondition C_j and effects $\text{eff}\, e_j$, it is either happening at that time, instantaneously; or its conditions are false. Precisely:

- When the event occurs, C_j is true – and, as noted earlier, events must delete one of their own preconditions;
- A period then begins in which $\neg C_j$ holds;
- If the event re-occurs, then beforehand there is an ϵ-long period in which neither $\neg C_j$ nor C_j: the conditions are mutually exclusive with any discrete effects that led to C_j.

We can capture this using a durative action do-e_j, that must restart *immediately after* it ends, i.e. meet with itself. $\text{pre}_{\vdash}\text{do-}e_j = \text{pre}\, e_j$ and $\text{eff}_{\vdash}\text{do-}e_j = \text{eff}\, e_j$ reflect that e_j occurs at its start. Then, $\text{pre}_{\leftrightarrow}\text{do-}e_j = \neg C_j$ enforces the subsequent period in which e_j does not fire. Finally, the absence of effects/conditions at the end gives an ϵ gap in which a snap-action can change the facts/variables in C_j. Defined thus, do-e_j must be restarted at exactly the times e_j would occur.

Finally..





Reasoning with processes and events using PDDL2.1 requires a compilation that enforces three conditions:

1. Clipping together of process/event actions (as Figure 1);
2. Ensuring that not-run_p_i (or run_p_i), and a variant of do_e_j, start *immediately*, at the start of the plan;
3. Allowing processes/event actions to end in goal states.

References

- Introduction and formal description of PDDL+
  - https://jair.org/index.php/jair/article/view/10471

- AIBR description: Scala et al 2016
- CASP: link
- DiNo: link
- SMTPlan+: link
- Use of PDDL+ in battery management: link