PDDL+ level 5: An Extension to PDDL2.1 for Modelling Planning Domains with Continuous Time-dependent Effects

Maria Fox and Derek Long
University of Durham
South Road, Durham, UK
emails: maria.fox@dur.ac.uk and d.p.long@dur.ac.uk

September 5, 2001

1 Introduction

This document describes PDDL+ level 5, a significant extension of PDDL2.1 intended to support the representation of deterministic real time problem domains involving numeric-valued resources. PDDL2.1 is the agreed standard to be used in the AIPS 2002 planning competition. It comprises modifications to McDermott's PDDL (referred to as levels 1 and 2) and two additional levels constituting significant extensions. These levels are described in an accompanying document, the PDDL2.1 document [2]. As is discussed there, levels 3 and 4 introduce durative actions and allow continuous time to be modelled under certain restricting assumptions. Level 5 removes these assumptions and is powerful enough to model domains containing both discrete and continuous behaviours.

We use the name PDDL+ to refer collectively to PDDL2.1 and level 5. The key extension that level 5 provides is the ability to express durative action in terms of the initiation and termination of *processes* that act on the numeric components of states. Processes run over time and have a continuous effect on numeric values. Logical state changes are effected by the instantaneous initiation and termination activities marking the end points of active processes. We believe that this approach overcomes some important philosophical problems, discussed in the PDDL2.1 document, raised by durative actions.

In this paper we provide a detailed discussion of the features of level 5 of PDDL+, and the reasons for their addition, and a formal semantics in terms of a formal mapping from level 5 to hybrid automata. In the PDDL2.1 document we discussed aspects of the informal semantics of level 4 – we now show how a formal semantics can be given to level 4 by a mapping from level 4 to level 5. We consider the complexity of planning with the highly expressive language of level 5 and note that, under certain constraints, the Plan Existence problem (which corresponds to the Reachability problem for the corresponding hybrid automaton) remains decidable. We describe what it means for a proposed plan at level 5 to be valid and discuss the question of how model-checking techniques suited to hybrid automata might be exploited in the design of planners able to handle full PDDL+.

This paper should be seen as a continuation of the PDDL2.1 document. All of the definitions given in PDDL2.1 hold, and will be referred to here. In the PDDL2.1 document we provide the objectives and ground rules governing the extensions we have incorporated in PDDL+.

2 Modelling Continuous Time

In PDDL+ one of the numerically varying quantities is time. The objective is to be able to model continuous processes as these arise in planning problems. An agent must be able to interact with continuously changing quantities and to plan with up-to-date information about their values. The model of time in level 5 is therefore real-valued and continuous.

Several researchers have addressed the problem of modelling time and numeric quantities in a discretized way [1, 7]. The basic idea is to add a value to an action specification indicating the duration of that action. For example, the action of driving between two cities would have associated with it a duration of, say, two hours, at the end of which the fuel level of the vehicle would be updated to have been reduced by an amount determined by the rate of fuel consumption and the distance travelled. We adopted a variation on this approach in PDDL2.1 level 3. In some ways this is an intuitively appealing way of modelling change – the continuous aspects of change are abstracted away leaving a model of the key points at which episodes of change are completed. However, there are some important flaws in this approach.

The first flaw is that the unavailability of information about the values of a numeric variable, during the interval of a durative action that updates it, means that actions that refer to the same variable cannot be applied in parallel. This is the case even if such actions only consult the variable and do not update it themselves.

The second flaw is that, if an action is associated with a duration then the action will have its effect at the end of that duration without further intervention by the planner. This prevents the modelling of effects that occur as the consequence of an action initiating some process that then needs to be explicitly terminated. For example, the action of boiling a pan of water has to be explicitly terminated or else the pan will burn dry. Modelling this action as having a duration of, say, five minutes, means that the water will reach boiling point after five minutes and then remain in this condition until some other action affects its state.

The following example illustrates some additional difficulties with this discretized approach to the modelling of time. Consider a simple world in which a vehicle can move between (equi-distant) locations, consuming fuel as it does so; this action takes 10 time units. The vehicle can traverse a turret in one time unit, consuming fuel to do so, and it can also jettison its external fuel tank once the fuel level has dropped to half capacity. This also takes one time unit. A final action available is to rescue a trailer. This action (requiring two time units) needs the vehicle to be heavy enough to pull the trailer out of mud and it must therefore have at least a quarter load of fuel, even though it does not consume extra fuel in carrying out the action. It seems reasonable that a plan for this domain should allow a vehicle to move between locations, traversing its turret, rescuing a trailer and jettisoning its external fuel tank all in parallel, provided that the vehicle has enough (but not too much) fuel to do so. With naive durative actions this is not easy to achieve, since there is ambiguity about the state during the period of execution of an action. In particular, if the precondition of drive, that the fuel be sufficient for the journey, is interpreted as being true throughout the interval then the vehicle cannot jettison its tank until it arrives. This makes the plan 11 time units long instead of 10. On the other hand, if the precondition is only required to be true at the start of the interval then the vehicle could apparently traverse its turret several times during the period, so that the effects of the drive action will leave the vehicle with negative fuel, which is clearly unintended and undesirable. If the effect of the drive action on the fuel is executed immediately, with only the logical effect postponed until the end of the interval then the rescue of a trailer is ruled out during the drive. Again, this will lead to a longer plan than seems necessary. The problem is caused by the fact that there are actions that depend on the intermediate values of the fuel level during execution of the drive, and this simple model fails to represent that value.

This harmful interaction can be identified, although to a lesser extent, in logical pre- and post-conditions as well as with numeric ones, leading us to the conclusion that these simple durative actions cannot be given an adequate semantics to allow us to represent dynamic domains effectively.

These examples led us to distinguish between the discrete states of the domain and the underlying continuous processes that affect its numeric variables. At any point the domain will be in a given state in which time passes and numeric quantities change (according to which processes are active). The domain changes state both when an agent acts upon it, by changing its logical configuration, or when the dynamics of the domain bring about a spontaneous state change. In these terms, a domain is seen as a mixed discrete-continuous system requiring a hybrid model.

To achieve such a model we introduced two additional modelling components to supplement the notion of action. In PDDL+ level 5, as in PDDL and its precedents, actions have instantaneous effects that result in a change in the logical state of the domain. Their effects are deterministic and the planner has perfect knowledge about their outcomes – indeed, when grounded (to remove quantifiers and conditional effects), the semantics of their logical pre- and post-conditions are taken directly from Lifschitz' original STRIPS semantics [6]. Actions also have numeric pre-and post-conditions which both consult and modify numeric variables.

In level 5 actions can initiate *processes* that then run over time. A process maintains the logical aspects of a state whilst modifying numeric aspects of the state over time. The important distinction between actions and processes is that actions cause state transitions whilst processes do not. *Events* are also instantaneous state transition functions which can have numeric pre-and post-conditions. They are distinguished from actions only by the fact that the domain designer labels them *event* to prevent the planner from selecting them to develop a plan. Processes and events are distinguished by the fact that the numeric post-conditions of events cannot be time-dependent whilst those of a process always are. A process can be terminated by an action, or by some event that occurs in the world (such as the pan burning dry, or the vehicle running out of fuel). The introduction of processes and events means that domain features that might be modelled by actions with duration are modelled, in level 5, by a three-part structure consisting of a point of initiation, a process and a point of termination. The points of initiation and termination can be the points of application of actions or events or they can be the points at which the effects of active processes cause numeric values to reach critical thresholds. We refer to this as the *start-process-stop* model. Figure 1 shows part of the start-process-stop model of filling a bath. Appendix C gives the full PDDL+ description of this domain. As can be seen in this fragment, process effects are described in terms of the effect on continuous variables of the passage of time. The special literal $\#t$ is used to denote the time period over which a process instance runs. That is, each separate continuous period over which the preconditions of the process are satisfied will have a duration whose value is denoted by $\#t$ for that process instance.

The start-process-stop model is very flexible, because it allows the representation of very detailed aspects of the processes being modelled - for example, the rate of water evaporation over time - but it also allows these details to be completely abstracted out and the passage of

```
(:action turn_on
        :parameters
                (?tap ?b)
        :precondition
                (and (off ?tap ?b)
                     (plug_in ?b))
        :effect
                (and (not (off ?tap ?b))
                     (on ?tap ?b)
                     (increase (flow ?b) (flow-rate ?tap)))
)

(:process bath_filling
        :parameters
                (?b)
        :precondition
                (<= (level ?b) (capacity ?b))
                (> (flow ?b) 0))
        :effect (increase (level ?b) (* #t (flow ?b)))
)

(:event flood
        :parameters
                (?b)
        :precondition
                (and (>= (level ?b) (capacity ?b))
                     (> (flow ?b) 0)
                     (dry_floor ?b))
        :effect
                (and (wet_floor ?b)
                     (not (dry_floor ?b)))
)
```

Figure 1: A fragment of the PDDL+ bath-filling domain, showing an action, process and event.

time only to be modelled. For example, *walking* is an action that is naturally seen as having duration, and walking between locations A and B can be modelled by a start-process-stop triple that abstracts every detail other than that the walker arrives at B within a certain time of leaving A. The only numeric variable one is likely to be interested in including in an encoding of the walking process is the passage of time. In level 5, walking can be modelled as the planner initiating, by setting off on the walk, a process of (walking-)time passing. This process is terminated by an arrive event which becomes applicable as soon as the amount of (walking-)time required, to reach B from A, has passed.

Note that neither events nor processes are visible in a final plan but are implied by the actions in the plan.

3 Semantics

In the following sections we present a formal semantics for level 5. This semantics subsumes the semantics given in the PDDL2.1 document, so we refer, in this discussion, to full PDDL+ rather than just to level 5.

Researchers concerned with the modelling of real-time systems have developed, now well-established, techniques for modelling and reasoning about mixed discrete-continuous systems. The theory of hybrid automata [4, 3, 5], which has been a focus of interest in the model-checking community for some years, provides an ideal formal basis for the development of a semantics for PDDL+. Our examination of this theory revealed that the issues we have considered in the development of PDDL+ have been considered and, in many important cases, resolved, by research done in this field. Our main contribution has therefore been to develop a language for succinct encodings of hybrid automata, for use in planning. Our hope is that both the formal (semantics and formal properties) and practical (model-checking techniques) results in hybrid automata theory will be able to be exploited by the planning community in addressing the problem of planning for discrete-continuous planning domains.

We have developed the semantics of PDDL+ by means of a mapping to hybrid automata, effectively demonstrating how the succinct encoding of a domain in PDDL+ is used to construct an explicit hybrid automaton model.

Henzinger [4] describes a digital controller of an analog plant as a paradigmatic example of a mixed discrete-continuous system. The discrete states (control modes) and dynamics (control switches) of the controller are modelled by the vertices and edges of a graph. The continuous states and dynamics of the plant are modelled by vectors of real numbers and differential equations. The behaviour of the plant depends on the state of the controller, and vice versa: when the controller switches between modes it can update the variables that describe the continuous behaviour of the plant and hence bring about discrete changes to the state of the plant. A continuous change in the state of the plant can cause invariant conditions on the control modes of the controller to be violated, and result in a control switch.

In a similar way, PDDL+ distinguishes processes, responsible for continuous change, from events and actions, responsible for discrete change. Further, the constraint in PDDL+, that numeric values only appear as the values of functions whose arguments are drawn from the finite domain, corresponds to the requirement made in hybrid automata that the dimension of the automaton be finite.

The following formal definition is taken verbatim from Henzinger's article. Note that, in this paper, definitions are numbered to continue the sequence defined in PDDL2.1.

Definition 21 Hybrid Automata. *A* hybrid automaton *H consists of the following components:*

- **Variables.** *A finite set $X = \{x_1, \ldots, x_n\}$ of real-valued variables. The number n is called the* dimension *of H. We write $\dot{X}$ for the set $\{\dot{x_1}, \ldots, \dot{x_n}\}$ of dotted variables, representing first derivatives during continuous change, and X' for the set $\{x'_1, \ldots, x'_n\}$ of primed variables, representing values at the conclusion of discrete change.*

- **Control Graph.** *A finite directed graph $\langle V, E\rangle$. The vertices in V are* control modes. *The edges in E are* control switches.

- **Initial, invariant and flow conditions.** *Three vertex labelling functions, init, inv, and $flow$, that assign to each control mode $v \in V$ three predicates. Each initial condition $init(v)$ is a predicate whose free variables are from X. Each invariant condition $inv(v)$ is a predicate whose free variables are from X. Each flow condition $flow(v)$ is a predicate whose free variables are from $X \cup \dot{X}$.*

- **Jump conditions.** *An edge labelling function* jump *that assigns to each control switch $e \in E$ a predicate. Each jump condition $jump(e)$ is a predicate whose free variables are from $X \cup X'$.*

- **H-Events.** *A finite set Σ of h-events, and an edge labelling function $hevent : E \to \Sigma$ that assigns to each control switch an h-event.*

An *h-event* is referred to by Henzinger as an *event*, but we have changed the name to avoid terminological confusion with PDDL+. In hybrid automata (as presented by Henzinger) there is no distinction made between control switches that are under the control of an executive and those that are initiated by the world. Therefore all control switches are events. In other theories [8] control switches can be modelled as signals that are synchronised with changes in the world indicating that these changes are deliberately brought about. This corresponds to our notion of action. For our purposes a syntactic distinction between actions and events is made and the semantics reflects this distinction in the semantics of plans.

The following simple example depicts a fragment of the hybrid automaton model of the bath filling domain. In this example the variable $level$ represents the level of water in the bath and $flow$ represents the rate of flow of water into the bath. The variable $\dot{level}$ represents the rate of change of $level$ and $level'$ represents the value of $level$ after a discrete change. In control mode [*tap_off*, *plug_in*, *floor_dry*] the level of the water is unchanging (although the bath might not be empty). The *turn_on* control switch causes the bath to enter the new control mode in which the tap is on and the level is changing according to the rate of flow over time. This mode can only be maintained while the level is less than the capacity of the bath (this constraint is an invariant condition). Flooding can only occur when the level has reached capacity, and it leads to a new control mode in which the floor is wet and the level in the bath is unchanging. If the plug is pulled out, when in the mode in which the tap is off and the plug is in, the bath enters the mode in which the level in the bath is unchanging. A discrete change is brought about (in this simple model) in which the level becomes zero when this context switch occurs. This discrete change appears as a jump condition on the transition.

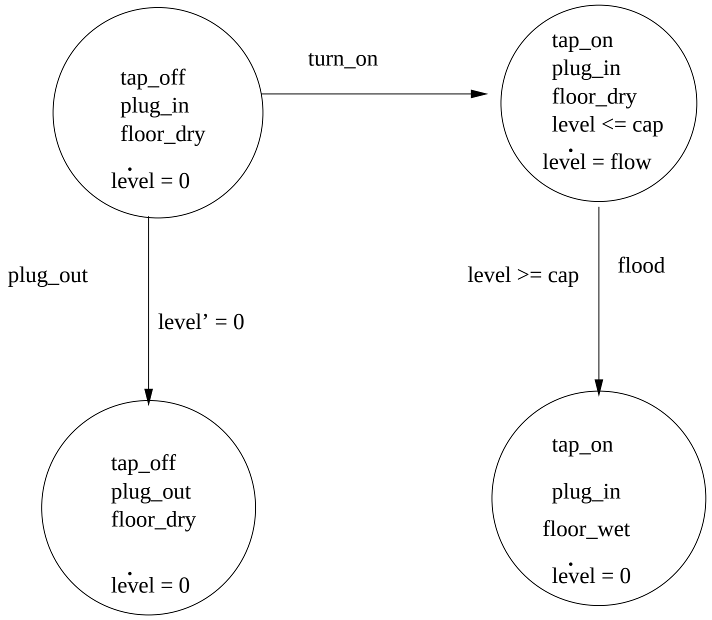

Figure 2: A fragment of the Bath Domain modelled as a hybrid automaton

4 Mapping PDDL+ Constructs into Hybrid Automata Constructs

Definition 22 Planning Instance *A* planning instance *is defined to be a pair*

$$I = (Dom, Prob)$$

where $Dom = (Fs, Rs, As, Es, Ps, arity)$ *is a tuple consisting of (finite sets of)* function symbols, relation symbols, actions, events, processes *and a function arity mapping all of these symbols to their respective arities.* $Prob = (Os, I, G)$ *is a triple consisting of the objects in the domain, the initial state specification and the goal state specification.*

The functional expressions *for a planning instance,* FEs*, are the terms constructed from the function symbols of the domain applied to (an appropriate number of) objects drawn from* Os*.*

I *consists of two parts:* $I_{logical}$ *is a set of literals from the finite collection of literals formed from the relation symbols in* Rs *and the objects in* Os*.* $I_{numeric}$ *is a set of propositions asserting the initial values of a subset of the functional expressions of the domain. These assertions each assign to a single functional expression a constant real value. A goal condition is a proposition that can include both atoms formed from the relation symbols and objects of the planning instance and numeric propositions between functional expressions and numbers.*

As *is a collection of action schemas (non-durative actions),* Es *is a collection of event schemas and* Ps *is a collection of process schemas, each expressed in the syntax of PDDL+. The functional expression schemas and atom schemas used in these action, event and process schemas are formed from the function symbols and relation symbols (used with appropriate arities) defined in the domain applied to objects in* Os *and the schema variables.*

The definition of a grounded simple domain, given in Definition 7, is extended in the obvious way to introduce grounded events $\mathcal{E}$ and grounded processes $\mathcal{P}$. Grounded actions are defined as in PDDL2.1 (Definition 8). Grounded events are defined in exactly the same way (the fact that they require a numeric precondition makes them syntactically a special case of actions). Definition 12, defining the mutual exclusion relation between action pairs, can be extended in a similar way to pairs containing one or two events.

Definition 23 Grounded Process *Each $p \in \mathcal{P}$ will have the following components:*

- **Name** *The process schema name together with its actual parameters.*
- **Precondition** *This is a proposition, Pre_p, the atoms of which are either ground atoms in the planning domain or else comparisons between terms constructed from arithmetic operations applied to functional expressions or real values. The set of ground atoms that appear in Pre_p is referred to as $GPre_p$.*
- **Numeric Postcondition** *The numeric postcondition is a conjunction of additive assignment propositions, NP_p, the rvalues of which are expressions that can be assumed to be of the form* `(* #t exp)` *where* `exp` *is* `#t`*-free.*

  A process is valid *if the function, $\mathrm{DNPF}_p : \mathbb{R}^{dim}_{\perp} \to \mathbb{R}^{dim}_{\perp}$, given by the following definition is well defined.*

  $\mathrm{DNPF}_p(\mathbf{x}) = \mathbf{x}'$ *where for each functional expression x'_i that does not appear as an lvalue in $\mathcal{N}(\mathrm{NP}_p)$, $x'_i = 0$ and $\mathcal{N}(\mathrm{NP}_p)[\mathbf{X}' := \#\mathtt{t} * \mathbf{x}' + \mathbf{x}, \mathbf{X} := \mathbf{x}]$ is satisfied.*

The following sets of functional expressions are defined for each grounded process, p:

- $L_p = \{f|f$ *appears as an lvalue functional expression in NP_p*$\}$
- $R_p = \{f|f$ *appears as an rvalue functional expression in NP_p or anywhere in Pre_p*$\}$
- $L^*_p = \{f|f$ *appears as an lvalue in an additive assignment proposition in NP_p*$\}$

In the above definition the function $DNPF_p$ gives the rate of change of the numeric values implied by the process p.

The precise measurement of numeric quantities raises similar problems to those, discussed in Section 8, involved in the precise measurement of time. In particular, there is a potentially paradoxical situation when we consider the behaviour we intend for events. Events are intended to trigger when their preconditions become true, without any further passage of time. If the preconditions require a numeric quantity to lie in an open range (for example, $x > 3$) it is impossible to tell exactly when the value of the quantity falls within the interval and hence impossible to precisely define when the event is triggered. Suppose $x = 3$ and is increasing under the control of an active process. Then the precondition of the event is not *yet* satisfied, since the precondition requires $x > 3$, but if any finite interval of time passes then we will have violated the intention that no time can pass once the event precondition becomes true. Imagine, for example, that a bath is full to capacity, but the tap is off. The precondition of the flooding event will require that the flow into the bath satisfy the condition $flow > 0$. Suppose, further, that the action of turning on the tap causes the flow to increase continuously, starting at 0 and increasing up to some limit. Then it would appear impossible to determine when the flooding event occurs – it does not occur when the tap is turned on, since the flow does not become non-zero until time has passed, but if any finite interval passes then the flooding event happens too late. One way to prevent this paradox is to insist that all event numeric preconditions are non-strict inequalities, so that they define closed intervals.

However, there are situations in which strict inequalities do not cause difficulties because of their interaction with discrete changes (for example, when flow changes in discrete jumps then a precondition $flow > 0$ causes no problem). Therefore, we do not insist that preconditions for events be non-strict inequalities, but instead rule out models in which the precise time at which events occur cannot be identified semantically. This is achieved by adding an extra real variable to our domains that measures the passage of time when event preconditions are true. This *time-slip* will only become non-zero if an event precondition becomes true, but the event it is associated with is not immediately triggered. We now insist that in all valid models the time-slip will be zero. The consequence of this device is that in all valid models events behave exactly as intuition requires: they must trigger precisely when their preconditions become true. In the paradoxical situations, such as that described above, the device will prevent a valid model from being found, since the passage of time will deadlock at the threshold of the triggering of the event.

It should be observed that Henzinger's model needs to be extended, in a simple way, to include the undefined value, $\perp$, for real-valued variables. This introduces no semantic difficulties and we have left the details of this modification implicit.

Definition 24 Semantics of a Planning Instance *Given a planning instance* $(Dom, Prob)$ *we interpret it as a hybrid automaton,* H*, as follows:*

- **Variables.** *The variables of* H *are* $X = \{x_1, \ldots, x_{n+1}\}$*, where* n *is the dimension of the planning problem. The* $n+1th$ *variable is a special control variable used to measure time-slip as discussed above. We will overload* X *to refer to the vector of variables* $\langle x_1, \ldots, x_{n+1}\rangle$*.*

- **Control Graph.** *The set of vertices* V *is formed from all subsets of the ground atoms of the planning instance. The set of edges in* E *contains an edge* e *between* v *and* v' *iff there is an action or event,* t*, applicable (definition 10) in* v *and:*

$$v' = (v - Del_t) \cup Add_t$$

We say that the action or event t *is associated with the edge* e*.*

- **Initial, invariant and flow conditions.** *The vertex labelling function* init *is defined as:*

$$init(v) = \begin{cases} false & \text{if } v \neq I_{logical} \\ \mathcal{N}(I_{numeric}) \wedge \bigwedge_{\{x_i \mid x_i \text{ not a term in} \mathcal{N}(I_{numeric})\}} x_i = \perp & \text{otherwise} \end{cases}$$

The vertex labelling function inv *is the simple proposition that ensures time-slip is zero.*

$$inv(v) = (x_{n+1} = 0)$$

The vertex labelling function flow *describes the continuous change, over all of the numeric variables, effected by the processes active on the state represented by* v*.*

Given a state v *we define* $\mathcal{P}_v$ *to be all the processes applicable in* v*. Now we can give a formal definition of* $flow(v)$ *in which time-slip is monitored along with the process* *effects:*

$$flow(v) = (\bigwedge_{i=1}^{n} \bigwedge_{A \in \mathbb{P}(\mathcal{P}_v)} \textit{context-flow}(i, A)) \wedge \textit{time-slippage}$$

where

$$\textit{context-flow}(i, A) = \bigwedge_{p \in A} \mathcal{N}(N_p) \wedge \bigwedge_{p \in \mathcal{P}_v \setminus A} (\neg \mathcal{N}(N_p) \rightarrow \dot{x_i} = \sum_{p \in A} proj_i(DNPF_p(X))$$

($proj_i$ is the ith value in the vector argument) and

$$\textit{time-slippage} = (\dot{x_{n+1}} = 0 \vee \dot{x_{n+1}} = 1) \wedge (\bigvee_{\{e \in \mathcal{E} | v \vdash S_e\}} \mathcal{N}(N_e) \rightarrow \dot{x_{n+1}} = 1)$$

- **Jump conditions.** *The edge labelling function jump is defined in two parts, one part dealing with edges associated with actions, the other with edges associated with events.*

  Given an edge e, associated with an action a, that is an exit edge for state v:

  $$jump(e) = NPre(v, Pre_a)(X) \wedge X' = NPF_a(X) \wedge \\ \bigwedge_{\{ev \in \mathcal{E} | ev \text{ is applicable in } v\}} \neg NPre(v, Pre_{ev})(X)$$

  The final conjunct ensures that the state cannot be left by an action if there is an event the preconditions of which are satisfied. It is possible for more than one event to fire simultaneously in the same state. This is discussed further in the section on plan validity.

  Given an edge e, associated with an event q, that is an exit edge for state v:

  $$jump(e) = NPre(v, Pre_q)(X) \wedge X' = NPF_q(X)$$

- **H-events.** *Σ is the set of all names of all ground actions and events. The edge labelling function hevent : $E \rightarrow \Sigma$ assigns to each edge the name of the action or event associated with the edge.*

The definition of the semantics of a planning instance is constructed around a basic framework of the discrete state space model of the domain. This follows a familiar discrete planning model semantics. The continuous dimensions of the model are constructed to ensure that the hybrid automaton will always start in an initial state consistent with the planning instance (modelled by the *init* labelling function). The state invariants ensure that no time-slip occurs in the model, and that, therefore, events always occur once their preconditions are satisfied. It will be noted that since the negation of the event preconditions is added to all jump conditions for other exiting transitions, it will be impossible for a state to be exited in any other way than by the triggered event. Finally, $flow$ models the effect on the real values of all of the active processes in a state. An additional monitoring process is added to ensure that time does not slip once an event should be triggered. The $flow$ function assigns a proposition to each state that determines a piece-wise continuous behaviour for each of the real values in the planning domain. The function is piece-wise differentiable, with only a finite number of segments within any finite interval. This definition requires a minor extension of the standard hybrid automaton as discussed below.

5 Semantics of Plans

We begin by describing a semantics for plans that is not constructive and therefore is independent of the procedure that we will construct for demonstrating plan validity. The semantics we give can be straightforwardly converted to successful simulation of the execution of actions in the corresponding hybrid automaton, but the general question of whether a plan can be simulated is only semi-decidable (assuming that the question cannot be decided in zeno systems).

The following definitions come from Henzinger's paper [4].

Definition 25 Labelled Transition System *A* labelled transition system*, S, consists of the following components:*

- **State Space.** *A (possibly infinite) set, Q, of states and a subset, $Q^0 \subseteq Q$ of initial states.*
- **Transition Relation.** *A (possibly infinite) set, A, of labels. For each label $a \in A$ a binary relation $\stackrel{a}{\to}$ on the state space Q. Each triple $q \stackrel{a}{\to} q'$ is called a transition.*

Definition 26 The Transition Semantics of Hybrid Automata *The* timed transition system S^t_H *of the hybrid automaton H is the labelled transition system with components Q, Q^0, A and $\stackrel{a}{\to}$, for each $a \in A$, defined as follows:*

- *Define $Q, Q^0 \subseteq V \times \mathbb{R}^n$, such that $(v,\mathbf{x}) \in Q$ iff the closed proposition $inv(v)[X := \mathbf{x}]$ is true, and $(v,\mathbf{x}) \in Q^0$ iff both $init(v)[X := \mathbf{x}]$ and $inv(v)[X := \mathbf{x}]$ are both true. The set Q is called the* state space *of H.*
- $A = \Sigma \cup \mathbb{R}_{\geq 0}$.
- *For each event $\sigma \in \Sigma$, define $(v, \mathbf{x}) \stackrel{\sigma}{\to} (v', \mathbf{x}')$ iff there is a control switch $e \in E$ such that: (1) the source of e is v and the target of e is v', (2) the closed proposition $jump(e)[X, X' := \mathbf{x}, \mathbf{x}']$ is true, and (3) $hevent(e) = \sigma$.*
- *For each non-negative real $\delta \in \mathbb{R}_{\geq 0}$, define $(v, \mathbf{x}) \stackrel{\delta}{\to} (v', \mathbf{x}')$ iff $v = v'$ and there is a differentiable function $f : [0, \delta] \to \mathbb{R}^n$, with the first derivative $\dot{f} : (0, \delta) \to \mathbb{R}^n$ such that: (1) $f(0) = \mathbf{x}$ and $f(\delta) = \mathbf{x}'$ and (2) for all reals $\epsilon \in (0, \delta)$, both $inv(v)[X := f(\epsilon)]$ and $flow(v)[X, \dot{X} := f(\epsilon), \dot{f}(\epsilon)]$ are both true. The function f is called a* witness *for the transition $(v, \mathbf{x}) \stackrel{\delta}{\to} (v', \mathbf{x}')$.*

In order to handle the form of flow conditions we have constructed in the mapping to hybrid automata we have to manage a slightly more general definition of the semantics, as follows.

Definition 27 The Generalised Transition Semantics of Hybrid Automata *The* generalised timed transition system GS^t_H *of the hybrid automaton H is the labelled transition system with components Q, Q^0, A and $\stackrel{a}{\to}$, for each $a \in A$, defined as follows:*

- *Define $Q, Q^0 \subseteq V \times \mathbb{R}^n$, such that $(v,\mathbf{x}) \in Q$ iff the closed proposition $inv(v)[X := \mathbf{x}]$ is true, and $(v,\mathbf{x}) \in Q^0$ iff both $init(v)[X := \mathbf{x}]$ and $inv(v)[X := \mathbf{x}]$ are both true. The set Q is called the* state space *of H.*

- $A = \Sigma \cup \mathbb{R}_{\geq 0}$.

- *For each event* $\sigma \in \Sigma$, *define* $(v, \mathbf{x}) \stackrel{\sigma}{\rightarrow} (v', \mathbf{x}')$ *iff there is a control switch* $e \in E$ *such that: (1) the source of* e *is* v *and the target of* e *is* v', *(2) the closed proposition* $jump(e)[X, X' := \mathbf{x}, \mathbf{x}']$ *is true, and (3)* $hevent(e) = \sigma$.

- *For each non-negative real* $\delta \in \mathbb{R}_{\geq 0}$, *define* $(v, \mathbf{x}) \stackrel{\delta}{\rightarrow} (v', \mathbf{x}')$ *iff* $v = v'$ *and there is a finite set of differentiable functions* $f_i : [\delta_{i-1}, \delta_i] \rightarrow \mathbb{R}^n$, *for* $i = 1, \ldots, n$ *and* $\delta_0 = 0$ *and* $\delta_n = \delta$, *with the first derivative* $\dot{f}_i : (\delta_{i-1}, \delta_i) \rightarrow \mathbb{R}^n$ *such that: (1)* $f_1(0) = \mathbf{x}$, $f_n(\delta_n) = \mathbf{x}'$ *and* $f_i(\delta_i) = f_{i+1}(\delta_i)$, *for each* $i = 1, \ldots, n-1$, *and (2) for all* $i = 1, \ldots, n$ *and for all reals* $\epsilon \in (\delta_{i-1}, \delta_i)$, *both* $inv(v)[X := f_i(\epsilon)]$ *and* $flow(v)[X, \dot{X} := f_i(\epsilon), \dot{f}_i(\epsilon)]$ *are both true.*

  The set of functions f_i *is called a* witness set *for the transition* $(v, \mathbf{x}) \stackrel{\delta}{\rightarrow} (v', \mathbf{x}')$.

The generalisation is simply that, instead of the witness function being a single differentiable function, it is replaced with a finite set of differentiable functions over linked segments. This is an equivalently expressive model, although a transformation from the generalised form to the original might be computationally expensive. This is not of direct concern here, since the model supplies a semantics rather than a computational procedure.

Definition 28 Trace and trace acceptance *Given a hybrid automaton,* H, *a* trace *for* H *is an element of the language* $(\mathbb{Q}_{>0} \times \Sigma)^*$, *such that the sequence of rationals formed by projecting the trace onto the first element of each pair is in ascending order. That is, a trace is a finite sequence of pairs of rational values and h-events, with the rational values appearing in ascending order along the sequence.*

A trace, $\tau = \langle t_i, \sigma_i \rangle_{i=1,\ldots,n}$ *is* accepted *by* H *if there is a sequence* $r = \langle u_i, q_i \rangle_{i=1,\ldots,2n+1}$, *where the* q_i *are states in the generalised timed transition system* GS^t_H *of* H, *and:*

- $q_1 \in Q^0$.

- *For each* $i = 1, \ldots, n$, $u_{2i} = t_i - \sum_{j=1,\ldots,i-1} t_j$ *and* $u_{2i-1} = 0$. *Also,* $q_{2n+1} = 0$.

- *For each* $i = 1, \ldots, n$, $q_{2i-1} \stackrel{u_{2i}}{\rightarrow} q_{2i}$ *and* $q_{2i} \stackrel{\sigma_i}{\rightarrow} q_{2i+1}$ *are both transitions in* GS^t_H.

r *is called an* accepting run *for* τ *in* H *and, if* $q_{2n+1} = (v, \mathbf{x})$, *we say that it* ends in state v with final values $\mathbf{x}$.

The original Henzinger definition allows traces where the first transition is time-stamped at 0. We disallow this in order to be consistent with the semantics of PDDL2.1 in which actions are forbidden to occur at time 0. The consequence is that initial conditions that are immediately affected by events cannot be encoded. However, processes that are initially active are legal because they do not cause state change.

Definition 29 Ordered permutation equivalent traces. *Two traces for a hybrid automaton,* τ_1 *and* τ_2, *are* ordered permutation equivalent, *written* $\tau_1 \equiv \tau_2$ *if they contain the same pairs, with the action pairs not necessarily in the same order. Event pairs must be in the same order and in the same positions in the two sequences. Note that since the numbers appearing in the pairs of these sequences must be in ascending order, the only variation in order between pairs in the two traces will be in the order of pairs of actions having the same number component.*

We will say that one sequence is *contained in* a second sequence if the elements of the first appear in the second in the same order, though not necessarily consecutively. The difference between a sequence a and a sequence b that contains a is the sequence of pairs from b that are not in a.

In the following definition we distinguish between *semantic* and *effective* validity of plans. A plan containing multiple actions and events with the same time-stamp is considered semantically valid if all orderings of the actions and events lead to the same state, but this is computationally expensive to check. We consider a plan to be effectively valid if all pairs of such actions and events are pairwise non-mutex according to the definition of the mutex relation given in Definition 12.

Definition 30 Semantics of a plan *Given a planning instance, $(Dom, Prob)$, and a plan P (using ground action names from this planning instance), the interpretation of P, $I[P]$, is the set of all traces, t, accepted by H, the hybrid automaton constructed from the planning instance, that:*

- *t contains P;*
- *the difference between P and t includes no pairs in which h-event is an action label.*

This means that an accepting trace can only add events and time-transitions to the actions in P.

We will say that P is semantically valid *if:*

- *$I[P]$ is non-empty.*
- *For each $\tau \in I[P]$, for all traces τ' such that $\tau \equiv \tau'$, $\tau' \in I[P]$.*
- *For each $\tau \in I[P]$, for each state, s, visited by τ such that τ leaves s on an event transition at time t, all events the preconditions of which are satisfied in s at t must be commutative.*

We will say that P is effectively valid *if:*

- *$I[P]$ is non-empty.*
- *For each $\tau \in I[P]$, for each state, s, visited by τ such that τ leaves s on an event transition at time t, all events the preconditions of which are satisfied in s at t must be pairwise non-mutex (by Definition 12).*
- *For each $\tau \in I[P]$, all actions with the same time stamp must be pairwise non-mutex (by Definition 12).*

Finally, we say that P achieves the goal, G, of the planning instance if every accepting run of every trace in $I[P]$ ends in a state v such that G is satisfied in v. It can be observed that plans for zeno systems cannot be constructed because of the constraint that all traces are finite.

Effective validity replaces the computationally expensive conditions of semantic validity with stronger constraints that can be confirmed efficiently. This means that some semantically valid plans will be excluded as not effectively valid. The complexity of checking effective validity is discussed in section 8.

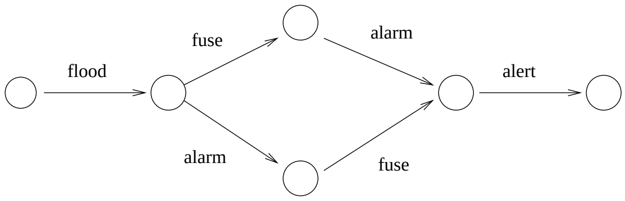

Figure 3: A cascade of events all occuring at the same instant

The third point in the definition of semantic validity is a subtle one. The execution of a plan must remain successful regardless of what order actions or events that are time-stamped at the same instant actually occur in. Events can be required, by the physics of the world, to occur in a specific order but, where there is non-determinism in event-triggering (and where it is implied by actions being given equal time-stamps) we require the execution of the plan to be robust to that non-determinism. An example arises in the following situation: suppose there is a live electricity socket on the floor of a bathroom in which the bath is filling by an active process. If the floor becomes wet by flooding the electricity supply will fuse at the same instant, but these two events must be successive (because of the dependency between them) even though they have the same time stamp. However, if a water-sensitive alarm has also been placed on the floor, then the order in which the alarm goes off and the electricity supply fuses is non-deterministic, and the correctness of the plan cannot rely on the order in which they occur. Finally, if the police are alerted when the alarm sounds in a darkened house then the event of the police being alerted is simultaneous with the flooding, fusing and alarm events, but must succeed them because of its dependence on the conditions they produce. Figure 3 depicts the chain of cascading events that this situation entails.

It might seem counter-intuitive to cram all of the events in the above situation into the same instant in time. One might prefer to try to model the time it takes for a fuse to burn through, so as to separate the fuse-blowing from the alarm triggering by epsilon time. Similarly, one might want to model the police reaction time in terms of the time it takes for the sound of the alarm to prompt a reaction in the brain. However, an attempt to capture these timing constraints in the model carries two disadvantages. First, one is required to make predictions about the granularity at which an executive can practically interact with the world. Of course, if one wishes to model reaction-time explicitly, PDDL+ allows it either through the use of processes or through the use of durative actions (as provided at levels 3 or 4). These mechanisms can be used to model the time that passes between an event and the reaction to it. Second, the explicit separation of the start and end points of the intervals over which fine-grained events occur opens up the possibility of the planner deliberately scheduling activities to take place within these intervals, leading to plans that are not robust with respect to execution in the real world.

6 Semantics of Level 4 Durative Actions

The syntax for durative actions is purely a syntactic convenience – durative actions can be mapped directly into an equivalent start-process-stop representation using the basic syntax of PDDL+. The reason for performing the mapping is in order to give durative actions a semantics in terms of the underlying structures of PDDL+, which are themselves given meaning in terms of hybrid automata as discussed in Section 3. The mapping is as follows.

Given a durative action at level 4 of PDDL2.1, it is possible to construct an equivalent collection of structures in PDDL+ level 5. The following conversion illustrates the process. Let the canonical form of a PDDL2.1 durative (level 4) action be as follows:

```
(:durative-action name
     :parameters (p⃗)
     :duration (and (at start D_S[?duration]) (at end D_E[?duration]))
     :condition (and (at start Pre_S) (at end Pre_E) (over all Inv))
     :effect (and (at start Post_S)(at end Post_E[?duration])
         (when (at start Pre^w_S) (and (at start Post^w_S)
                (at end Post^w_E[?duration]) (increase fe^w (* #t exp^w)))
         (when (and (at start Pre^x_S) (over all Inv^x)
                (at end Pre^x_E)) (at end Post^x_E[?duration]))
         (increase fe^y (* #t exp^y))))
```

It should be observed that the final three effect constructions, superscripted in their components by w, x and y respectively, can appear multiply often in a single durative action (with different propositions and expressions) For example, a single action can have several conditional effects, each dependent on different initial conditions and each having different outcomes.

```
(:action name-start
     :parameters (p⃗)
     :precondition (and Pre_S (name_clock_stopped p_1 ...p_n))
     :effect (and Post_S (not (name_clock_stopped p⃗))
                  (name_clock_started p⃗)
                  (assign (name_clock p⃗) 0)
                  (when Pre^w_S (and Post^w_S (name^w_memory p⃗)))
                  (when Pre^x_S (name^x_memory p⃗))
                  (record D_S sub-expressions)))

(:process name-process
     :parameters (p⃗)
     :precondition (and Inv (name_clock_started p⃗))
     :effect (and (increase (name_clock p⃗) #t)
                  (increase fe^y (* #t exp^y))
                  (when (name^w_memory p⃗) (increase fe^w (* #t exp^w)))))

(:event name-monitor
     :parameters (p⃗)
     :precondition (and ¬Inv (name_clock_started p⃗))
     :effect (not (name_clock_OK p⃗)))

(:event name^x-monitor
     :parameters (p⃗)
     :precondition (and ¬Inv^x (name^x_memory p⃗))
     :effect (not (name^x_memory p⃗)))
```

```
(:action name-end
     :parameters (p⃗)
     :precondition (and Pre_E (name_clock_started p⃗)
                   D_E[(name_clock p⃗)]
                   D_S[(name_clock p⃗, recorded sub-expressions)])
     :effect (and Post_E[name_clock p⃗]
                   (when (and Pre^x_E (name^x_memory p⃗)) (Post^x_E[(name_clock p⃗)]))
                   (when (name^w_memory p⃗) (Post^w_E[(name_clock p⃗)]))
                   (name_clock_stopped p⃗)
                   (not (name_clock_started p⃗))))
```

To complete the transformation, the initial and goal states must have the conditions ($name$_clock_stopped $o_1 \ldots o_n$) and ($name$_clock_OK $\vec{p}$) added to them for each appropriate durative action and argument collection.

The start and end actions need to communicate, from the former to the latter, in the case where duration constraints must be evaluated in the context of the start state. Duration constraints of this form highlight the fact that use of a durative-action entails a *promise* about the future. It is clear that any check on the duration of a durative-action that is just starting can only check the value of an *anticipated* duration. In our transformation for handling durative-actions in the PDDL2.1 manual we handled this by converting durative-actions into "meta-action schemas", in which the *?duration* value (which cannot be used as a standard expression in preconditions, since we do not support number-valued variables) had to be treated as a meta-variable to be instantiated with the duration from a plan term before the resulting action could be considered complete. This means that the transformation described in the PDDL2.1 manual requires that a domain with durative-actions, a problem *and a plan* be transformed together in order to arrive at an appropriate level 2 (durative-action-free) domain, problem and plan description. In contrast, the transformation we describe here allows the domain and problem to be transformed independently of the plan. The advantage is that the previous transformation is appropriate for plan validation (since a plan is then available), but could not be used as the basis for conversion of a planner handling simple actions into a planner capable of handling durative-actions, while the transformation above provides a way to convert durative actions into simple actions for a planner to work with (of course, it depends on a much more expressive language and this could very well make the potential benefits of the transformation debateable). The technique by which communication is achieved is that the start action must ensure that the parts of the context required to confirm the duration constraint are available to the end action, once the duration is actually known, so that the duration constraint can actually be checked. This is achieved by recording the values of all of the terms that appear in D_S (other than *?duration* itself) in new expression terms and then using these terms in the end action to construct the values required in D_S. This process is straightforward to apply in practice, if somewhat cumbersome to describe.

Note that the clock is uniquely determined by the name and arguments of the durative action, so the clock is not shared between different durative actions. Also note that there is only one action that can start each such clock and only one that can terminate it. If a plan makes use of the durative action then in the transformed domain the durative action can be simulated by the actions that start and stop the clock. In order for these actions to execute successfully the conditions must be identical to those stipulated in the durative action. If the process invariant becomes false whilst the clock is running then the monitor event will fire, causing the clock to cease to be OK. Since the clock being OK is necessary for the goal state, no valid plan can allow the process invariant to become prematurely false.

Packaging actions, events and processes into a durative action abstracts the details of how the interval ends – whether by an action or by an event. In the bath-filling example the filling interval must be ended by an action (since the durative action does not model the possibility of overflowing) whilst in the flying action the flight must be ended by the event of landing (since no traveller has control over the landing of the plane). However, there is no syntactic difference between the two examples, so the mapping to PDDL+ must ensure that both cases are consistently modelled.

The mapping converts every durative action into the family of PDDL+ constructs shown above – two actions to start and end the interval, a process to execute during the interval, a monitoring event for the invariant and a family of monitors for the conditional effects. Because an action is always used to end the interval the planner must always choose to apply it in order to obtain a goal state. In the final plan the terminating action will appear, as though it were freely chosen by the planner, even though its application was in fact necessary, and its point of application must be consistent with the duration constraint of the original durative action, so not really under the control of the planner (for example, an end-flight action will appear in a traveller's travel plan). This need not be seen, because the plan can be trivially post-processed to refer to the durative actions instead of to their underlying components.

It is important to observe that the proposed mapping introduces a monitoring event with a negative precondition. Negative preconditions can be problematic to handle, although perhaps not the most problematic feature of a language as expressive as PDDL+. It is possible to construct alternative mappings that preserve the same semantics without introducing negative preconditions. However, these have an important impact which will be discussed in Section 3.

Durative actions are given their semantics directly by means of the transformation described in this section. Several observations can be made:

1. It is not possible to have two concurrent applications of the same durative action. This is because the clock starts running as soon as a durative action is applied, which violates the pre-condition of a copy of the same action, that the clock be stopped.

2. Two separate durative actions that have final effects updating the same functional expression might not model the desired behaviour. For example, two filling actions, using different taps, applied to the same bath concurrently will cause the bath to fill without overflowing after exactly the same amount of time that it would take to fill it from a single tap. So the bath is full at the end of the concurrent execution of the two actions, but not flooded as one might expect. More strangely, if the two periods of execution overlap, then the tail-end of the second action continues to fill an already full bath without flooding it.

We have taken the decision to leave the responsibility for correct modelling in the hands of the domain designer, rather than add syntactic constraints to attempt to exclude the apparently odd semantic behaviour.

7 Complexity of Reachability in Hybrid Automata

The Reachability problem for hybrid automata is the problem of determining, given an automaton H, whether there is a trajectory of GS_H^t that visits a state of the form $(v, \mathbf{x})$. In the context of hybrid automata, v is an error state and Reachability questions are posed to determine whether the automata is safe. In the context of planning, the Reachability question is equivalent to Plan Existence. As is discussed in Henzinger's paper [4], the general Reachability question for hybrid automata is undecidable. This is unsurprising, since introducing numbers into the language results in enormous expressive power which can be used to model Turing Machine computations. What might be surprising is that, under certain constraints, which do not exclude the use of numbers, Reachability is decidable. We can exploit these constraints in the context of planning in certain classes of domains.

The Reachability problem can be decided for the class of *rectangular hybrid automata* defined by Henzinger [4]. The subset of rectangular automata that suit our present purposes impose the following constraints:

1. for each control mode v of H, the flow condition, $flow(v)$ has the form $\dot{x} = \mathbf{c}$ where $\mathbf{c}$ is a vector of rational constants.
2. for each control mode v of H, the invariant condition, $inv(v)$, constrains each x_i to lie in a single interval the end points of which are either constant rationals or infinite.
3. for each control switch e of H, the jump condition $jump(e)$ constrains each x_i to be in a single interval the end points of which are either constant rationals or infinite, and constrains each x'_i to either remain unchanged (that is: $x'_i = x_i$), or to be a rational constant.

These constraints can be satisfied in the modelling of many real-time domains. It is therefore of interest to consider how variants of PDDL+, with different expressive power, might be identified that correspond to hybrid automata models with at most the expressive power of rectangular hybrid automata.

Henzinger demonstrates that Reachability is PSPACE complete for the class of rectangular hybrid automata, and gives a PSPACE algorithm for solving it. It is interesting to consider whether this algorithm might provide a basis for planning in PSPACE. If so, planning with constrained variants of PDDL+ that still include time and numbers might remain in PSPACE if the PDDL+ variant is *sufficiently* constrained. However, there is a subtlety to be considered: Henzinger shows that Reachability is PSPACE-complete given the instance as an explicit encoding of the rectangular hybrid automaton. A PDDL+ encoding of a domain is a succinct encoding, and the extraction of the explicit automaton model can involve an exponential blow-up. If answering the Reachability question requires the explicit encoding of the automaton, then the complexity of the Reachability problem might be increased by an exponential factor. We conjecture that the explicit automaton is not required and that, for certain restricted PDDL+ variants, Reachability remains PSPACE-complete if the input encoding is succinct. Our conjecture is based on the observation that activity in a hybrid automaton is local, and there are only a polynomial number of control switches on any control mode.

In Figure 4 we present a hierarchy of graded PDDL+ variants, each associated with a hybrid automaton model of different expressive power. Critically, all but the top level of the hierarchy have decidable Reachability problems. In particular, the level 4 PDDL+

variant corresponds to rectangular hybrid automata for which the Reachability problem is PSPACE-complete. It should be noted that the exclusive use of actions and durative actions provides a convenient syntactic way of restricting the expressive power of the hybrid automaton model. This provides a strong motivation for their introduction, to complement the practical motivation that durative actions provide an intuitive way of modelling certain time-dependent behaviours.

8 Plan Validation

The validity of a plan depends on confirming that the actions are applicable in the states that result from any events and/or processes triggered by the initial state or by actions in the plan. Neither events nor processes will be visible in the finished plan, so it will no longer be safe to assume that the actions in the plan will chain together as they do in PDDL plans (actions might rely on preconditions that are established by active processes or events). Plan validation must confirm that actions interact with any active processes and events to perform a successful state transition between the initial and goal states.

Our hybrid automaton-based semantics is very powerful and allows us to give meaning to far more plans than we can efficiently validate. We cannot base the validation procedure on the requirement that the whole automaton model be available for analysis because of its exponential size with respect to the encoding of the planning instance and plan. Fortunately, the whole automaton is not needed because the current state is always known in a deterministic system[1] and there are at most k^n events leaving any state, where k is the number of domain objects (assuming that we restrict the number of arguments of an event to be n). We believe that, in practice, very dense automata are unlikely to be seen, except in very highly dynamic problem domains where the logical states of the domain are only barely distinguishable. It therefore seems to be a reasonable assumption that the local activity within an automaton will be simple (certainly over a bounded interval of time).

The key requirement for the plan validation process is that it be polynomial in the size of the problem instance and plan. We impose the following two restrictions that must be satisfied for a plan to be validated.

1. Recall that a plan is semantically valid if there is at least one trace accepted by the corresponding hybrid automaton and all *ordered permutation equivalent* traces are also accepted by that automaton. To confirm this explicitly requires consideration of the exponential number of orderings of actions with the same time stamp. We handle this by adding the condition, which is sufficient to confirm ordered permutation equivalence with only polynomial work, that all actions and events with the same time stamp are non-interfering. This condition is only sufficient because the presence of mutex interference need not prevent a trace from being accepted. However, this strong condition enables polynomial time plan validation, subject to the constraint that cascading event sequences be polynomial in the length of the plan. It is possible to construct examples, which we consider pathological, in which event sequences that are exponential in the length of the plan can appear between two actions. We intend to impose some external constraint on the number of events that can occur between two actions in order to exclude this from practical plan construction.

[1]In fact our automata are not strictly deterministic because many events can leave the same state. However, because of the constraint on all such collections of events that they be commutative this non-determinism can be ignored.

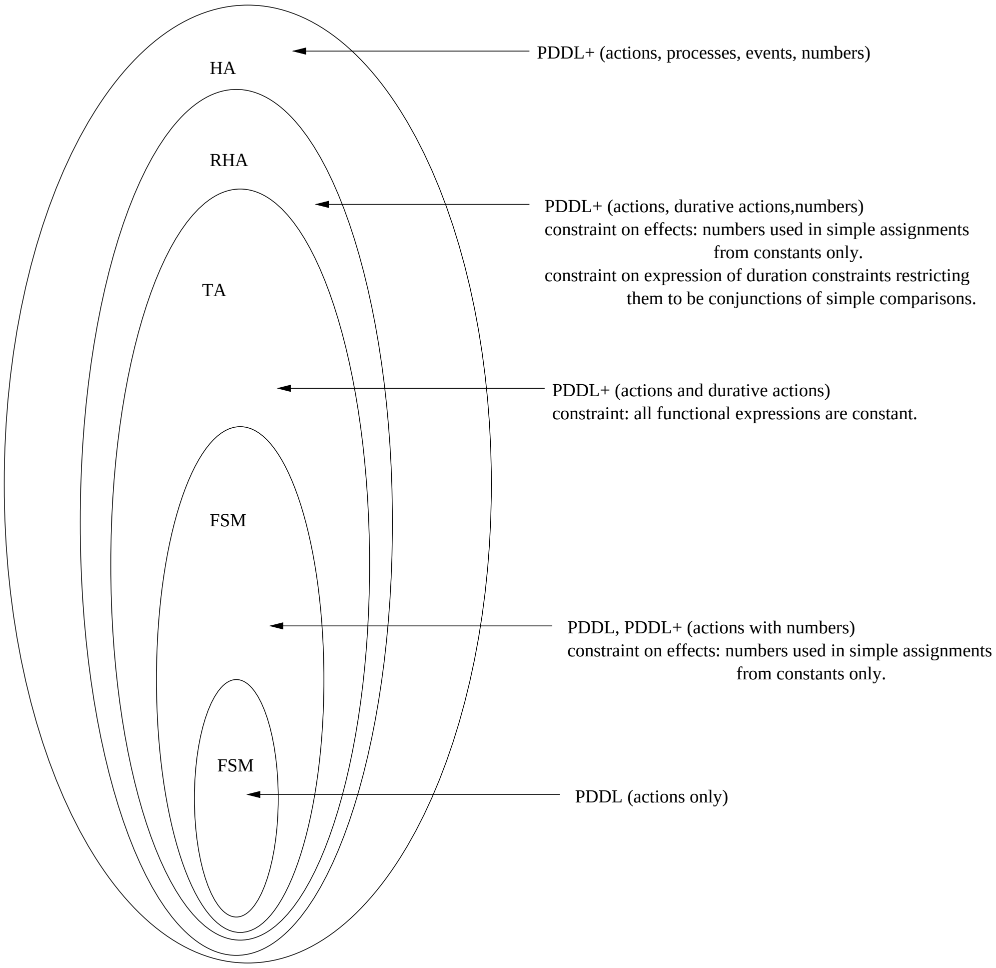

Figure 4: A Hierarchy of PDDL+ Variants corresponding to Hybrid Automaton Variants. All variants are based on ADL expressiveness, as described in the BNF (Appendix A), although a similar hierarchy can be constructed for STRIPS-based variants. The full PDDL+ language requires the expressive power of Hybrid Automata. The restricted variants require progressively less expressive power. RHA stands for Rectangular Hybrid Automaton, TA for Timed Automaton (no continuous modelling). FSM is a Finite State Machine. A *simple comparison* is a proposition of the form $(\leq ?duration\, value)$, $(\geq ?duration\, value)$ or $(= ?duration\, value)$, with *value* a rational constant.

2. It is possible for actions and processes to interact in a way that requires arbitrary precision in the measurement of time. For example, suppose the domain, discussed earlier, in which a vehicle can jettison a fuel tank, were to allow the fuel tank to be jettisoned once a *third* of the fuel is consumed. If one drive action consumes the entire tank (from full to empty) then a possible plan wopuld be to drive from start to destination, jettisoning the fuel tank at a point one third of the way into the journey. This plan depends on the ability to avoid executing the jettisoning action infinitesimally too early, so it relies on synchronisation. Provided that time points are always rational arbitrary precision does not pose a major problem and this example can be managed properly (although there is a philosophical problem, discussed below). However, if irrational values are allowed then the finite representation of numbers could lead to ambiguity about plan validity, dependent on the accuracy with which different computers represent numbers. Since we have constrained all explicit constants to be rational, the only opportunity for irrationals to be generated is by execution of processes. Provided that all change is first-order, the functions that describe continous change will be closed under rationals. For second or higher-order derivatives this will not generally be the case, but because all of the allowed functions are polynomials solutions can always be found in surds. This is not sufficient to guarantee efficient validation, since the solution of complex polynomials requires numerical methods and complexity therefore depends on the degree of accuracy desired in the solution of such expressions.

   Interestingly, the introduction of numbers complicates automatic validation simply because validity is no longer a purely logical property but is now dependent upon solving numerical equations. This is an issue that we are currently considering in the development of automatic validation tools for use at levels 3 and 4 of PDDL2.1.

It should be emphasised that these restrictions are not semantic problems but simply expedient approaches to efficient plan validation. We are in the process of defining a plan validation procedure for the subset of interpretable plans that satisfy the above criteria. We are also exploring the relationship between the hybrid automaton semantics, given here, and the simpler semantics given in our PDDL2.1 document, for PDDL2.1 levels 1, 2 and 3. We believe it is straightforward to demonstrate that these semantics are equivalent at these language levels (that the same level 1-3 plans are valid in both semantics), possibly subject to minor modifications and clarifications.

There is an interesting philosophical issue that arises and is discussed by Henzinger and his co-authors [3, 5]. It is, in fact, not possible to achieve exact precision in the measurement of time or other continuous numeric quantities. Henzinger *et al* have considered this problem through the development of *robust automata*, and also discretized time. Robust automata only accept a trace if there exists a *tube* of traces within a distance $\epsilon > 0$ of the original trace, all of which are acceptable by the original acceptance criteria. These are called *fuzzy tubes* indicating that time is fuzzily, rather than precisely, detectable. The difficulty with this approach for our purposes is that, since the ϵ is existentially quantified the domain designer cannot specify the precision with which time can actually be measured whereas, since the execution of a plan depends on the granularity being appropriate, it would be helpful if the domain designer could supply such information and require a plan that is robust to that measurement of time. On the other hand, the discretization of time, where the domain designer imposes the granularity with which time and other quantities can be measured, introduces a new difficulty because the world should be able to trigger events at precisely measured points in time. We do not yet have a solution to this philosophical problem, but are in the process of developing a semantics for *robust plans*.

9 Conclusions

We have extended PDDL+ to enable the representation of a class of deterministic mixed discrete-continuous domains as planning domains. The semantics of the language is given by means of a mapping showing how hybrid automata can be constructed from PDDL+ domains. The power of hybrid automata means that we can interpret more plans than we can efficiently validate. We describe the criteria that a plan must satisfy in order to be practically verifiable.

We identify a hierarchy of PDDL+ variants and show how they map to increasingly powerful hybrid automata models. Whilst the Plan Existence problem for full-scale PDDL+ is undecidable in general, we identify some constraints under which it is decidable – specifically in PSPACE-complete – and show that these constraints make the corresponding hybrid automata rectangular. PSPACE algorithms exist for answering the Reachability question in rectangular automata [4], and we observe that such algorithms might be able to be exploited in planning with these PDDL+ variants.

We draw attention to an important philosophical issue that arises from the fact that time cannot be measured with arbitrary precision. Although this issue has been resolved in hybrid automata theory, by means of *fuzzy tubes*, these do not have immediate utility in the planning context in which we wish to distinguish between the planner's ability to measure time and the ability of the world itself to measure time. We are working on a semantics for robust plans in which the different granularities at which these measurements take place can be resolved.

A BNF Description of level 5

We present only the extensions required to add level 5 to the BNF defined in PDDL2.1.

```
<structure-def>      ::=:time <event-def>
<structure-def>      ::=:time <process-def>
```

A.1 Events and Processes

An event is syntactically similar to an action, with the constraint that it must have at least one fluent-comparison precondition. It should also be noted that the semantics of events require that in the context of a state in which the discrete precondition of an event is true, the numeric precondition must be true within an interval that is at least semi-closed, since the moment an event becomes applicable must be identifiable.

```
<event-def>       ::= (:event <event-symbol>
                             :parameters ( <typed list (variable)> )
                             <event-def body>)
<event-symbol>    ::= <name>
<event-def body> ::= :precondition <GD-event>
                      [:effect <effect>]
```

```
<GD-event> ::= <GD>* <f-comp> <GD>*
```

Processes are also similar in syntactic style to actions. In this case, however, the effects are constrained to be entirely fluent-updates (or conditional fluent-updates) that are time-dependent. The expression of time-dependence is currently restricted to simple rate-of-change-multiplied-by-time expressions. An important constraint that we wish to observe is that if process P is active during the period from time A to time B, then the updating effects over that period are the same as would be calculated if the process were considered to be active over two time periods, A to C and C to B, where C is any time point between A and B. Because the changes effected by processes are continuous, the question of whether intervals of activity are closed or open is irrelevant, since the change in a value will be calculated to have the same value in either case. The open-closed debate is only important in considering logical effects. Since logical effects are restricted to instantaneous events or actions in this model, the standard STRIPS assumption can be used: negative effects are enacted before positive effects.

```
<process-def>      ::= (:process <process functor>
                              :parameters ( <typed list (variable)> )
                              <process-def body>)
<process functor>  ::= <name>
<process-def body> ::= [:precondition <GD>]
                       [:effect <process-effect>]
```

```
<process-effect>    ::= (and <process-effect>*)
<process-effect>    ::=:conditional-effects
                        (forall (<variable>*) <process-effect>)
<process-effect>    ::=:conditional-effects
                        (when <GD> <process-effect>)
<process-effect>    ::= (<assign-op-t> <f-head> <f-exp-t>)
```

A.2 Requirements

Requirement	*Description*
`:time`	Allow events and processes (implies `:fluents`)

B Sample Domain: Temporal Logistics

```
(define (domain timed-logistics)
  (:requirements :strips :time)
  (:predicates (obj ?o)(truck ?t)(at ?o ?l)(in ?o ?t)(airplane ?p)(airport ?s)
               (in-city ?s ?city)(city ?c)(location ?l))
  (:functors
               (time-sheet ?d)
               (resttime ?d)
               (distance ?s ?d)
               (fuel ?v)
               (fuel_consumption_rate ?v)
               (speed ?v)
               (travelled ?v))

  (:action load-truck
               :parameters (?o ?truck ?loc)
               :precondition (and (obj ?o)
                                (truck ?truck)
                                (at ?o ?loc)
                                (at ?truck ?loc))
               :effect (and (not (at ?o ?loc))
                                (in ?o ?truck)))
  (:action load-plane
               :parameters (?o ?p ?loc)
               :precondition (and (obj ?o)
                                (airplane ?p)
                                (at ?o ?loc)
                                (at ?p ?loc))
               :effect (and (not (at ?o ?loc))
                                (in ?o ?p)))
  (:action unload
               :parameters (?o ?v ?loc)
               :precondition (and (in ?o ?v)
                                (at ?v ?loc))
               :effect (and (at ?o ?loc)
                                (not (in ?o ?v))))

  (:action set-off
               :parameters (?truck ?s ?d ?city ?driver)
               :precondition
                   (and (at ?truck ?s)
                        (truck ?truck)
                        (driver ?truck ?driver)
                        (>= (time-sheet ?driver)
                            (/ (distance ?s ?d) (speed ?truck)))
                        (in-city ?s ?city)
                        (in-city ?d ?city)
                        (> (fuel ?truck)
                            (/ (* (distance ?s ?d) (fuel_consumption_rate ?truck))
                                              (speed ?truck)))
                                  )
              :effect (and (enroute ?truck ?s ?d)
                           (not (at ?truck ?s))
```

```
                        (assign (travelled ?truck) 0)
                                  ))

 (:action take-off
              :parameters (?plane ?s ?d)
              :precondition
                      (and (at ?plane ?s)
                           (airplane ?plane)
                           (airport ?d)
                           (> (fuel ?plane)
                              (/ (* (distance ?s ?d) (fuel_consumption_rate ?plane))
                                             (speed ?plane)))
                                 )

              :effect (and (enroute ?plane ?s ?d)
                           (not (at ?plane ?s))
                           (assign (travelled ?plane) 0)
                                  ))

 (:event arrive
            :parameters (?v ?s ?d)
            :precondition (and (enroute ?v ?s ?d)
                                 (>= (travelled ?v) (distance ?s ?d))
            :effect (and (at ?v ?d)
                         (not (enroute ?v ?s ?d))))

 (:process travelling
              :parameters (?v ?s ?d)
              :precondition (and (enroute ?v ?s ?d))
              :effect (and (decrease (fuel ?v) (* #t (fuel_consumption_rate ?v)))
                           (incease (travelled ?v) (* #t (speed ?v)))))

 (:action get-in
             :parameters (?truck ?driver ?location)
             :precondition (and (working ?driver)
                                        (at ?driver ?location)
                                        (at ?truck ?location)
                                        (empty ?truck))
             :effect (and (driver ?truck ?driver)
                                (not (at ?driver ?location))
                                (not (empty ?truck))))


(:action get-out
             :parameters (?truck ?driver ?location)
             :precondition (and (driver ?truck ?driver)
                              (at ?truck ?location))
             :effect (and (at ?driver ?location)
                                (empty ?truck)
                                (not (driver ?truck ?driver))))



 (:action start-shift
```

```
            :parameters (?driver)
            :precondition (available ?driver)
            :effect (and (assign (time-sheet ?driver) 0)
                         (working ?driver)
                         (not (available ?driver))))

  (:event end-shift
            :parameters (?driver)
            :precondition (and (>= (time-sheet ?driver) 6)
                               (working ?driver))
            :effect (and (assign (resttime ?driver) 0)
                         (not (working ?driver))
                         (resting ?driver)
                                 ))

  (:process work
           :parameters (?driver)
           :precondition (working ?driver)
           :effect (increase (time-sheet ?driver) #t))

  (:process rest
           :parameters (?driver)
           :precondition (resting ?driver)
           :effect (increase (resttime ?driver) #t))

  (:event standby
           :parameters (?driver)
           :precondition (and (resting ?driver)
                              (>= (resttime ?driver) 6))
           :effect (and (available ?driver)
                        (not (resting ?driver))))


)
```

C Sample Domain: The Bath

```
(define (domain bath)
  (:requirements :strips :time)
  (:predicates (off ?x ?y) (on ?t ?b) (plug_in ?b)
             (wet_floor ?b) (dry_floor ?b))
  (:functors (capacity ?b) (level ?b) (flow ?b)
             (flow-rate ?t))
(:action turn_on
        :parameters
                (?tap ?b)
        :precondition
                (and (off ?tap ?b)
                     (plug_in ?b))
        :effect
                (and (not (off ?tap ?b))
                     (on ?tap ?b)
                     (increase (flow ?b) (flow-rate ?tap)))
)

(:action turn_off
        :parameters
                (?tap ?b)
        :precondition
                (and (on ?tap ?b)
                     (plug_in ?b))
        :effect
                (and (not (on ?tap ?b))
                     (off ?tap ?b)
                     (decrease (flow ?b) (flow-rate ?tap)))
)

(:action put_plug_in
        :parameters (?b ?tap)
        :precondition
                (and (off ?tap ?b)
                     (plug_out ?b))
        :effect
                (and (not (plug_out ?b))
                     (plug_in ?b))
)
(:action pull_plug_out
        :parameters (?b ?tap)
        :precondition
                (and (off ?tap ?b)
                     (plug_in ?b))
        :effect
                (and (not (plug_in ?b))
                     (plug_out ?b)
                     (assign (level ?b) 0))
)

(:process bath_filling
        :parameters
```

```
                (?b)
        :precondition
                (<= (level ?b) (capacity ?b))
                (> (flow ?b) 0))
        :effect (increase (level ?b) (* #t (flow ?b)))
)

(:event flood
        :parameters
                (?b)
        :precondition
                (and (>= (level ?b) (capacity ?b))
                     (> (flow ?b) 0)
                     (dry_floor ?b))
        :effect
                (and (wet_floor ?b)
                     (not (dry_floor ?b)))
)
)
```

References

[1] F. Bacchus and K. Kabanza. Using temporal logic to express search control knowledge for planning. Technical report, University of Waterloo, Canada, ftp://logos.uwaterloo.ca/pub/bacchus/BKTlplan.ps, 1998.

[2] Maria Fox and Derek Long. PDDL2.1 : An extension to PDDL for modelling time and metric resources. Technical report, University of Durham, UK, 2001.

[3] V. Gupta, T.A. Henziner, and R. Jagadeesan. Robust timed automata. In *HART'97: Hybrid and Real-time Systems, LNCS 1201*, pages 331–345. Springer-Verlag, 1997.

[4] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on ogic in Computer Science. Invited tutorial.*, pages 278–292. IEEE Computer Society Press, 1996.

[5] T.A. Henzinger and J-F. Raskin. Robust undecidability of timed and hybrid systems. In *Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control. LNCS 1790.*, pages 145–159. Springer-Verlag, 2000.

[6] E. Lifschitz. On the semantics of STRIPS. In *Proceedings of 1986 Workshop: Reasoning about Actions and Plans*, 1986.

[7] D. Smith and D. Weld. Temporal Graphplan with mutual exclusion reasoning. In *Proceedings of IJCAI-99, Stockholm*, 1999.

[8] W. Yi, K.G. Larsen, and P. Pettersson. Uppaal in a nutshell. *International Journal of Software Tools for Technology Transfer*, 1(1), 1997.