Validating Plans with Exogenous Events

Richard Howey, Derek Long and Maria Fox
Department of Computer and Information Systems, University of Strathclyde, Glasgow, UK
firstname.lastname@cis.strath.ac.uk

Abstract

We are concerned with the problem of deciding the validity of a complex plan involving interacting continuous activity. In these situations there is a need to model and reason about the continuous processes and events that arise as a consequence of the behaviour of the physical world in which the plan is expected to execute. In this paper we describe how *events*, which occur as the outcome of uncontrolled physical processes, can be taken into account in determining whether a plan is valid with respect to the domain model. We do not consider plan generation issues in this paper but focus instead on issues in domain modelling and plan validation.

1 Introduction

In any real world situation things happen as a result of changes in the physical world that are not the direct consequences of deliberate action. For example, water eventually reaches boiling point when heated, a ball hits the ground when dropped, a match burns a short time after being lit, and so on. These physical changes happen instantaneously following a period in which a continuous process (heating, falling, burning, etc) has been at work. Although these processes are often initiated by deliberate action, the changes that follow are the indirect consequences of these actions. These changes can be called *events* to distinguish them from the deliberate activities of agents.

The reason for modelling events in a planning domain description is to enable a planning system to plan around them in finding a solution to a complex problem in which there are constraints on continuously changing values. For example, if a temperature must never rise above a certain level, but a component might overheat after a period of time if certain conditions are allowed to hold, then the planner must plan to avoid the overheating event by taking action to undermine these conditions. We discuss the motivations for modelling and reasoning about events in section 2.

In this paper we consider both the modelling of events and the validation of plans in which events have been either exploited or avoided. Our plan validator VAL has been extended to determine the validity of a plan in the presence of events. The planning language PDDL has been extended to PDDL+ [4] to express both events and the processes that lead to them. We summarise the key extensions in section 3. The semantics of events in PDDL+ are discussed in section 4. In section 5 we address the computational problem of grounding events and propose a solution. In section 6 we discuss the semantic ambiguities that may arise in certain modelling circumstances and the problems of implementing plan validation for PDDL+. This is followed by a summary of how these problems are handled in the implementation of VAL. Some example plans in which events are triggered during plan execution are presented in section 7.

2 Motivation

In many potential application domains for planning there are complications that cannot be captured in classical planning domain descriptions. Various extensions of classical planning have been proposed and explored, such as planning under uncertainty and planning with partial observability, with the intention of moving planning towards more realistic application domains. In the real-time control systems community there is a focus on controlling systems that are driven by physical processes as well as controller actions. Systems of this kind offer a major challenge to the planning community, being a huge opportunity for the application of deliberative and planned control, while at the same time extending beyond the expressive capabilities of classical planning languages. The extensions that are most important for representing such systems are the expression of continuous processes and the expression of system responses to situations. A system response is a state change that is triggered not by an action on the part of the executive, but by a mechanisms inherent to the physical system.

For example, if a ball is dropped onto a floor then at the point of impact with the floor the ball will change its velocity. The change is not brought about directly by the executive, as would be the case where the executive throws the ball, but by a mechanism within the physical environment (the bouncing of the ball of the floor). As can be seen in this example, the event is a consequence of an interaction between a process (the falling of the ball) and its environment (the position of the ball relative to the floor). In many real-time control problems there are situations similar to this. The reason that it is important to model this in a planning domain is that, if a planner is to attempt to control a system like this, it is necessary for the planner to be able to predict and plan around events that are triggered by the actions (or inactions) that are planned for the executive.

In PDDL2.1 we already have the expressive power to represent events that occur simply as a consequence of the passage of time (so, events such as sunrise and sunset) [3], but events that are triggered by activities that are initiated by the executive cannot be modelled without extension of the language.

In earlier work we have already examined how some forms of continuous change can be modelled and the problems it creates for plan validation. Events represent another step in the progress towards modelling and planning with models of rich physical systems.

As examples of events that more strongly motivate the need to be able to model event-behaviours in planning domains, consider the following:

- In the management of a chemical process plant, the event of a reaction being initiated when a heating vessel arrives at a certain temperature, requiring valves to be opened to release pressure or to draw off reactant.
- In the control of an orbiting observation satellite, the event of the heaters being triggered to protect sensitive parts of the satellite that are in shadow because of the orientation of the satellite, requiring planned activities to work within constrained energy supply levels.
- In a logistics-type domain, the event of a driver having reached maximum safe driving hours under European law and having to take a break, requiring planned activity to either work with the necessary delay in transit or else to have provided for a relief driver to be available at an appropriate driver-exchange site.

3 PDDL and Continuous Effects

The series of International Planning Competitions was introduced to facilitate comparison of different planning systems. A central aspect of these competitions has been the adoption of a common language to represent planning problems. The language PDDL has since become the most widely used planning language, supporting comparison between planning systems and promoting the combination of different planning techniques. PDDL has been extended, in particular by Fox and Long for the 3rd competition [8] who introduced time and numeric resources.

An important part of understanding the semantics of PDDL has been the implementation of an automatic plan validator, VAL. As PDDL incorporates more features and is able to model real world problems more accurately the role of an automatic plan validator becomes more important. This is for a number of reasons including: (i) to validate complex plans, (ii) to understand the details of the semantics and (iii) to provide insights into how to solve planning problems. The implementation of continuous effects in VAL is the first step towards planning with these effects, as well as towards planning with events. As the motivating examples in section 2 illustrate, events are typically triggered by the effects of continuous change. Indeed, where events are triggered *only* by the effects of discrete change then they must always coincide with the end points of durative actions, and could therefore be possibly rolled into the action effects as additional (possibly conditional) effects.

Continuous Effects Continuous effects were added to PDDL for accurately modelling change in real world situations. A continuous effect may only change a metric variable called a *Primitive Numerical Expression (PNE)*. A durative action that has a continuous effect on a PNE changes it so that the values taken are described by a continuous function of time. We have developed formal semantics for the inclusion of continuous effects in PDDL, called PDDL2.1 level 4. The semantics can be described by continuous activity on a real time line punctuated with discrete activity; for details see [6, 7].

Continuous effects are defined by the rates of change of PNEs in the domain model, these rates of change may refer to other PNEs that are themselves changing continuously. Thus the continuous effects are defined by a system of differential equations. Durative actions may have invariant conditions requiring that a PNE must be above (or below) a certain threshold, for example $f > k$ on $(0, T)$. If f is changing continuously we need to consider the roots of $f - k$ on $(0, T)$ in order to confirm that invariants such as this are satisfied. For details on differential equations and rooting finding techniques in the context of plan validation see [6]

4 Semantics of Exogenous Events

Events were introduced into PDDL by Fox and Long with the definition of PDDL+ [4], a powerful extension of PDDL intended to model a wide range of interesting real world scenarios. The language includes the ability to define background processes of the world and exogenous events into the planning process. For example, a bath might overflow if too much water flows into it, representing an event triggered by the process of filling. In PDDL+ we could encode this event by:

```
(:event flood
 :parameters (?b - bath)
 :precondition (and (>= (volume ?b) (capacity ?b))
                    (> (flow ?b) 0))
 :effect (and (wet_floor ?b)
              (assign (flow ?b) 0)))
```

Fox and Long in [4] defined the formal semantics of PDDL+ in terms of hybrid automata [5], which is attractive as it is a widely accepted model of mixed discrete-continuous activity. However, certain details, in the semantics of events and, particularly, in the implementation of events, leave unanswered questions. This paper will not attempt to define a formal semantics, but will explain when events are triggered and also discuss several of the complexities that arise in implementing machinery for handling events. The (informal) semantics presented relate to the implementation of the validation of events in VAL.

4.1 Events Triggered by Discrete Change

An event is *triggered* in a plan when its precondition *becomes* true. That is, when the state of the world progresses from a state that does not satisfy the event precondition to a state that does satisfy the precondition. Firstly, let us consider events that are triggered by *discrete* change. After the execution of an action (or a number of actions) all events that

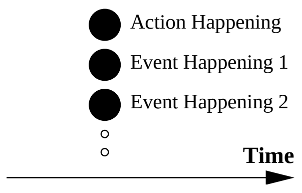

Figure 1: Event happenings

have their preconditions satisfied are executed together as an *event happening* (see figure 1). These events might trigger another event happening and so on. The event happenings are equivalent to action happenings as defined in [2], and are therefore subject to the same execution constraints. In particular *mutex* conditions, which state that for any pair of actions (events in this case) their preconditions and effects must not interfere with one another, otherwise the plan that contains them is invalid. Note that events in different happenings are not subject to these execution constraints.

Two important considerations must be taken into account, here. Firstly, we adopt the view that events are causally linked to the actions (or events) that trigger them, so that they succeed their triggering actions (events). This means that we do not consider events in one happening to be mutex with actions that triggered them even though they might occur at the same time point. This leads us to the second consideration: in PDDL2.1 considerable effort was made to enforce a view that actions that are mutex should be executed in succession, with their times of execution being separated by a minimal (context dependent) separation, ε, representing the minimum response time of the executive responsible for execution of the plan. This separation cannot be justified for events: events are a response triggered in the world, not the actions of an executive (although, in some domains events might represent the response of an executive that is not under the direct influence of the planner — we will ignore that possibility in this discussion). Thus, events are caused by state transitions, occurring after the triggering conditions are achieved and as a direct consequence of them, but there is no reason for there to be a delay between the cause and its effect. Therefore, events have to occur at the same time as their causes trigger them. A consequence of this is that event happenings can occur at the same instant as the action happenings that trigger them, but are, nevertheless, considered to occur after the actions that caused them. We arrive at the notion of multiple happenings at a single time point that are, nevertheless, sequenced. This idea appears to be exactly the semantics proposed by Bacchus and Kabanza for actions in TLPlan [1] and, by McDermott, for OPTOP [9]. The key difference here is that we only allow event happenings to stack at the same instant, never actions.

4.2 Triggering and Untriggering Events

An important observation of when events should be triggered is that they are only triggered when event preconditions *become* true. This naturally leads to the question of when is an event triggered again once it has already been triggered? It cannot be triggered at every point when the precondition is satisfied, otherwise the event would be triggered constantly on a subset of the real time line which is non-sensical. However it is sensible to trigger an event more than once; in order to achieve this we say:

- *An event cannot be triggered again until its precondition has not been satisfied at an intermediate time since it was last triggered.*

The fact that the time at which the precondition is not satisfied is an intermediate time point is important as this implies that an event may not be triggered twice at the same time point within different event happenings. Since the number of possible events in a given domain is finite (there is only a finite number of objects that can be used as parameter values to distinguish events), this constraint implies that only a finite number of possible events can occur at any time. We say that an event is *untriggered* when its precondition first becomes false after previously being satisfied when the event was triggered. Checking when events are untriggered is as important as checking when events are triggered, since this determines when the event may be triggered again.

Another consequence of an event only being triggered when its precondition becomes true is that no events are triggered in the initial state. If an event precondition is satisfied in the initial state then the event cannot be triggered until its precondition is firstly not satisfied.

In general, we would expect an event to delete one or more of its own preconditions, preventing the event from recurring without further actions or events to re-establish the conditions for the event to occur.

4.3 Events Triggered by Continuous Change

Events are most interesting when they are triggered by continuous change, as in the bath example where the volume of water in the bath becomes too great and the bath overflows. With the ability to model both continuous change and events it is possible to accurately model interesting real world situations that are otherwise impossible to express. For example, a simple thermostat which operates a heater to regulate temperature. When the temperature is too cold the heater is switched on and the temperature rises, when it is too hot the heater switches off and the temperature begins to drop, and so on.

The extension of the semantics of continuous effects in PDDL to continuous effects with events can be seen as an extension of the *semi-simple plan* as described in [7]. A plan with continuous effects can potentially trigger an event at any point during execution. Consider the following example of triggering events between two actions as in figure 2:

1. Between the two actions we check for any events that may be triggered, see figure 2 part 1.
2. The event with the earliest time to be triggered is considered, event 1, see figure 2 part 2.
3. Before the execution of event 1 we manage the continuous activity on the interval from action 1 to event 1. We check any invariant conditions, then update the value of any continuously changing PNEs so they are correct for the application of the effects of event 1.

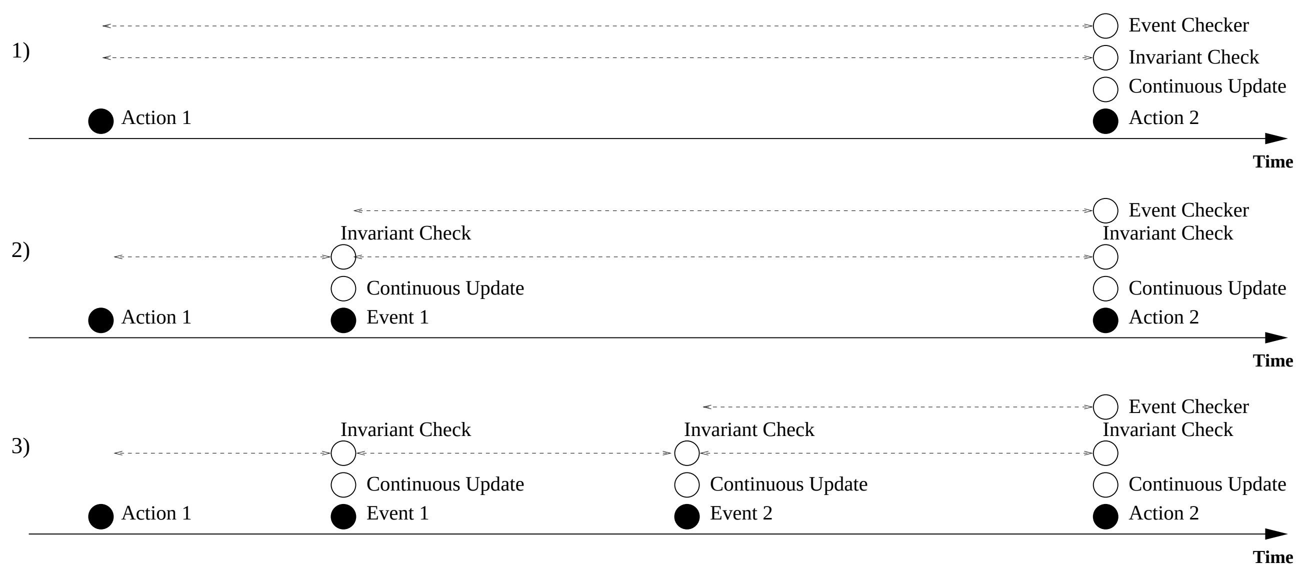

Figure 2: Events triggered by continuous effects

4. Event 1 is executed.
5. Next we must check the interval from event 1 until action 2 for events that may be triggered.
6. Consider the event with the earliest time to be triggered, event 2, see figure 2 part 3.
7. Again we manage the continuous change by checking any invariant conditions on the interval from event 1 to event 2. The continuously changing PNEs are then updated.
8. Event 2 is executed.
9. As before we must check the interval from the last event, event 2, to action 2 for events that may be triggered.
10. If there are no events to be triggered on the interval from event 2 to action 2 then we return to considering action 2.
11. All invariant conditions on the interval from event 2 until action 2 are checked. The continuously changing PNEs are updated and then action 2 is executed.

This example demonstrates the basic semantics of events within a plan with continuous effects. To extend this to the general case there are several points to take into account:

1. Happenings consisting of sets of actions or events are executed rather than single actions and events.
2. The time at which an event is executed may be identical to the last action or event. Then no invariant conditions are checked or continuously changing PNEs updated.
3. The event checking is on an interval that is closed on the left and open on the right. This is consistent with the semantics of mixed discrete-continuous plans in PDDL.
4. We always need to check for events on the interval from an event to the next action even if this interval has already been considered. This event may have changed the state of the world, in particular the continuous effects.
5. We always check to untrigger events when checking to trigger events. Events are untriggered at arbitrary time points within a plan affecting which events may be triggered after these time points.

Despite the simple basic semantics of events within PDDL there are still a number problems that can arise. These problems are discussed in section 6.

5 The Grounding Events Problem

When validating a plan with events using VAL there are various problems that arise, in particular the computation problem of handling the vast number of ground events. Events are defined in PDDL using similar syntax to that of an action, for example:

```
(:event blow-up
 :parameters (?kitchen ?match)
 :precondition (and (gas-leak ?kitchen)
                    (in ?match ?kitchen)
                    (lit ?match))
 :effect (and (explosion) (not-happy)))
```

Events are never explicitly part of a plan, so there is no event such as `(blow-up my-kitchen match-113)` listed in any plan, even though the plan might contain actions that will bring about the preconditions (say, turning on the gas in `my-kitchen` and striking `match-113`). An event is triggered whenever its precondition becomes true within the plan. In the domain definition an event is not *ground*: that is, the parameters are not instantiated. So, when the plan is executed we need to consider every ground instance of each event. If there are 5 kitchens and 362 matches then that is $5 \times 362 = 1810$ events to check at every point of execution of the plan! Or perhaps consider an event with four parameters, each of which could have 25 different values, that is $25^4 = 390625$ events! Clearly the number of ground events grows exponentially with the number of parameters that an event has and polynomially with the number of objects. Although there are reasonable bounds to the number of parameters and objects there can still be a huge number of ground events. Therefore checking each event precondition may be problematic in terms of computation of time, thus we wish to check the event preconditions as efficiently as possible.

5.1 Reduction of the Grounding Events Problem

The first observation to be made when checking event preconditions is that events can only be triggered when something in the world is changed. Therefore one of the parameters must be an object in the world that has just changed and, moreover, the property of the object that has changed must occur in the event precondition. For example consider the event `(blow-up ?kitchen ?match)`, and at some point in a plan the action `(light-match match-94)` is executed. Then we only need to check the precondition of `(blow-up ?kitchen match-94)` for every kitchen in the domain. Using this as a starting point we can reduce the number of event preconditions to check.

The second observation to be made is that the change of literals and PNEs can occur in two ways: i) Discretely when non-durative actions are executed. ii) Continuously between discrete happenings when PNEs are defined by a system of differential equations. Therefore event preconditions need to be checked after happenings if they are affected by the changed literals or PNEs. Also, we can check event preconditions that depend on continuously changing PNEs between discrete activity. This requires more consideration, but only requires an extension of the techniques used for discrete change, which is discussed later.

Map into a set of parameter lists The grounding events problem can be managed through a map which is used to (efficiently) calculate which events are triggered at a particular point in a plan. The map is from an unground event, a state and a set of literals and PNEs that have changed since the last state to a set of *parameter lists*. The resulting ground events are only triggered subject to further checks that are discussed in section 4.2.

Definition 5.1 Parameter list *A parameter list is an ordered list of object names for a given unground event written* $(p_1, p_2, \cdots, p_n)$*, where each* p_i *is an object name with the correct type corresponding to the unground event, or an undefined parameter denoted by* $\perp$.

A given parameter list and unground event represents a ground event. Using the undefined parameter, $\perp$, we are able to express a set of parameters where $\perp$ could be any object. The use of $\perp$ is appropriate when a parameter does not affect the truth value of a proposition.

Definition 5.2 *Let* ϕ *be the map from an unground event,* E*, an unground proposition,* P*, a state,* S*, a set of changed literals and* $PNEs$*,* L*, to a set of parameter lists,* K*, where* K *is the complete set of parameter lists defining all of the ground events that could be triggered, written:*

$$\phi(E, P, S, L) = K.$$

The proposition, P, is included so that ϕ may be applied to sub-formulae of the precondition of the unground event, E.

5.2 Addressing the Grounding Events Problem

Let E be an unground event, P an unground precondition, S a state, L a set of literals and PNEs then we define $\phi(E, P, S, L)$ as follows depending on the structure of P:

Literal If P is a literal then for each literal in L with truth value true and the same name as P we add a parameter list to K. For one such literal, L_i in L, we match the parameters with those of P and then construct a parameter list by instantiating the corresponding parameters in the event parameter list. For example: if E is `(blow-up ?kitchen ?match)`, P is `(lit ?match)` and L_i is `(lit match-34)` then the parameter list is `(`$\perp$`,match-34)`.

Disjunction If P is a disjunction, $\vee_i X_i$, then $\phi(E, \vee_i X_i, S, L) := \cup_i \phi(E, X_i, S, L)$. That is, the set of parameter lists given by each disjunct.

Conjunction If P is a conjunction, $\wedge_i X_i$, then it is not as simple as the disjunctive case since all the conjuncts must be satisfied for each parameter list. The set of parameter lists, K, is constructed as follows:

1. For each conjunct, X_a, calculate the set of parameter lists $J_a = \phi(E, X_a, S, L)$.
2. Each parameter list in J_a satisfying $\wedge_{i \neq a} X_i$ is added to K.

The last point implies that every undefined parameter must be defined if it has an impact on the truth value of $\wedge_{i \neq a} X_i$. Thus we need to use the map, ψ, as defined in section 5.3 (in point 2. we calculate $\psi(E, \wedge_{i \neq a} X_i, S, J_a)$). Note that the set K is pair-wise unique.

Implication If P is an implication, $X \rightarrow Y$, then $\phi(E, X \rightarrow Y, S, L) := \phi(E, \neg X \vee Y, S, L)$.

Negation If P is a negation, $P = \neg Q$, then ϕ is defined as below if Q is a literal, comparison, conjunction, disjunction, implication and negation respectively.

$$\begin{aligned}
\phi(E, \neg Q, S, L) :=& \text{the set of parameter lists given by} \\
& \text{each literal in } L \text{ with truth value false} \\
& \text{and the same name as } Q. \\
\phi(E, \neg Q, S, L) :=& \text{as for comparisons without negation} \\
& \text{except we check that the comparison} \\
& \text{is not satisfied instead of satisfied.} \\
\phi(E, \neg(\wedge_i X_i), S, L) :=& \phi(E, \vee_i \neg X_i, S, L) \\
\phi(E, \neg(\vee_i X_i), S, L) :=& \phi(E, \wedge_i \neg X_i, S, L) \\
\phi(E, \neg(X \rightarrow Y), S, L) :=& \phi(E, X \wedge \neg Y, S, L) \\
\phi(E, \neg\neg Q', S, L) :=& \phi(E, Q', S, L)
\end{aligned}$$

Comparison If P is a comparison then ϕ returns the list of parameters that satisfy the comparison, where at least one of the parameters is derived from a PNE in L. This implies that we must test the comparison for every set of parameters, where at least one is derived from a PNE in L. For example consider the comparison

$$a^5 - 2b^4 + 3c^3 - 2d^2 + e > 0$$

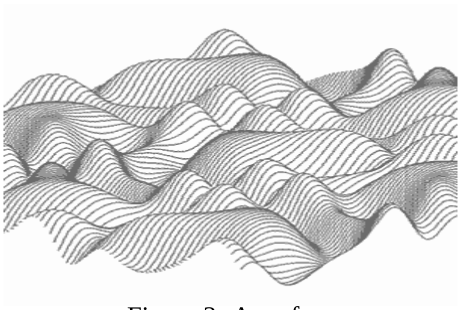
Figure 3: A surface

where a, b, c, d and e are PNEs which are given by separate parameters for the event, E. If only one PNE in the comparison can correspond to a PNE in L, say a, this still leaves four undefined parameters. Supposing each PNE in the comparison can be given by ten different parameters, then this is $10^4 = 10000$ comparisons that must be checked!

In general it can be very complex to calculate which combinations of PNEs satisfy the given inequality. For example, consider an inequality, $f(a, b) > 0$, with just two PNEs for some function f of a and b. The function f may be arbitrarily complex, see figure 3, so calculating the values of a and b that satisfy the inequality is far from trivial. Therefore the best way to find which values of a and b satisfy the inequality, from our limited set of values, is to test each combination.

5.3 Grounding Events Problem and Conjunction

In order to calculate the set of parameter lists that satisfy an unground proposition with conjunction using the map ϕ it is necessary to define the following map:

Definition 5.3 *Let ψ be a map from an event, E, an unground proposition, P, a state S, a set of parameter lists, K_1, to a set of parameter lists, K_2. Where K_2 is the set of parameter lists for E given by K_1 that satisfy P in S, written:*

$$\psi(E, P, S, K_1) = K_2.$$

Let E be an event, P an unground proposition, S a state, K_1 a set of parameter lists then we define $\psi(E, P, S, K_1)$ as follows depending on the structure of P:

Literal If P is a literal then K_2 is the set of parameter lists derived from K_1 such that P has truth value true. If P has undefined parameters then this requires systematically testing the truth value of P for every instantiation of the undefined parameters in K_1 which correspond to those in P.

Disjunction If P is a disjunction, $\vee_i X_i$, then $\psi(E, \vee_i X_i, S, K_1) = \cup_i \psi(E, X_i, S, K_1)$. That is, the set of parameter lists given by each disjunct.

Conjunction If P is a conjunction, $\wedge_i X_i$, then K_2 is the set of parameter lists derived from K_1 such that P has truth value true. This is calculated by systematically testing P against every set of parameters derived from K_1.

Implication If P is an implication, $X \rightarrow Y$, then $\psi(E, X \rightarrow Y, S, K_1) := \psi(E, \neg X \vee Y, S, K_1)$.

Negation If P is a negation written $P = \neg Q$ then ψ is defined as below if Q is a literal, comparison, conjunction, disjunction, implication and negation respectively.

$$\begin{aligned}
\psi(E, \neg Q, S, K_1) :=& \text{the set of parameter lists given by} \\
& \text{instantiating } K_1 \text{ such that } Q \text{ is false.} \\
\psi(E, \neg Q, S, K_1) :=& \text{as for comparisons without negation} \\
& \text{except we check that the comparison} \\
& \text{is not satisfied instead of satisfied.} \\
\psi(E, \neg(\wedge_i X_i), S, K_1) :=& \psi(E, \vee_i \neg X_i, S, K_1) \\
\psi(E, \neg(\vee_i X_i), S, K_1) :=& \psi(E, \wedge_i \neg X_i, S, K_1) \\
\psi(E, \neg(X \rightarrow Y), S, K_1) :=& \psi(E, X \wedge \neg Y, S, K_1) \\
\psi(E, \neg\neg Q', S, K_1) :=& \psi(E, Q', S, K_1)
\end{aligned}$$

Comparison The set of parameters, K_2, is calculated in a the same way as ϕ for comparisons except the parameter list starting point is K_1 and not L.

5.4 Events Triggered by Continuous Effects

Events triggered on an interval between two happenings, $I = [a, b)$, by continuous activity are calculated as follows:

1. The map ϕ is applied to each unground event precondition where L consists of the PNEs that are changing continuously, yielding a set of ground events, E. We assume that any comparisons with continuously changing PNEs are satisfied at this point (or not satisfied if appropriate).
2. Next we calculate the subset of intervals of I that each ground event precondition is satisfied on. From this we obtain an event happening at time t, E_t, which contains the set of events that have the earliest satisfied precondition (we also check that these events have not been triggered already).
3. Let U_r be the set of events from E such that: (i) each event was triggered before I, and (ii) each event precondition is not satisfied at the minimum time in I, r, for any event precondition to be not satisfied from the events in (i).
4. If E_t is defined and U_r is not then we execute the event happening E_t as described in section 4.3 accounting for continuous change. Then $[t, b)$ is the next interval to be considered for triggering events by continuous activity.
5. If U_r is defined and E_t is not then we untrigger the events in U_r at time r. Then $[r, b)$ is the next interval to be considered for triggering events by continuous activity, since untriggering events may allow these events to be triggered on the remaining interval.
6. If both E_t and U_r are defined then if $t \leq r$ then we execute the happening E_t and $[t, b)$ is the next interval to be considered. Otherwise we untrigger the events in U_r at time r and $[r, b)$ is the next interval to be considered.

After an event happening has been executed we check for events triggered by discrete change before considering the next interval for events that may be triggered by continuous change. When there are no more events to trigger or untrigger on I it is no longer considered.

5.5 Grounding Events Conclusion

Although, in the general case, checking every ground event can be a very expensive task this need not be the case if the domain and problem are written carefully. In order that events can be quickly processed in VAL the following advice should be followed.

- *Each unground event should be defined so that when its precondition may be satisfied due to a PNE or a literal that has changed the resulting number of events that can be ground from the partially ground event is small. Ideally this number should be one.*

For example in section 4 a `flood` event may have its precondition satisfied due to three PNEs changing. If any one of these PNEs are considered then the `flood` event is completely ground, that is the bath in question is named, so there is only one ground event precondition to check. If we consider the `blow-up` event from the beginning of section 5 and suppose that a match has been lit then we only need to check the same number of ground events as there are kitchens. However, if a kitchen develops a gas leak then we need to check the same number of ground events as there are matches! This could be a large number, but perhaps still acceptable. The more undefined parameters there are after taking into account a changed PNE or literal the worst the situation is, depending on the number of objects.

6 Problems with Events

6.1 Problems with Semantics

Timing of Events Suppose two events are triggered at times close to one another, the first event, e_1 at time t_1 and the second event e_2 at time t_2. If the events are such that $t_2 - t_1 < 0.01$, where 0.01 is the given tolerance for plan validation then are the events considered to be executed at the same time? If the events are considered to be executed at the same time then the mutex conditions between the two events needs to be checked. The issue of whether events are mutex or not, in relation to the tolerance value of plan validation, is very much similar to that of actions and tolerance as discussed in [2]. There are some extra considerations when considering events (see also the discussion concluding section 4.1):

- The timing of events could be considered as precise, since events are triggered when conditions are met in the world state and are not dependent on any executive, so there are no timing inaccuracies. However, the accuracy to which numbers can be represented prevents this from being totally workable, although attractive otherwise. Calculating the exact times at which events are triggered also causes problems for this option.
- A plan is modelled as being totally deterministic in its execution, even when events have been considered. When two mutex events occur in different happenings they are considered not to interfere, even if the happenings occur at the same instant. This is not entirely intuitive and raises significant questions about the way in which the temporal flow associated with causal chains is abstracted in the management of events.

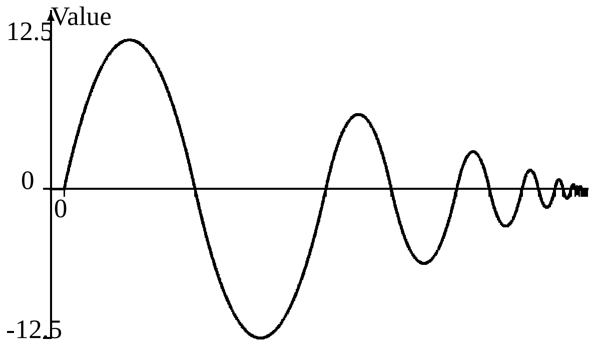

Figure 4: Graph of a converging PNE.

Infinite Sequence of Events in Finite Time It is possible for events to cascade, occurring at successively closer instants, so that there is an infinite number of events in a finite time. For example, a bouncing ball could be modelled so that each time it hits the ground an event is triggered that gives it an upward velocity equal to some fraction of its downward velocity on impact, leading to a sequence of smaller and smaller bounces, taking less and less time to complete. For this to happen the times of the events must converge to a certain limit. Let us call this time c. Let e_n be a sequence of events triggered at corresponding times t_n which converge to time c. Let S_n be the state of the world after the application of event e_n. If we firstly assume that the time at which the events occur can be measured accurately and that an infinite amount of events can be processed then there are a number of outcomes to this situation:

1. The state sequence, S_n, converges to a particular state. Let the value of a particular PNE be f_n for each state S_n, then f_n will converge to a certain limit, for example see figure 4. The sequence f_n may be constant after a certain value of n or it may continually be approaching the limit. The logical state of the world must also converge, and since there are a finite number of predicates this implies that the logical state must eventually become constant. In this case the outcome of the sequence of events is deterministic.
2. The state sequence, S_n, does not converge to a particular state. In this case S_n may be such that the logical states in the sequence do not converge. Thus after time c the truth value of some predicates is unknown meaning that the outcome of the sequence of the events is indeterministic. The numerical state may also contain PNEs where the value is unknown after time c. To ensure that the plan is still deterministic after time c any undefined PNEs or predicates must be redefined before they are accessed. If any undefined PNE or predicate is accessed before the end of the plan then it is no longer deterministic.

Timing is a problem for the above sequence of events, since the events are triggered arbitrarily close together.

From an implementation point of view the occurrence of an infinite number of events within a finite time causes many problems. No matter how accurately we measure the time at which events are triggered the times will soon be too close

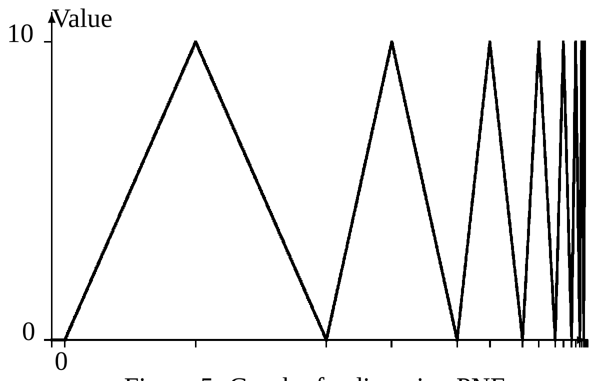

Figure 5: Graph of a diverging PNE.

to represent with different numbers. However, executing an infinite number of events by considering them each in turn is simply impossible. In a convergent case we could calculate the state limit and apply the correct state at the time of the sequence limit. Any consequences during the time at which the sequence of events are triggered would need to be taken into account. Unfortunately to account for such cases, in general, is extremely problematic and unlikely to be of practical value.

Evaluating the Goal The goal of a plan is normally checked after the execution of the last action, but when events are present in the domain this may not be appropriate. If an event is triggered after the last action then the goal condition may not be satisfied. Therefore it might be more appropriate to check the goal after the last event is triggered. On the other hand, an alternative perspective is to argue that if an event undermining the goals is sufficiently far away from the end of the plan, then the plan can be said to have usefully achieved its goals. In domain models in which all continuous change is encapsulated within durative actions, and therefore within the lifetime of a plan, there is no possibility of events being triggered after the last timed literal has had its effects. Therefore, it is possible to identify a point of stability for every plan in such domain models.

6.2 Problems with Implementation

Roots of Functions Events triggered by continuous effects are triggered by a continuous function of time crossing some threshold. Consequently the times of such events are calculated from the roots of continuous functions (the values where the function is zero). The value of the roots are always calculated to within a certain accuracy. Thus the timing of the event is always to within a certain accuracy. This point is particularly important when two events (or one event and an action) are very close together. The ordering of the events may have a different outcome, so it should be considered if the two events are mutex or not.

When events are triggered or untriggered The time at which events are triggered can have a significant impact on the execution of a plan, since the ordering of events and actions may cause different events to be triggered in the subsequent plan. However, equally important is whether events are triggered or untriggered at all. Consider a curve with non-negative values that only touches the axis at a point, and an event is triggered if the curve is non-positive. Now, in this case the event would be triggered, but if the curve had to be strictly negative to trigger the event then the event would not be triggered. Therefore we must carefully consider where the cut off point for triggering an event is, and no matter where it is drawn there could be accuracy issues to whether or not it is triggered. If a planning problem contains events triggered by changes in PNEs then *the inaccuracies of PNEs may affect the accuracy of the logical state of the world.*

The same considerations apply when determining whether to untrigger an event as this has a serious impact on the outcome of a plan. In fact when to untrigger an event can be even more problematic. If an event, e, has been triggered due to a continuous change in a PNE, f, and no more events are triggered at this time we must check for events that are untriggered after this time since e has changed the state of the world. Now when evaluating the event precondition of e to check if the event should be untriggered (which may be the case if e changed the value of f) we may find that the precondition is not satisfied implying that e should be untriggered. This could be due to inaccuracy in testing the event precondition. The problem is that the event precondition is (usually) on the cusp of being satisfied so the slightest inaccuracy could result in an incorrect answer.

Comparisons for event preconditions may use strict or non-strict inequalities, but since the accuracy of roots and the representation of numbers is limited this distinction is meaningless for the time at which events are triggered.

6.3 Implementation in VAL

In the implementation of VAL events are triggered at the time calculated to the best degree of accuracy available. The degree of accuracy necessarily includes the error of roots of continuous functions and the representation of numbers. Therefore two events can occur as close to one another as possible subject to the representation of numbers, without checking if the two events are mutex or not. However, if two events are to be triggered at the same time then the mutex conditions must be checked. Two events may be triggered at the same time due to accuracy problems or even in the incorrect order which is unfortunately unavoidable. Well written planning problems should aim to minimize problems created by inaccuracies.

The interesting ‘infinite events in finite time’ example is not explicitly handled in VAL as in the majority of cases this would result in an indeterministic outcome. To sensibly handle this scenario is not a worthwhile pursuit since any real world model is unlikely to require this feature, certainly not initially as events are first considered in planning problems. VAL will try to validate such a plan and fail in an unpredictable manner or never terminate, such a sequence of events should then be obvious to the user. In conclusion the domain, problem and plans should be written to avoid such sequences of events (which should be easy).

The goal of a plan is checked after the last event to be triggered after the last action has been executed.

7 Examples

7.1 Emptying a Tank

Consider a tank which is full of some mystery liquid that we wish to empty. In order to empty this particular tank we must firstly open the valve to the tap on the tank which can only stay open for a limited time of 150 time units. The tap can be turned to increase and decrease the flow of the mystery liquid within certain parameters. We wish to empty the tank but the tank is not allowed to run dry in case it rusts, so we must settle for reducing the volume to below a certain small amount. Below is a domain encoding in PDDL:

```
(:durative-action open-tank
 :parameters (?t)
 :duration (= ?duration  150)
 :condition (and )
 :effect (and (decrease (volume ?t)
               (* #t (volume-rate ?t)))))

(:durative-action increase-flow
 :parameters (?t)
 :duration (>= ?duration  0)
 :condition (and (over all (<= (volume-rate ?t) 6.5)))
 :effect (and (increase (volume-rate ?t)
           (* #t (volume-rate-constant ?t)))))

(:durative-action decrease-flow
 :parameters (?t)
 :duration (>= ?duration  0)
 :condition (and (over all (>= (volume-rate ?t) 0)))
 :effect (and (decrease (volume-rate ?t)
           (* #t (volume-rate-constant ?t)))))

(:event dry-tank
 :parameters (?t)
 :precondition (<= (volume ?t) 0)
 :effect (and (assign (volume-rate ?t) 0)   (dry ?t)
              (assign (volume-rate-constant ?t) 0)))
```

If the volume of liquid becomes zero then the event of the tank becoming dry is triggered, this event has the effect of ensuring that the model of the volume of liquid in the tank is correct. This allows the plan to continue if necessary. Suppose we execute the following plan:

```
1: (open-tank tank) [150]
5: (increase-flow tank) [25]
86: (decrease-flow tank) [25]
```

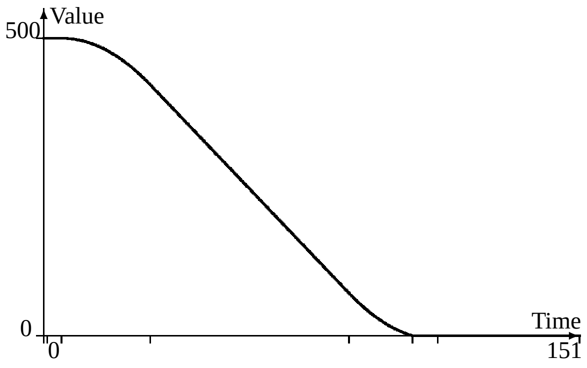

Figure 6: Graph showing the `dry-tank` event triggered

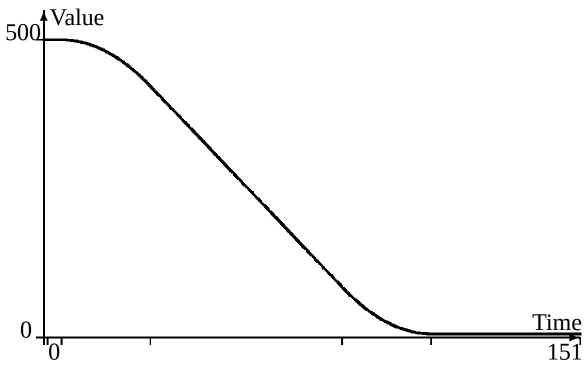

Figure 7: Graph of `(volume tank)` and no events

Firstly we open the valve on the tank and then increase the flow of liquid out of the tank. Recognizing that the tank must not run dry we lastly reduce the flow of liquid to zero. Unfortunately the flow of liquid was reduced too late and the `dry-tank` event was triggered, see figure 6. However if we execute the last action a bit sooner we can avert the tank running dry and reduce the volume sufficiently, see figure 7.

7.2 Thermostat

With a small number of simple events it is possible to model quite complex background behaviour of continuously changing PNEs, for example consider the following example of a thermostat:

```
(:durative-action change-temp
 :parameters (?t)
 :duration (>= ?duration  0)
 :condition (and )
 :effect (and (increase (temp ?t) (* #t (temp-rate ?t)))
    (increase (temp-rate ?t) (* #t (temp-rate-rate ?t)))
    (increase (integral-temp) (* #t (temp ?t)))))

(:event too-hot
 :parameters (?t)
 :precondition (> (temp ?t) 15)
 :effect (and (assign (temp-rate-rate ?t) -2)))

(:event too-cold
 :parameters (?t)
 :precondition (< (temp ?t) 5)
 :effect (and (assign (temp-rate-rate ?t) 1.5)))
```

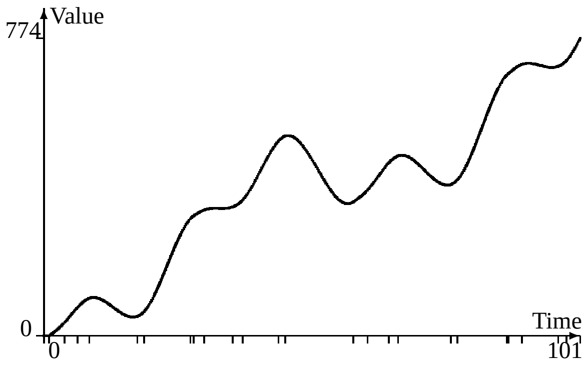

Figure 8: Graph of `(integral-temp)`.

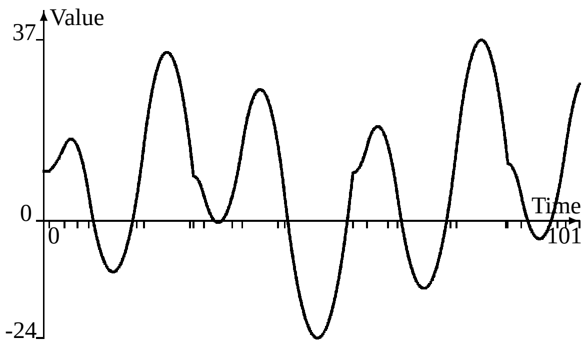

Figure 9: Graph of `(temp unit)`.

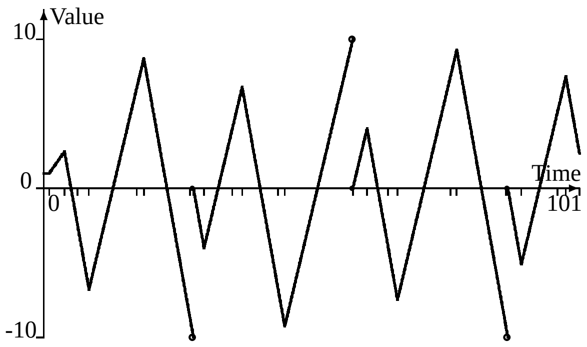

Figure 10: Graph of `(temp-rate unit)`.

```
(:event too-much-temp-rate
 :parameters (?t)
 :precondition (>= (temp-rate ?t) 10)
 :effect (and (assign (temp-rate ?t) 0)))

(:event too-little-temp-rate
 :parameters (?t)
 :precondition (<= (temp-rate ?t) -10)
 :effect (and (assign (temp-rate ?t) 0)))
```

The plan ‘`1: (change-temp unit) [100]`’ executed produces the continuous change in the PNEs as shown in figures 8, 9 and 10, for certain initial values. The values taken by the PNEs seem to exhibit quite unpredictable values, yet in fact, the values are deterministic.

8 Conclusion

The first step to developing planners that are able to handle planning problems with events is to present an unambiguous semantics. This paper has discussed the semantics of PDDL with events and any issues arising. An important part of the presentation of the semantics is the implementation of the semantics, this has been done in our plan validation tool, VAL, see figures 4, 5, 6, 7, 8, 9 and 10 for example output. VAL is a very important and useful tool for the development of any planner that is to handle events, not only to validate plans produced by the planner. The implementation within VAL provides many insights and starting points for extending planners to handle events, as well as the possibility of using VAL directly in the planning process itself.

There are various problems with the semantics and implementation of validating plans using events, but many interesting problems can be modelled by making certain restrictions. In particular to ensure fast validation of plans with events the problem of grounding events must be taken into account, this was the focus of section 5. Ideally a change in the world state should lead directly to the events which have to be triggered without considering many irrelevant candidate events. Other considerations must be accounted for, such as avoiding cascading events that create an infinite number of events in finite time. The accuracy of PNEs must also be taken into account when they are used. We are reviewing our semantics of events in order to decide whether an alternative model would remove these problems. In particular, a model in which a (tiny) separation is imposed between the triggering of events and their enactment appears plausible and promising.

The availability of the automatic plan validator, VAL, to validate plans with events is one of the first steps in building planners than can handle events. The scope of the planning problems that can be captured using events is greatly increased, supporting much more accurate models of real world situations. The objective is to ultimately bring these aspects together in order to have planners capable of planning with even richer domain representations than at present.

References

[1] F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[2] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of AI Research*, 20, 2003.

[3] M. Fox, D. Long, and K. Halsey. An investigation into the expressive power of PDDL2.1. In *Proceedings of ECAI'04*, 2004.

[4] Maria Fox and Derek Long. PDDL+ : Planning with time and metric resources. Technical report, University of Strathclyde, UK. Available at: `planning.cis.ac.uk/competition/`, 2002.

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

[6] R. Howey and D. Long. Validating plans with continuous effects. In *Proc. of the 22nd Workshop of the UK Planning and Scheduling SIG*, 115–124, 2003.

[7] R. Howey, D. Long, and M. Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proc. of 16th IEEE International Conference on Tools with Artificial Intelligence*, 2004.

[8] D. Long and M. Fox. The 3rd International Planning Competition: Results and analysis. *Journal of AI Research*, 20, 2003.

[9] D. McDermott. Reasoning about autonomous processes in an estimated-regression planner. In *Proc. of ICAPS'03*, 2003.