PDDL2.1 : An Extension to PDDL for Expressing Temporal Planning Domains

Maria Fox and Derek Long
University of Durham, UK
maria.fox@durham.ac.uk d.p.long@durham.ac.uk

June 11, 2001

1 Introduction

This document describes the first three levels of PDDL+, an extension of PDDL intended to support the representation of deterministic planning domains involving time and other numeric-valued resources. We have developed five levels of the language, the first four of which will be used extensively in the AIPS-2002 planning competition. Levels 1 to 4 of PDDL+ will be referred to as PDDL2.1 and comprise the officially agreed extensions to PDDL for use in the competition. The full details of levels 4 and 5 are given in an accompanying document [3].

The five levels of PDDL+ are organised in the following way. Level 1 corresponds to the propositional and ADL levels of McDermott's PDDL. Level 2 supplements level 1 with numeric variables and the ability to test and update their values instantaneously. Levels 3 and 4 contain *durative actions* the effects of which occur some time after execution of the action. Level 3 durative actions contain no continuous effects, whilst level 4 durative actions can have continuous effects. Because durative actions allow only restricted access to the states of the variables they affect, level 4 simplifies the modelling of real-time domains. Level 5 is a natural extension of level 4 and presents a way forward for the design of an expressive planning domain description language, capable of representing arbitrary real-time, mixed discrete/continuous domains. Level 5 is a completed language with a formal semantics, and therefore provides a strong foundation for further development of the PDDL sequence.

In this paper we describe the syntax and semantics of the first three levels of PDDL2.1. The syntax of level 4, together with an intuitive semantics, is also given here. The formal semantics of Level 4, and the syntax and semantics of level 5, are described in a separate paper [3]. Because these levels allow the modelling of continuous change their semantics is more complex than that required to describe discrete state change. The semantics we have developed constructs a mapping from level 4 and 5 plans to hybrid automata models of mixed discrete/continuous change. Although level 5 is not part of the officially agreed PDDL2.1, and therefore will not be used in the competition, we describe it in that paper because we believe it poses some important challenges to the community and raises many important issues in domain representation and temporal planning.

2 Objectives and Ground Rules

The domain modelling language PDDL2.1 has been developed for use in the AIPS-2002 planning competition in which we intend to make use of domains involving time and numbers. In particular we wish to consider mixed discrete-continuous domains in which the planning agent is able to bring about continuous change (for example: in the level of available resources such as fuel, or in the continuous state of a system such as a water tank) by making discrete state changes (for example: by driving a vehicle between two locations, or by turning on a tap). Mixed domains of this kind cannot be captured fully by a purely discrete model, and yet they arise very commonly in planning domains of interest and many researchers have attempted to handle them in a variety of ways (for example [1, 10, 9]). In designing levels 3 and 4 of PDDL2.1 we have been influenced by these approaches and have attempted to combine their best features whilst resolving a number of semantic problems. We discuss these problems, and our solutions to them, in the following sections.

It is desirable for PDDL2.1 to be backwardly compatible with the fragment of PDDL that has been in common usage in order to facilitate participation in the competition by as wide a section of the community as possible. McDermott's original PDDL provides a clean and well-understood basis for development and embodies a number of design principles that we wish to retain. We have therefore extended PDDL in principled ways to achieve the additional expressive power without compromising the role of PDDL as a neutral modelling language expressing "physics, not advice" [6]. In order to encourage wide participation in the competition we intend to introduce the extensions incorporated in PDDL2.1 in a staged way. These stages correspond to the different levels of PDDL2.1 described in this paper.

One of our objectives in extending PDDL has been to supply a means by which more interesting plan metrics than plan length can be used to judge the value of a plan. The use of numbers in a domain provides a platform for measuring consumption of critical resources and other parameters. An example of a metric we can now model is that fuel use must be minimized or that overall execution time must be minimized. The syntax for specifying these metrics is supplied in Appendix A.

We make the following two guarantees of backward compatibility:

1. All existing PDDL domains (in common usage) are valid PDDL2.1 domains. This is important to enable existing libraries of benchmark problems to remain valid.

2. Valid PDDL plans are valid PDDL2.1 plans.

3 Level 2: Numeric effects

The existing PDDL provides support for numbers by allowing numeric quantities to be assigned to and updated. These are captured as numeric *fluents* because they express change. We refer to numeric fluents as *functional expressions* throughout the paper. An example of their use is illustrated in Figure 1, taken from Drew McDermott's PDDL paper in AI Magazine [6]. The syntax of functors used in this article does not appear in the PDDL manuals 1.1 and 1.2 [7], where fluent variables are defined instead (see figure 2). We consider that the approach taken in the AI Magazine article is the better one, because functional expressions are not allowed to appear as arguments to predicates. The branching of the planner's search space, at choice points corresponding to action selection, is therefore

```
(define (domain jug-pouring)
  (:requirements :typing :fluents)
  (:types jug)
  (:functors
        (amount ?j -jug)
        (capacity ?j -jug)
          - (fluent number))
  (:action empty
        :parameters (?jug1 ?jug2 - jug)
        :precondition (fluent-test
                (>= (- (capacity ?jug2) (amount ?jug2))
                         (amount ?jug1)))
        :effect (and (change (amount ?jug1) 0)
                     (change (amount ?jug2)
                           (+ (amount ?jug1) (amount ?jug2)))))
)
```

Figure 1: Pouring water between jugs, AI Magazine style.

always over finite ranges. The idea of fluent variables conflicts with this because these can occur as arguments to any predicate and they do not define finite ranges.

Our approach in PDDL2.1 is to use the AI Magazine proposal and not allow terms to include functional expressions. Using our proposed syntax for expressing numeric assignments and updates we would, in PDDL2.1, express the jug-pouring operator of Figure 1 as presented in Figure 3.

4 Level 3: Durative Actions

In PDDL2.1 one of the numerically varying quantities is time. An 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. In level 4 numeric values can change continuously and are accessible at arbitrary points on the time-line of the plan. Level 3 is a simplification of level 4 in which there are no continuously changing quantities.

There are several alternative ways of modelling temporal planning domains using durative actions. The simplest way is to make durative actions encapsulate continuous change so that the correct values of any affected variables are guaranteed only at the end points of the implied intervals. For example, a durative action modelling bath-filling will specify the duration of the action, perhaps as a function of the rate of flow of water into the bath and the capacity of the bath, and the effect of the action is that the bath be full. The level of water in the bath is only guaranteed to be correctly updated at the end points of the interval determined by the duration. This means that no action that might occur in parallel can access the correct level of water in the bath – so it will not be possible to fill the bath and add bath salts when a specified level has been reached. The bath-filling behaviour is described by specifying the precondition that the taps be on and the plug in, and the effect that the bath be full and the taps off. The duration is given by the (pre-computable) value of duration multiplied by rate of flow.

```
(define (domain jug-pouring)
  (:requirements :typing :fluents :conditional-effects)
  (:types container)
  (:predicates
         (contents ?x - container ?y - (fluent number))
         (capacity ?x - container ?y - number))

  (:action pour
        :parameters (?source ?dest - container)
        :vars (?sfl ?dfl - (fluent number) ?dcap - number)
        :precondition (and (contents ?source ?sfl)
                           (contents ?dest ?dfl)
                           (capacity ?dest ?dcap)
                           (fluent-test (<= (+ ?sfl ?dfl) ?dcap)))
        :effect (when (and (contents ?source ?sfl)
                           (contents ?dest ?dfl))
                           (and (change ?sfl 0)
                           (change ?dfl (+ ?dfl ?sfl)))))
)
```

Figure 2: Pouring water between jugs, PDDL Manual style.

```
(define (domain jug-pouring)
   (:requirements :typing :fluents)
   (:types jug)
   (:functions
        (capacity ?j - jug)
        (amount ?j - jug))

   (:action pour
       :parameters (?jug1 ?jug2 - jug)
       :precondition (>= (capacity ?jug2) (+ (amount ?jug1) (amount ?jug2)))
       :effect (and (assign (amount ?jug1) 0)
                    (increase (amount ?jug2) (amount ?jug1)))
)
```

Figure 3: Pouring water between jugs, PDDL2.1 style.

This approach is problematic because there is no way of confirming that the plug remain in the bath, or that the taps be left on, throughout the specified interval, if other actions referring to the plug or the taps are allowed to occur concurrently. Indeed, there is no way of syntactically distinguishing between *pre*conditions and *invariant* conditions in an action representation. A coarse solution to the resulting semantic problem is to prevent concurrent actions that refer to the same objects, but this excludes many valid plans.

A more sophisticated approach allows preconditions to be annotated with time points, or intervals, so that the requirement that a condition be true at some point, or over some interval, within the duration of the action can be expressed. For example, using such an annotated precondition it would be possible to express the requirement that bath salts be added within two minutes of the start of the bath-filling action. If preconditions of an action are annotated, with times other than the start time of the action, they are not really *pre*conditions but simply conditions on the success of the action. If effects can also be specified to occur at arbitrary points within the duration of the action then it is possible to express effects that occur before the end of the specified duration. It is also possible to distinguish between conditions that are local to specific points in the duration of the action and those that are invariant throughout the action.

Allowing reference to arbitrary time points makes both the syntax and the semantics of action representation complex. Syntactically, if time points can be expressed as functions of the end points of the interval and other values, how complex can these functions be? What is the computational impact of allowing arbitrarily complex functions to be expressed? Semantically, does it matter if the time point referred to computes to a point outside of the duration of the action? Perhaps a plan is simply invalid if this problem arises, or perhaps it can be given an interpretation in some cases. There is another semantic problem in requiring an action to be applied (to achieve a timed condition) at a specified point in time because it is not possible to determine the outcome of possibly interacting synchronized actions until the actions are physically performed. The need for this arises because the power of the language would enable interval relationships to be expressed and these rely on synchronizing activities in a precise way.

Even if the above problems are overcome, this increased expressive power has a price in terms of the *nature* of the semantics that must be developed. Durative actions necessarily move away from the modelling of the physics of the domain towards the modelling of advice about how to behave in the domain. This becomes clear when we consider how to give meaning to plans consisting of durative actions. If a plan commits to the execution of an action at time t, and the successful application of that action depends upon a condition being true at time $t+t'$, then the plan must be considered meaningless if that condition is not true at time $t + t'$. Given this, it can be seen that durative actions make guarantees about how plans that use them will develop over time. These guarantees are the embodiment of advice about how to achieve particular intentions in the domain – they do not model the physics of the domain, which are independent of the intentions of any actor in that domain. A consequence is that meaning cannot be given to plans that initiate actions without committing in advance to how (and when) they will be terminated. That is, having started a durative action it must be allowed to complete as specified. This can only be justified by saying that the domain designer knows in advance all of the desirable ways of behaving in the world and that the planner never needs to identify novel ways of combining the more primitive elements of these behaviours.

For these reasons we have philosophical objections to an extensive development of PDDL in the durative actions direction. Level 5 of PDDL+ deliberately excludes the modelling of advice in favour of pure physics. However, because most of the work on temporal planning has been based on various forms of durative action we have developed levels 3 and 4 as a compromise to facilitate participation in the competition and to push forward the planning horizon to include benchmark domains with temporal characteristics. The compromise we have made is to allow the specification only of restricted forms of timed conditions and effects in the description of a durative action. In particular, at level 3, the modeller specifies the *local* pre- and post-conditions of the end-points of the interval, as well as (optionally) invariant conditions that must hold throughout the interval. That is, the modeller can refer to the start and end points of the duration but not to any intermediate points in the interval. This makes it possible to exclude as invalid precisely those plans that violate the necessary conditions for successful completion of the durative action without allowing the modeller the expressive power required to model complex temporal interactions. The details of the syntax in which these relationships are expressed are given in the BNF description of PDDL2.1 in the appendix.

Durative actions at level 3 can be given a formal semantics in a straightforward way. If no invariant is specified this means that a plan containing durative actions can be transformed into one containing just level 2 actions, one for each of the end-points of every durative action in the original plan. When invariants are specified it is necessary to confirm that the invariant remains true after every level 2 action that occurs between the start and end-points of the durative action specifying that invariant. This is achieved by a mapping described in Section 6. The specification of local pre- and postconditions, and the fact that invariant conditions can be identified, means that we can be more flexible about allowing concurrent behaviours. Provided that no action accesses a variable at the exact point at which it is updated by another action it is possible to allow concurrent actions even when they refer to the same variables. Conflicts over variables can only occur at the start and end points of actions. It is possible for concurrent actions to update and access the same variables within an overlapping interval, because it can be determined whether an invariant property is being maintained on either side of any update operation. We explain how invariants can be monitored during such intervals in Section 8. Although accessing and updating the same variable simultaneously is not allowed, simultaneous updates to the same variable can be made provided that the update operations are commutative. The actions performing commutative updates are non-interacting because the order in which they are performed does not affect the final value of the updated variable. These issues are explained in Section 8.

In level 3 of PDDL2.1 the modelling of temporal relationships in a durative action is done by means of *temporally annotated* conditions and effects. All conditions and effects of durative actions must be temporally annotated. The annotation of a condition makes explicit whether the associated proposition must be asserted at the *start* of the interval (the point at which the action is applied), the *end* of the interval (the point at which the final effects of the action are asserted) or over the interval from the start to the end (invariant over the duration of the action). The annotation of an effect makes explicit whether the effect is immediate (it happens at the start of the interval) or delayed (it happens at the end of the interval). No other time points are accessible, so all discrete activity takes place at the identified start and end points of the actions in the plan.

As an example of a level 3 durative action Figure 4 shows how the action of flying from a to b is modelled. The local precondition of the start of the period is that the traveller be at an appropriate location within a for catching the plane. Local effects include that the traveller be on the plane and no longer at a. The first of these is immediate and is then

```
(:durative-action fly
   :parameters (?p - plane ?t - traveller
                ?a ?b - location)
   :duration (= ?duration (flight-time ?a ?b))
   :condition (and (at start (at ?p ?a)) (at start (at ?t ?a))
                   (over all (aboard ?t ?p)) (over all (inflight ?p))
                   (over all (> (fuel_level ?p) 0))
   :effect (and (at start (aboard ?t ?p))
                        (at start (not (at ?t ?a)))
                        (at start (not (at ?p ?a)))
                        (at start (inflight ?p))
                        (at end (at ?p ?b))
                        (at end (at ?t ?b))
                        (at end (not (inflight ?p)))
                        (at end (not (aboard ?t ?p)))
                        (at end (decrease (fuel_level ?p)
                               (* ?duration (consumption_rate ?p))))
           )
)
```

Figure 4: A durative action for flying.

invariant throughout the execution of the action. The effect of the final end-point is that the plane (and the traveller) be at location b. The duration constraint makes `?duration` equal to the flight time from a to b.

Durative actions can have conditional effects. The antecedents and consequents of a conditional effect are temporally annotated so that it is possible to specify that the condition be checked at start or at end, and that the effect be asserted at either of these points. The semantics makes clear that a well-formed durative action with conditional effects cannot require the condition to be checked after the effect has been asserted. Conditional effects arise at all levels in PDDL2.1. We discuss how their occurrence in level 3 actions is interpreted in Section 8.1.

5 The Syntactic Structure of a Plan

A level 3 plan is a sequence of timed actions, where a timed action has the following syntactic form:

$$t : (action\, p_1 \ldots p_n)$$

In this notation t is a non-negative rational number expressed in floating point syntax, and $(action\, p_1 \ldots p_n)$ is the name and actual parameters of the action to be executed at that point in time. Actions from any of the first three levels can co-occur in a plan. No special separators are required to separate timed actions in the sequence and the actions are not required to be presented in time-sorted order. It is possible for multiple actions to be given the same time stamp, suggesting that they should be executed concurrently. This issue is discussed further in Section 6. Durative actions must be given with an additional field indicating the duration. This is given with the syntax:

$$t : (action\, p_1 \ldots p_n)\, [d]$$

where d is a rational valued duration, written in floating point syntax.

In order to retain compatibility with the output of current planners we make the following concession. If the plan is presented as a sequence of actions with no time points, then it is inferred that the first action is applied at time 1 and the succeeding actions apply in sequence at integral time points one unit apart. Note that GraphPlan plans must be modified to explicitly associate the appropriate time point with each action so that their potential concurrency can be exploited. A sequence of actions in which only a subset of the actions are time-stamped is not accepted as a plan.

6 Introduction to the Semantics of Levels 2 and 3

In sections 7 and 8 we provide a formal semantics for levels 2 and 3 of PDDL2.1. Together these sections contain 20 definitions. The lengthy treatment is necessary because the semantics we have developed adds four significant extensions over classical planning and the semantics Lifschitz developed for STRIPS [5]. These are:

- the introduction of time, so that plans describe behaviour relative to a real time line;
- related to the first extension, the treatment of concurrency – actions can be executed in parallel and can lead to plans that contain concurrent interacting processes (although these processes are encapsulated in durative actions in PDDL2.1);
- an extension to handle numeric-valued fluents;
- the use of conditional effects, including with all of the above extensions.

All of these represent important extensions of the expressive power of the planning language. In some cases, we have made decisions that ensure that the validity of plans can be checked efficiently and we have also made decisions that run counter to our intuitions about the longer-term interpretation of valid plans, but which are necessary simplifications at this stage. Amongst the former is the decision to prevent concurrent actions from updating numeric-valued fluents, except when the operations concerned have been predefined as commutative, even though there might be intuitively acceptable accounts of the interpretation of the concurrent activity. Amongst the decisions we made to simplify the language was the decision to allow plan validity to *depend* on precise synchronisation of actions at these levels. We do not believe that, in fact, a plan that depends on actions being enacted at times measured with absolute precision can reasonably be considered valid. However, the extension of the semantics to ensure that plans of this kind are rejected (and, more importantly, with an efficient procedure) is complex and raises questions that we consider in [3] but cannot yet fully answer.

The structure of the definitions is as follows. Definitions 1 to 15 define what it means for a plan to valid at level 2. Definitions 1 to 8 set up the basic terminology, the foundational structures and the framework for handling conditional effects and functional expressions. Definitions 9 and 10 extend the classical notion of state and satisfaction of goals, or preconditions of actions, to include time and numeric-valued fluents. Definition 11 defines a plan, extending the classical definition by adding time. Definitions 12 and 13 show what we mean by execution of actions, including concurrent execution of actions. The first of this pair introduces our restrictions on valid concurrent activity. Definitions 14 and 15 are essentially standard definitions of the execution of a plan and what it means for a plan to be valid, given the basis laid in the previous definitions.

The definitions up to 15 are necessary for interpretation of plans at level 2. In section 8 we extend these ideas to level 3. We begin with definition 16, which defines ground durative actions analogously to definition 8 for simple actions. Similarly, definition 17 parallels definition 11 and definitions 19 and 20 parallel definitions 14 and 15. Definition 18 is the critical definition for the semantics of level 3 of the language, supplying a transformation of level 3 plans into simple plans whose validity can be used to determine the validity of the original level 3 plans.

7 Level 2 Semantics

We begin by defining an extension to the semantics for STRIPS first defined by Lifschitz [5]. The extension is required for several reasons. Firstly, we want to handle actions that are each executed at a specific *time* and this requires that the semantic model define not only the world state, but also the time at which that state holds. Furthermore, actions can be timed to execute at the same time point, so we have to manage concurrent actions. Secondly, Lifschitz did not address the more complex expressive power introduced by Pednault [8], as ADL. This extension adds relatively little complexity to the model except in the case of conditional effects. Finally, we have to handle effects that update functional expressions as well as purely logical state. This initial extension is designed to work with simple (instantaneous) actions, but will form the foundation for the semantics of durative actions.

The semantics is intended to describe how *plans* are given meaning. Plans in which preconditions are not satisfied are considered meaningless. We could adopt a convention that actions whose preconditions are not satisfied have no effects, but this would not extend to the case of durative actions, undermining our intention to use the simple action semantics as the foundation for the durative action semantics.

Definition 1 Simple Planning Instance *A* simple planning instance *is defined to be a pair*

$$I = (Dom, Prob)$$

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

The functional expressions *of 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.*

$Init$ consists of two parts: $Init_{logical}$ is a set of literals from the finite collection of literals formed from the relation symbols in Rs and the objects in Os. $Init_{numeric}$ 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. The 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) each expressed in the syntax of PDDL. The functional expression schemas and atom schemas used in these action 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.

Definition 2 Arithmetic Expression *An* arithmetic expression *is a term constructed from arithmetic operators applied to functional expressions and rational values.*

Definition 3 Assignment Proposition *An* assignment proposition *is formed with a simple or compound assignment operator, one functional expression, referred to as the* lvalue, *and an arithmetic expression (the* rvalue*). It should be noted that an assignment proposition cannot be interpreted directly as a proposition since it carries an implicit update operation. However, it can be converted into a proposition by (if necessary) transforming it into a simple assignment and then replacing the lvalue with the "primed" version of the same functional expression (that is, x' for x) and interpreting the assignment as standard boolean equality. For example: (*`increase` *p q) becomes $(= p' (+ p\ q))$ after this two-stage process (*`increase` *is the prefix form of the C operation $+=$ and* `decrease` *is the prefix form of $-=$).*

The "priming" of functional expressions is used to distinguish the postcondition value of a functional expression from its precondition value. The binding of the functional expressions to the values in these states is managed in subsequent definitions.

Definition 4 Additive Assignment Proposition *An* additive assignment proposition *is an assignment proposition whose assignment operator is either* `increase` *or* `decrease`*.*

Definition 5 Dimension *The* dimension, *dim, of a planning domain is defined to be the number of distinct functional expressions in the domain. Let index : $FEs \rightarrow \{1, \ldots, dim\}$ be an arbitrary correspondence between the functional expressions and integer indices into the elements of a vector of dim real values, $\mathbb{R}^{dim}$.*

Definition 6 Normalisation *The operation of normalising a ground proposition in a planning instance is defined to be the process of substituting, for each functional expression f, the literal $X_{index(f)}$. Given a ground proposition, p, its normalised form will be referred to as $\mathcal{N}(p)$. Assignment propositions are first converted into propositions in the way described in Definition 3 and then normalised as usual. Primed functional expressions are replaced with their corresponding primed literals.* **X** *is used to represent the vector $\langle X_1 \ldots X_n \rangle$*

In definition 6, the replacement of functional expressions with indexed literals is intended to allow convenient, and consistent, substitution of the vector of actual parameters for the vector of literals **X**.

Definition 7 Grounded Simple Domain *The* grounded simple domain, *$\mathcal{D}$, is obtained by instantiating the variables in As in all possible ways (according to the arity function) using the objects in Os. It should be observed that grounding removes quantifiers because Os is finite. Further, grounding causes multiple occurrences of actions and events with the same names and arguments when conditional effects are flattened [4], since the propositional modelling of* (`when` *P Q*) *is as two actions with the same name as the original, one in which the preconditions are conjoined with P and Q is added to the effect and one in which $\neg P$ is added to the preconditions. Following grounding, all actions will have the form described in the next definition and will fall into families with the same name, distinguished by their preconditions. The set of all ground propositions formed by grounding a planning instance, I, is called $\mathcal{P}_I$.*

Definition 8 Grounded Action *Each $a \in \mathcal{A}$ will have the following components:*

- **Name** *The action schema name together with its actual parameters.*
- **Precondition** *This is a proposition, Pre_a, 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_a is referred to as $GPre_a$.*
- **Positive Postcondition** *The positive postcondition is a set of ground atoms called* Add_a.
- **Negative Postcondition** *The negative postcondition is a set of ground atoms referred to as* Del_a.
- **Numeric Postcondition** *The numeric postcondition is a conjunction of assignment propositions,* NP_a. *An action is* valid *if the function,* $\mathrm{NPF}_a : \mathbb{R}^{dim} \rightarrow \mathbb{R}^{dim}$, *given by the following definition is well defined.*

  $\mathrm{NPF}_a(\mathbf{x}) = \mathbf{x}'$ *where for each functional expression x'_i that does not appear as an lvalue in* $\mathcal{N}(\mathrm{NP}_a)$, $x'_i = x_i$ *and* $\mathcal{N}(\mathrm{NP}_a)[\mathbf{X}' := \mathbf{x}', \mathbf{X} := \mathbf{x}]$ *is satisfied.*

The following sets of functional expressions are defined for each grounded action, a:

- $L_a = \{f | f$ *appears as an lvalue functional expression in* $NP_a\}$
- $R_a = \{f | f$ *appears as an rvalue functional expression in* NP_a *or anywhere in* $Pre_a\}$
- $L^*_a = \{f | f$ *appears as an lvalue in an additive assignment proposition in* $NP_a\}$

So far, the definitions have simply set up the terminology and structure required. Some comment is appropriate on the last definition: an action precondition might be considered to have two parts – its logical part and its functional expression-dependent part. Unfortunately, these can be interdependent (for example, `(or (clear ?x) (>= (room-in ?y) (space-for ?z))` might be a precondition of an action). In order to handle such conditions, we need to check whether they are satisfied given not only the current logical state, but also the current values of the domain functional expressions. This observation motivates definitions 9 and 10.

The condition that the numeric postcondition of a grounded action must satisfy rules out actions that attempt multiple updates of the same functional expression, since the proposition will attempt to constrain the primed value of the functional expressions (the postcondition value) to satisfy multiple inconsistent constraints. A valid action can only assign, or update, a functional expression in a single expression. In all that follows we will assume that actions are all valid and therefore that no functional expression appears as an lvalue in more than one assignment proposition.

The various sets of functional expressions defined in the definition 8 allow us to conveniently express the conditions under which two concurrent actions might interfere with one another. In particular, we are concerned not to allow concurrent assignment to the same functional expression, or concurrent assignment and inspection. We *do* allow concurrent increase or decrease of a functional expression. To allow this we will have to apply collections of concurrent updating functions to the functional expressions. This can be allowed provided that the functions commute. Additive assignments do commute, but all other updating operations cannot be guaranteed to do so, except if they do not affect the same functional expressions or rely on functional expressions that are affected by other concurrent assignment propositions. We use the three sets of functional expressions to determine whether we are in a safe situation or not. Note that within a single action it is possible for the rvalues and lvalues to intersect. That is, an action can update functional expressions using current values of functional expressions that are also updated by the same action. All rvalues will have the values they take in the state prior to execution and all lvalues will supply the new values for the state that follows.

Definition 9 Logical States and States *Given the finite collection of atoms for a planning instance, P_I, a* logical state *is a subset of P. For a planning instance with dimension dim, a* state *is a tuple $(\mathbb{R}, s, \mathbb{R}^{dim})$, where s is a logical state. The first value is the* time *of the state and the last dim values are the values of the dim functional expressions in the planning instance.*

The initial state for a planning instance is $(0, Init_{logical}, \mathbf{x})$ where $\mathbf{x}$ is the vector of real values corresponding to the initial assignments given by $Init_{numeric}$ (treating unspecified values as 0).

Definition 10 Satisfaction of Preconditions *Given a logical state, s, the precondition of an action, a, can be used to define a predicate on $\mathbb{R}^{dim}$, $NPre(s, Pre_a)$, as follows:*

$$NPre(s, Pre_a)(\mathbf{x}) \quad \textit{iff} \quad s \vdash_{\text{CWA}} \mathcal{N}(Pre_a)[\mathbf{X} := \mathbf{x}]$$

where $s \vdash_{\text{CWA}} q$ means that q is derivable from s under the closed world assumption – for each $p \in \mathcal{P}_I \setminus s$, p is assumed to be false.

If $\{\mathbf{x} \in \mathbb{R}^{dim} | NPre(s, Pre_a)(\mathbf{x})\}$ is not empty, then a is said to be applicable *in logical state s. The precondition of an action, Pre_a, is* satisfied *in a state, $(t, s, \mathbf{x})$, if*

$$NPre(s, Pre_a)(\mathbf{x})$$

7.1 Semantics of a Simple Plan

We now present the basis for determining the validity of a simple plan. A simple plan is a slight generalisation of the more familiar STRIPS-style classical plan, since actions must now be labelled with the time at which they are to be executed.

Definition 11 Simple Plan *A* simple plan*, SP, for a planning instance, I, consists of a finite collection of* timed simple actions *which are pairs (t, a), where t is a rational-valued time and a is an action name.*

The happening sequence*, $\{t_i\}_{i=0...k}$ for SP is the ordered sequence of times in the set of times appearing in the timed simple actions in SP. All t_i must be greater than 0. It is possible for the sequence to be empty (an empty plan).*

The happening *at time t, E_t, where t is in the happening sequence of SP, is the set of (simple) action names that appear in timed simple actions associated with the time t in SP.*

So, a plan will consist of a sequence of happenings, each being a set of action names applied concurrently at a specific time, the sequence being ordered in time. The times at which these happenings occur forms the happening sequence. It should be noted that action names are ambiguous when action schemas contain conditional effects – the consequence of grounding is to have split these actions into multiple actions with identical names, differentiated by their preconditions. However, at most one of each set of actions with identical names can be applicable in a given logical state, since the precondition of each action in such a set will necessarily be inconsistent with the precondition of any other action in the set, due to the way in which the conditional effects are distributed between the pairs of action schemas they induce.

In order to handle concurrent actions we need to define the situations in which the effects of those actions are consistent with one another. This is an extension of the idea of *action mutex* conditions for GraphPlan [2]. The extension handles two extra features: the extended expressive power of the language (to include arbitrary propositional connectives) and the introduction of numeric functional expressions. We make a very conservative condition for actions to be executed concurrently, which ensures that there is no possibility of interaction. This rules out cases where intuition might suppose that concurrency is possible. For example:

```
(:action a
  :precondition (or p q)
  :effect (r))

(:action b
  :precondition (p)
  :effect (and (not p) (s)))
```

could perhaps, intuitively, be executed in parallel from a state in which both p and q hold. The following definition will still assert that the two actions are mutex. The reason we have chosen such a constrained definition is because checking for mutex actions must be tractable and handling the case implied by this example would appear to require checking the consequence of interleaving preconditions and effects in all possible orderings. The condition on functional expressions has already been discussed – it determines that the update effects can be executed concurrently and that they do not affect values which are the tested by preconditions (regardless of whether the results of those tests matter to the satisfaction of their enclosing proposition). Essentially, we impose a rule of "no moving targets": no concurrent actions can affect the parts of the state relevant to the precondition tests of other actions in the set, regardless of whether those effects might be harmful or not.

Definition 12 Mutex Actions *Two grounded actions, a and b are* non-interfering *if*

$$\begin{gathered} GPre_a \cap (Add_b \cup Del_b) = GPre_b \cap (Add_a \cup Del_a) = \emptyset \\ Add_a \cap Del_b = Add_b \cap Del_a = \emptyset \\ L_a \cap (R_b \cup L_b^*) = L_b \cap (R_a \cup L_a^*) = \emptyset \end{gathered}$$

If two actions are not non-interfering they are mutex*.*

We are now ready to define the conditions under which a simple plan is valid. We can separate the executability of a plan from whether it actually achieves the intended goal. We will say that a plan is valid if it is executable and achieves the final goal. Achievement of the final goal is straightforward, since the final goal specification is given in the same form as the precondition of a (ground) action, so its satisfaction can be determined using definition 10. Executability is defined in terms of the sequence of states that the plan induces by sequentially executing the happenings that it defines.

Definition 13 Happening Execution *Given a state,* $(t, s, \mathbf{x})$ *and a happening,* H*, the* activity *for* H *is the set of grounded actions*

$$A_H = \{a | \textit{the name for } a \textit{ is in } H \textit{ and } Pre_a \textit{ is satisfied in } (t, s, \mathbf{x})\}$$

The result of executing a happening, H*, associated with time* t_H*, in a state* $(t, s, \mathbf{x})$ *is undefined if* $|A_H| \neq |H|$ *or if any pair of actions in* A_H *is mutex. Otherwise, it is the state* $(t_H, s', \mathbf{x}')$ *where*

$$s' = (s \setminus \bigcup_{a \in A_H} Del_a) \cup \bigcup_{a \in A_H} Add_a$$

and $\mathbf{x}'$ *is the result of applying the composition of the functions* $\{\mathrm{NPF}_a \,|\, a \in A_H\}$ *to* $\mathbf{x}$.

Note that since the functions $\{NPF_a \,|\, a \in A_H\}$ must affect different functional expressions, except where they represent additive assignment propositions, these functions will commute and therefore the order in which the functions are applied is irrelevant. Therefore, the value of $\mathbf{x}'$ is well-defined in this last definition. The requirement that the activity for a happening must have the same number of elements as the happening itself is simply a constraint that ensures that each action name in the happening leads to an action that is applicable in the appropriate state. We have already seen that conditional effects induce the construction of families of grounded actions, but that at most one of each family can be applicable in a state. If none of them is applicable for a given name, then this must mean that the precondition is unsatisfied, regardless of the conditional effects. In this case, we are asserting that the attempt to apply the action has undefined interpretation.

Definition 14 Executability *A simple plan,* SP*, for a planning instance,* I*, is executable if it defines a happening sequence,* $\{t_i\}_{i=0\ldots k}$*, and there is a sequence of states,* $\{S_i\}_{i=0\ldots k+1}$ *such that* S_0 *is the initial state for the planning instance and for each* $i = 0 \ldots k$*,* S_{i+1} *is the result of executing the happening at time* t_i *in* SP*.*

The state S_{k+1} *is called the* final state *produced by* SP *and the state sequence* $\{S_i\}_{i=0\ldots k+1}$ *is called the* trace *of* SP*. Note that an executable plan produces a unique trace.*

Definition 15 Validity of a Simple Plan *A simple plan (for a planning instance,* I*) is* valid *if it is executable and produces a final state* S*, such that the goal specification for* I *is satisfied in* S*.*

8 Level 3 semantics

Plans with level 3 durative actions can be given a semantics in terms of the semantics of simple plans. Handling durative actions that have continuous effects is more complex and, although we have explored ways to extend the simple plan semantics to treat them we have found that the structures become at least as complex as, and much more baroque than, the semantics described in our companion paper, based on timed hybrid automata.

Level 3 durative actions are introduced into the framework we have defined so far by generalising definition 1 in the obvious way (adding durative action schemas). The definition of the grounded domain must now be extended to define the form of grounded durative actions. However, this definition can be given in such a way that we associate with each durative action two simple actions, corresponding to the end points of the durative action. These simple actions can, together, simulate almost all of the behaviour of the durative action. The only aspects that are not captured in this pair of simple actions are the duration of the durative action and the invariants that must hold over that duration. These two elements can, however, be simply handled in a minor extension to the semantics of simple plans, and this is the approach we adopt. By taking this route we avoid any difficulties in establishing the effects of interactions between durative actions – this is all handled by the semantics for the concurrent activity within a simple plan. As we will see, one wrinkle in this account is the handling of durative actions with conditional effects that contain conditions and effects that are associated with different times or conditions that must hold over the entire duration of the action. Since these cases complicate the semantics we will postpone treatment of them until the next section and begin with durative actions without conditional effects.

Definition 16 Grounded Durative Actions *Durative actions are grounded in the same way as simple actions, by replacing their formal parameters with constants from the planning instance and expanding quantified propositions. Each grounded durative action, DA, with no continuous effects and no conditional effects defines three simple actions, DA_{start}, DA_{end} and DA_{inv}, as follows. Note that the definition of durative actions requires that the condition be a conjunction of temporally annotated propositions. Each temporally annotated proposition is of the form* `(at start p)`, `(at end p)` *or* `(over all p)`, *where p is a standard (unannotated) proposition. Similarly, the effects of a durative action (without continuous or conditional effects) are a conjunction of temporally annotated simple effects (add effects, delete effects or assignment propositions).*

DA_{start} (DA_{end}) has precondition equal to the conjunction of the set of all propositions, p, such that `(at start p)` *(*`(at end p)`*) is a condition of DA and effect equal to the conjunction of all the simple effects, e, such that* `(at start e)` *(*`(at end e)`*) is an effect of DA (respectively).*

DA_{inv}, is defined to be the simple action with precondition equal to the conjunction of all propositions, p, such that `(over all p)` *is a condition of DA. It has an empty effect.*

Every conjunct in the condition of DA contributes to the precondition of precisely one of DA_{start}, DA_{end} or DA_{inv}. Every conjunct in the effect of DA contributes to the effect of precisely one of DA_{start} or DA_{end}. For convenience, DA_{start} (DA_{end}, DA_{inv}) will be used to refer to both the simple action and its name (respectively).

The duration condition *for DA, DC_{DA}, is the proposition in the duration field of DA, with terms being arithmetic expressions and* `?duration`.

Definition 17 Plans *A* plan, *P, with durative actions, for a planning instance, I, consists of a finite collection of* timed actions *which are pairs, each either of the form (t, a), where t is a rational-valued time and a is a simple action name – an action schema name together with the constants instantiating the arguments of the schema, or of the form $(t, a[t'])$, where t is a rational-valued time, a is a durative action name and t' is a non-negative rational-valued duration.*

Definition 18 Induced Simple Plan *If P is a plan then the* happening sequence *for P is $\{t_i\}_{i=0...k}$, the ordered sequence of time points formed from the set of times*

$$\{t \mid (t, a) \in P \; or \, (t, a[t']) \in P \; or \, (t - t', a[t']) \in P\}$$

The induced simple plan *for a plan P, simplify(P), is the set of pairs defined as follows:*

- *(t, a) for each $(t, a) \in P$ where a is a simple action name.*

- (t, a_{start}) *and* $(t + t', a_{end}[?\textbf{duration} := t'])$ *(this expression is a simple timed action – the square brackets denote substitution of* t' *for* `?duration` *in this case) for all pairs* $(t, a[t']) \in P$*, where* a *is a durative action name.*
- $((t_i + t_{i+1})/2, a_{inv})$ *for each pair* $(t, a[t']) \in P$ *and for each* i *such that* $t \leq t_i < t + t'$*, where* t_i *and* t_{i+1} *are in the happening sequence for* P*.*

The process of transforming a plan into a simple plan involves introducing actions to represent the end points of the intervals over which the durative actions in the plan are applicable. The complication to this process is that invariants cannot be associated with the end points, but must be checked throughout the interval. This is achieved by adding to the simple plan a collection of special actions responsible for checking the invariants. These actions are added between each pair of happenings in the original plan lying between the start and end point of the durative action. Because the semantics of simple plans requires that the preconditions of actions in the plan be satisfied, even though they might have no effects, the consequence of putting these monitoring actions into the simple plan is to ensure that the original plan is judged valid only if the invariants remain true, firstly, after the start of the durative action and, subsequently, after each happening that occurs throughout the duration of the durative action. One possibility is to make these monitoring actions occur at the same times as the updating actions, but this would require values to be accessed at the same time as they might be being updated, causing the plan to be invalidated by the "no moving targets" rule. In order to avoid this problem we interleave the monitoring actions with the updating actions by (arbitrarily) inserting them midway between pairs of successive happenings in the interval over which each durative action is executed.

At least two other solutions to the management of invariants are possible: one is to separately check the invariants in the trace for the simple plan generated using only the original simple actions and start and end actions for the durative actions in the plan. The second is to construct a semantics in which the states record their histories, rather than simply the current state of the world. In the latter case the end of a durative action could be executed as a special action that examines not only the current world state but also the history back to the start of the corresponding durative action. Both of these approaches offer advantages and disadvantages over the approach we have adopted. The key motivation for the approach we have taken is that the semantics rests, ultimately, on a state-transition model in a form that is essentially familiar to the classical planning community. That is, plans can be seen as recipes for state-transition sequences, with each state-transition being a function from the current state of the world to the next. Durative actions complicate this picture because they rely on a commitment, once a durative action has been started, to follow it through to completion. That commitment involves some sort of communication across the duration of the plan. The communication can be managed by structures outside the plan, that examine the trace, or by artificial modification of the plan itself to ensure that states carry extra information from the start to the end of the durative action. The latter approach has the disadvantage that as durative actions become more complex the artificial components that must be added to the plan become more intrusive. This becomes all the more apparent in the treatment of conditional effects that have conditions tested at the start of a durative action, or across its duration, but effects that are triggered at the end, since these cases require some sort of "memory" in the state to remember the tested conditions from the start of the durative action to the end point. These memory conditions allow us to avoid embedding an entire execution history in a state by substituting an *ad hoc* memory of the history for just those propositions and at just those times it is required.

Definition 19 Executability of a Plan *A plan, P (for a planning instance), is executable if simplify(P) is executable, producing the trace $\{S_i = (t_i, s_i, \mathbf{v}_i)\}_{i=0\ldots k}$, and, for each pair, $(t_{i+1}, a[t']) \in P$, $\mathcal{N}(DC_a)[?\mathtt{duration} := t', \mathbf{X} := \mathbf{v}_i]$ is satisfied.*

Note that the duration constraint must be evaluated in the state at the start of the durative action. This is a relatively arbitrary decision – evaluation of the duration constraint at the end of the durative action could also be justified.

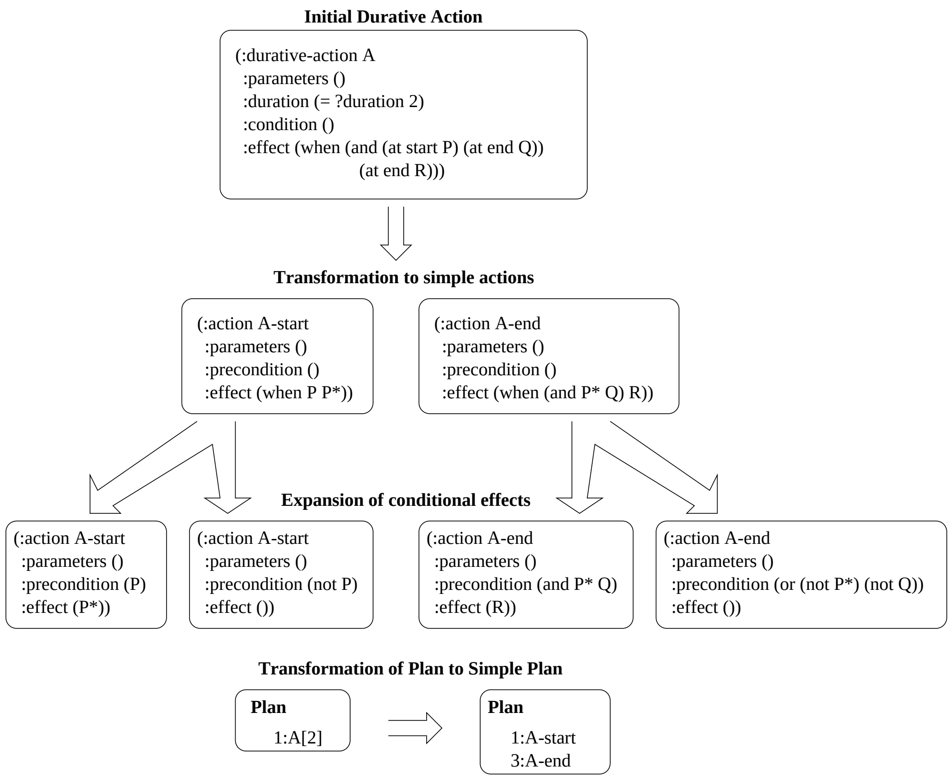
Definition 20 Validity of a Plan *A plan, P (for a planning instance), is valid if it is executable and if the goal specification is satisfied in the final state produced by its induced simple plan.*

8.1 Durative Actions with Conditional Effects

First, we observe that temporally annotated conditions and effects can be accumulated, because the temporal annotation distributes through logical conjunction. Therefore, we can convert conditional effects so that their conditions are simple conjunctions of at most one *at start* condition, at most one *at end* condition and at most one *over all* condition. Similarly, the effects can be reduced to a conjunction of at most one *at start* effect and at most one *at end* effect. Treatment of conditional effects then divides into three cases. The first case is very straightforward: any effect in a durative action of the form `(when (at t p) (at t q))`, where the condition and the effect bear the same single temporal annotation, can be transformed into a a simple conditional effect of the form `(when p q)` attached to the start or end simple action according to whether t is *start* or *end*. Since this case is straightforward we will not explicitly extend the previous definitions to cope with it. The second case is one in which the condition of a condition effect has *at start* conditions and the effect has *at end* effects.

Note that we consider conditional effects in which the effects occur at the start, but with conditions dependent on the state at the end or over the duration of the action, to be meaningless. This is because they reverse the expected behaviour of causality, where cause precedes effect. In any attempt to validate a plan by constructing a trace such reversed causality would be a huge problem, since we could not determine the initial effects of applying a durative action until we had seen what conditions held over the subsequent interval and conclusion of its activity, but, equally, we could not see what the effects of activity during the interval would be without seeing the initial effects of applying the durative action. This paradox is created by the opportunity for an action to change the past.

To handle this second case, then, we need to modify the state after the start of the durative action to "remember" whether the start conditions were satisfied and communicate this to the end of the durative action where it can then be simply looked up in the (then) current state to determine whether the conditional effect should be applied. We achieve this by applying a transformation to conditional effects of the form `(when (and (at start ps) (at end pe)) (at end q))` into a conditional effect added to the start simple action, `(when ps (`$\mathtt{M}_{ps}$`))`, and a conditional effect added to the end simple action, `(when (and pe (`$\mathtt{M}_{ps}$`)) q)`, where $\mathtt{M}_{ps}$ is a special new proposition, unique to the particular conditional effect of the particular application of the durative action being transformed. By ensuring that this proposition is unique in this way, there is no possibility of any other action in the plan interfering with it, so it represents an isolated memory of the fact that `ps` held in the state at which the durative action was started. If a conditional effect does not have *at*



Initial Durative Action
(:durative-action A :parameters () :duration (= ?duration 2) :condition () :effect (when (and (at start P) (at end Q)) (at end R)))
Transformation to simple actions
(:action A-start :parameters () :precondition () :effect (when P P*))
(:action A-end :parameters () :precondition () :effect (when (and P* Q) R))
Expansion of conditional effects
(:action A-start :parameters () :precondition (P) :effect (P*))
(:action A-start :parameters () :precondition (not P) :effect ())
(:action A-end :parameters () :precondition (and P* Q) :effect (R))
(:action A-end :parameters () :precondition (or (not P*) (not Q)) :effect ())
Transformation of Plan to Simple Plan
Plan
1:A[2]
Plan
1:A-start
3:A-end

Figure 5: Conversion of a durative action into level 2 actions and their grounded forms.

end conditions, the same transformation can be applied, simply ignoring `pe` in the previous discussion. Figure 6 shows the ambiguity that results from not remembering how a state, on the trajectory of a plan, was reached. The state $(P, Q, \neg R)$ needs to be visited following the application of *A-start* to the state $(P, Q, \neg R)$, at time 1, in order to reach (P, Q, R) after application of *A-end* at time 3. Figure 7 shows how a memory can be achieved using an additional proposition, $P*$, (denoted M_{ps} in the above discussion) and the ambiguity resolved.

The third, and final, case is where the durative action has conditional effects of the form `(when (and (at start ps) (over all pi) (at end pe)) (at end q))`. Again, if the effect has no *at start* or *at end* conditions the following transformation can be applied simply ignoring `ps` or `pe` as appropriate. In this case we need to construct a transformation that "remembers" not only whether `ps` held in the state at which the durative action is first applied, but also whether `pi` holds throughout the interval from the start to the end of the durative action. Unlike the invariants of durative actions, these conditions are not required to hold for the plan to be valid, but only determine what effects will occur at the end of the durative action. The idea is to use intervening monitoring actions, rather as we did for invariants in definition 18. This is achieved by adding a further effect to the start action: (M$_{pi}$). Then, the monitoring (simple) actions that are required have no precondition, but a single conditional effect: `(when (and (M`$_{pi}$`) (not pi)) (not (M`$_{pi}$`)))`. Once again, M$_{pi}$ is a special new proposition unique to the conditional effect for the application instance of the durative action being transformed. The monitoring actions are added at all the intermediate points that are used for the monitoring actions in definition 18. The same

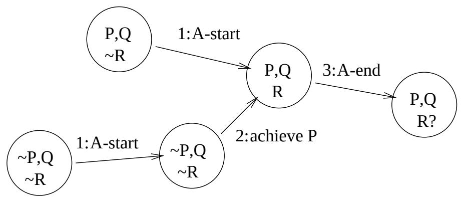


Figure 6: Flawed state space resulting from failure to record the path traversed when conditional effects span the interval of a durative action. The arc labelled *achieve P* indicates the possible application of some action that achieves the proposition P.

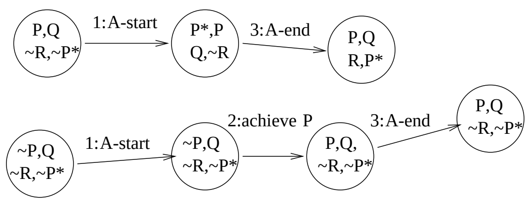


Figure 7: Correct state space showing use of "memory" proposition P*. The arc labelled *achieve P* indicates the possible application of some action that achieves the proposition P.

transformation used in the second case above is required again for the *at start* condition, `ps`, so `(when ps (`M_{ps}`))` is added as a conditional effect to the start simple action. Finally, we add a conditional effect to the end (simple) action: `(when (and (`M_{ps}`) (`M_{pi}`) pe) q)`. The effect of this machinery is to ensure that if the proposition `pi` becomes false at any time between the start and end of the durative action then M_{pi} will be deleted, but otherwise at the end of the durative action M_{ps} will hold precisely if `ps` held at the start of the action and M_{pi} will hold precisely if `pi` has held over the entire duration of the durative action. Therefore, the conditional effect of the end action achieves the intuitively correct behaviour of asserting its conditional effect precisely when the *at start* condition held at the start of the durative action, the *at end* condition holds at the end of the durative action and its *over all* condition has held throughout the duration of the action.

9 Level 4: Durative Actions with Continuous Effects

The objective of level 3 durative actions is to abstract out continuous change and concentrate on the end points of the period over which change takes place. In the syntax we have developed for expressing such abstractions we give the domain designer control over all of the key points during the execution of the action. In particular, a durative action models what conditions are invariant over the period of its execution, as well as the local pre- and post-conditions of the end-points.

Level 3 actions provide a slight increase in expressive power over level 2 because they can represent duration constraints that cannot be expressed in the level 2 syntax. Otherwise there is little additional expressive power (note that the semantics can be given by a polynomial time mapping of level 3 plans to level 2 plans) although there is significant syntactic convenience. However, when a plan needs to manage continuously changing values, as well as discretely changing ones, the durative action language and semantics need to be more powerful.

Level 4 durative actions can have continuous as well as discrete effects. These increase, or decrease, some numeric variable according to a specified rate of change over time for that variable. When determining how to achieve a goal a planner must be able to access the values of these continuous quantities at arbitrary points on the time-line of the plan. We use $\#t$ to refer to instants between happenings on the time-line. For example, to express the fact that the fuel level of a plane, $?p$, decreases continuously, as a function of the consumption rate of $?p$, we write:

```
            decrease (fuel-level ?p) (* #t (consumption-rate ?p))
```

This is importantly different from:

```
(at end (decrease (fuel-level ?p)
                              (* (flight-time ?a ?b) (consumption-rate ?p))))
```

because the latter is a single update happening at the end point of the flight action, whilst the former allows the correct calculation of the fuel level of the plane at any point in that interval. The former is a continuous effect, whilst the latter is a discrete one. Continuous effects are not temporally annotated because they can be evaluated at any time during the interval of the action.

In a level 4 plan it can happen that actions assigning to, consulting, and continuously modifying the same numeric variables, can happen concurrently. For example, in the flying and refuelling example given in Figure 8 it can be seen that the invariant condition that the fuel-level be greater than (or equal to) zero during the flight has to be maintained whilst the fuel is continuously decreasing. This could be expressed at level 3 by abstracting out the continuous decrease and making the final value available at the end point of the flight (see Figure 4). However, if a refuel operation happens during the flight time (in mid-air) then the fuel level after the flight will need to be calculated by taking into account both the continuous rate of consumption and the refuel operation. The level 3 action cannot calculate the fuel-level correctly because it uses the distance between the source and destination of the flight, together with the rate of consumption, to determine the final fuel level. In order to calculate the fuel level correctly it would be necessary to determine the time at which the refuel took place, and to use the remaining flight-time to calculate the fuel consumed. Level 3 actions do not give access to time points other than their own start and end points.

Of course, it can be argued that level 3 is able to express the desired combinations of flying and refuelling by providing additional durative actions, such as fly-and-refuel, that encapsulate all of the interactions just described and end up calculating the fuel level correctly. However, this approach requires more of the domain designer than it does of the planner – the domain designer must anticipate every useful combination of behaviours and ensure that appropriate encapsulations are provided.

By contrast with level 3, the level 4 action, in which the fuel consumption effect is given in terms of $\#t$, is powerful enough to express the fact that the mid-flight refuelling of the plane affects the final fuel level in a way consistent with maintaining the invariant of the fly action.

In Figure 9 the level 3 and level 4 actions of filling a bath are contrasted. The post-condition of the start of the period is that the tap is on. The condition that the tap be on and the plug be in is invariant, although the presence of the bath-filling agent is only a local precondition of the two end-points and is not invariant. The duration of the action is modelled by expressing the following duration constraint:

```
(:durative-action fly
     :parameters (?p - airplane ?a ?b - airport)
     :duration (= ?duration (flight-time ?a ?b))
     :condition (and (at start (at ?p ?a))
                     (over all (inflight ?p))
                     (over all (>= (fuel-level ?p) 0)))
     :effect (and (at start (not (at ?p ?a)))
                  (at start (inflight ?p))
                  (at end (not (inflight ?p)))
                  (at end (at ?p ?b))
                  (decrease (fuel-level ?p) (* #t (fuel-consumption-rate ?p)))))

(:action midair-refuel
     :parameters (?p)
     :precondition (inflight ?p)
     :effect (assign (fuel-level ?p) (fuel-capacity ?p)))
```

Figure 8: A level 4 durative action for flying.

```
(<= ?duration (/ (- capacity (level ?b)) flow))
```

and the effect at the end-point of the level 3 action is that the level of water in the bath is increased by `(* ?duration flow)` (where *flow* and *capacity* are domain constants). Duration constraints that express inequalities (such as the first expression above) are *duration inequalities* and are associated with a requirements flag (see Appendix).

The level 3 action models successful filling of a bath. The invariant condition ensures that the bath never overflows and the level of water at the end of the interval of execution is computed from the capacity of the bath and the flow-rate into the bath. If a plan attempts to add extra water to the bath, during the fill-bath interval then, provided that the concurrent action (empty-tank-into-bath) ends before the end of the fill-bath action, the invariant will be violated as soon as the capacity of the bath is exceeded and the plan will be invalidated. If empty-tank-into-bath ends simultaneously with the fill-bath action then the level of water in the bath will be able to be updated, since the updates are commutative, but the capacity will be exceeded and this will not be detected because the invariant is not checked beyond the end-point of the fill-bath action. The plan will be considered valid and the only way to prevent this is to encode the domain in such a way that empty-tank-into-bath cannot be performed while the fill-bath action is in progress. If a half-fill-bath action is encoded, which uses half-capacity of the bath to determine its duration, and the half-fill-bath action is performed in parallel with empty-tank-into-bath so that they end simultaneously, then the level of water at the simultaneous end-points can be correctly updated and the capacity will not be exceeded as long as the tank capacity is less than half of the bath capacity. The domain designer must be alert to these possibilities to ensure that only desirable plans are interpreted.

In the level 3 semantics we have exploited the fact that the only changes that can occur when a plan is executed are at points corresponding to the times of happenings, so the plan can be checked by looking at the activity focussed in this finite happening sequence. In fact, provided continuous effects are restricted to linear functions of time with only first order effects (so no continuous effects can affect functional expressions appearing as rvalues in another continuous effect), it is still possible at level 4 to restrict attention to the

```
(:durative-action fill-bath
   :parameters (?b)
   :duration (<= ?duration (/ (- capacity (level ?b)) flow))
   :condition (and (at start (plug-in ?b)) (at start (tap-off ?b))
                            (at start (in-bathroom))
                            (at end (in-bathroom))
                            (at end (plug-in ?b))
                            (at end (tap-on ?b))
                            (over all (plug-in ?b))
                            (over all (tap-on b?)))
                            (over all (< (level ?b) capacity)))
   :effect (and    (at start (tap-on ?b))
                   (at start (not (tap-off ?b)))
                   (at end (not (tap-on ?b)))
                   (at end (tap-off ?b))
                   (at end (increase (level ?b) (* ?duration flow)))
           )
)
```

Figure 9: A level 3 durative action for filling a bath.

```
(:durative-action fill-bath
   :parameters (?b)
   :duration (<= ?duration (/ (- capacity (level ?b)) flow))
   :condition (and (at start (plug-in ?b)) (at start (tap-off ?b))
                            (at start (in-bathroom))
                            (at end (in-bathroom))
                            (at end (plug-in ?b))
                            (at end (tap-on ?b))
                            (over all (plug-in ?b))
                            (over all (tap-on b?))
(over all (< (level ?b) capacity))
   :effect (and    (at start (tap-on ?b))
                   (at start (not (tap-off ?b)))
                   (at end (not (tap-on ?b)))
                   (at end (tap-off ?b))
                   (increase (level ?b) (* #t flow))))
```

Figure 10: A level 4 durative action for filling a bath.

happening sequence.

Despite the fact that arbitrary time points, within action intervals, are accessible to the planner, it is only necessary to gain access to numeric values at the start- and end-points of the actions in the plan that refer to them when checking a level 4 plan. These values are inaccessible at all other points. This is acceptable because level 4 does not allow the modelling of exogenous events so it is not necessary to take into account the exogenous activity of the environment in determining the validity of a plan.

As we discuss in [3], level 5 extends level 4 to enable the modelling of exogenous events initiated by the actions of the planning agent. When an agent acts in the world unintended consequences often arise from the actions performed. In some cases these can be taken advantage of and lead to improved quality plans. In other cases these unintended consequences must be avoided. Even simple domains, including the bath-filling domain, can exhibit exogenous events of this kind – initiating the filling of the bath can result in a flooded bathroom if the agent does not plan to be present in time to terminate the action.

Level 4 plans have the same syntactic structure as level 3 plans, except that actions from any of levels 1 to 4 can co-occur in a valid plan. The fact that level 4 actions effect continuous change is not visible in the final plan structure and is significant only during the validity-checking process.

10 Conclusions

This paper presents PDDL2.1 – an extension of PDDL allowing the modelling of temporal and metric effects. There are four levels: level 1 is PDDL as used in the STRIPS and ADL tracks of the AIPS-2000 competition; level 2 adds the ability to express numeric conditions and updates; level 3 introduces durative actions without continuous effects; level 4 extends level 3 to include continuous effects. All levels include conditional effects, quantified effects and negative preconditions. The key difference between levels 3 and 4 is that level 3 encapsulates continuous behaviour and discretizes it, so that the values of continuously changing variables are guaranteed correct only at the end points of actions. Level 4 allows continuous updating of numeric variables and access to their correct values at arbitrary points on the time-line of the plan.

The paper presents a BNF description of all four levels and a formal semantics for levels 2 and 3. Some observations are made about the interpretation of plans at level 4, but the formal semantics for level four is presented in a companion document [3]. In that document we also present level 5 and its semantics. Level 5 is an extension of level 4 that introduces additional modelling components and diverges from the durative actions direction. Although level 5 is not part of PDDL2.1, and will not be used in the competition, we present it in the hopes that it will encourage debate and widen the horizons of the planning community to include domains with mixed discrete and continuous behaviours.

Acknowledgements

We are grateful to Drew McDermott, David E. Smith, Fahiem Bacchus and Hector Geffner for many stimulating and helpful discussions about all levels of PDDL+. Their input was essential in helping us to identify the PDDL2.1 subset of PDDL+ and we would like to thank them for the time and effort they spent reading and commenting on our documents.

A BNF Description of PDDL2.1

Domain structures remain pretty much as specified in PDDL. The main alterations are to introduce a slightly modified syntax for numeric fluent expressions and to remove the syntax for hierarchical expansions and a couple of other features. This is not necessarily abandoned, but it will not be used in the competition in 2002 and has not, to the best of our knowledge, been used in any publically available planning systems or even domains.

```
<domain>            ::= (define (domain <name>)
                            [<require-def>]
                            [<types-def>]^:typing
                            [<constants-def>]
                            [<predicates-def>]
                            [<functions-def>]^:fluents
                            <structure-def>*)
<require-def>       ::= (:requirements <require-key>+)
<require-key>       ::= See Section A.4
<types-def>         ::= (:types <typed list (name)>)
<constants-def>     ::= (:constants <typed list (name)>)
<predicates-def>    ::= (:predicates <atomic formula skeleton>+)
<atomic formula skeleton>
                    ::= (<predicate> <typed list (variable)>)
<predicate>         ::= <name>
<variable>          ::= ?<name>
<atomic function skeleton>
                    ::= (<function-symbol> <typed list (variable)>)
<function-symbol>   ::= <name>
<functions-def>     ::=^:fluents (:functions <function typed list
                                      (atomic functor skeleton)>)
<structure-def>     ::= <action-def>
<structure-def>     ::=^:durations <durative-action-def>
```

A slight modification has been made to the type syntax – it is no longer possible to nest `either` expressions (a possibility that was never exploited, but complicates parsing). Numbers are no longer considered to be an implicit type – the extension to numbers is now handled only through functional expressions. This ensures that there are only finitely many ground action instances. A desirable consequence is that action selection choice points need never include choice over arbitrary numeric ranges. The use of finite ranges of integers for specifying actions is useful (see Mystery or FreeCell for example) and we are considering an extension of the standard syntax to allow for a more convenient representation of these cases. The syntax of function declarations allows functions to be declared with types. At present the syntax is restricted to number types, since we do not have a semantics for other functions, but the syntax offers scope for possible extension. Where types are not given for the function results they are assumed to be numbers.

```
<typed list (x)> ::= x*
<typed list (x)> ::=^:typing x+- <type> <typed list(x)>
<primitive-type> ::= <name>
```

```
<type>           ::= (either <primitive-type>⁺)
<type>           ::= <primitive-type>
```

```
<function typed list (x)> ::= x*
<function typed list (x)> ::=:typing x⁺- <function type>
                                       <function typed list(x)>
<function type>           ::= number
```

A.1 Actions

The BNF for an action definition is given below. Again, this has been simplified by removing generally unused constructs (mainly hierarchical expansions). It should be emphasised that this removal is not intended to be a permanent exclusion – hierarchical expansion syntax has proved a difficult element of the language both to agree on and to exploit. As the other levels of the language stabilise we hope to return to this layer and redevelop it.

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

Goal descriptions have been extended to include fluent expressions.

```
<GD>                   ::= <atomic formula(term)>
<GD>                   ::=:negative-preconditions <literal(term)>
<GD>                   ::= (and <GD>*)
<GD>                   ::=:disjunctive-preconditions (or <GD>*)
<GD>                   ::=:disjunctive-preconditions (not <GD>)
<GD>                   ::=:disjunctive-preconditions (imply <GD> <GD>)
<GD>                   ::=:existential-preconditions
                         (exists (<typed list(variable)>*) <GD> )
<GD>                   ::=:universal-preconditions
                         (forall (<typed list(variable)>*) <GD> )
<GD>                   ::=:fluents <f-comp>
<f-comp>               ::= (<binary-comp> <f-exp> <f-exp>)
<literal(t)>           ::= <atomic formula(t)>
<literal(t)>           ::= (not <atomic formula(t)>)
<atomic formula(t)>    ::= (<predicate> t*)
<term>                 ::= <name>
<term>                 ::= <variable>
<f-exp>                ::= <number>
<f-exp>                ::= (<binary-op> <f-exp> <f-exp>)
<f-exp>                ::= <f-head>
```

```
<f-head>            ::= (<function-symbol> <term>*)
<f-head>            ::= <function-symbol>
<binary-op>         ::= +
<binary-op>         ::= −
<binary-op>         ::= ∗
<binary-op>         ::= /
<binary-comp>       ::= >
<binary-comp>       ::= <
<binary-comp>       ::= =
<binary-comp>       ::= >=
<binary-comp>       ::= <=
<number>            ::= Any numeric literal
                          (integers and floats of form n.n).
```

Effects have been extended to include functional expression updates. The syntax proposed here is a little different to the syntax proposed in the earlier version of PDDL. The syntax of conditional effects proposed by Fahiem Bacchus for AIPS 2000 has been adopted, preventing the nesting of conditional effects. The assignment operators are prefix forms. Simple assignment is called `assign` (previously this was `change`) and operators corresponding to C update assignments, $+=$, $-=$, $*=$ and $/=$ are given the names `increase`, `decrease`, `scale-up` and `scale-down` respectively. The prefix form has been adopted in preference to an infix form in order to preserve consistency with the Lisp-like syntax and the non-C names to help the C and C++ programmers to remember that the operators are to be used in prefix form). We prefer `assign` to the original `change` because the introduction of `increase` and so on makes the nature of a *change* more ambiguous.

```
<effect>     ::= <a-effect>
<effect>     ::=:conditional−effects (forall (<variable>*) <effect>)
<effect>     ::=:conditional−effects (when <GD> <a-effect>)
<a-effect>   ::= (and <p-effect>*)
<a-effect>   ::= <p-effect>
<p-effect>   ::= (<assign-op> <f-head> <f-exp>)
<p-effect>   ::= (not <atomic formula(term)>)
<p-effect>   ::= <atomic formula(term)>
<p-effect>   ::=:fluents (<assign-op> <f-head> <f-exp>)
<assign-op>  ::= assign
<assign-op>  ::= scale-up
<assign-op>  ::= scale-down
<assign-op>  ::= increase
<assign-op>  ::= decrease
```

A.2 Durative Actions

Durative action syntax was one of the most contraversial areas of the extended syntax, partly because it is anticipated that it will, initially, be one of the most heavily used extensions.

```
<durative-action-def> ::= (:durative-action <da-symbol>
                              :parameters ( <typed list (variable)> )
```

```
                                        <da-def body>)
<da-symbol>             ::= <name>
<da-def body>           ::= :duration <duration-constraint>
                            :condition <da-GD>
                            :effect <da-effect>
```

The conditions under which a durative action can be executed are more complex than for standard actions, in that they specify more than the conditions that must hold at the point of execution. They also specify the conditions that must hold throughout the duration of the durative action and also at its termination. To distinguish these components we introduce a simple temporal qualifier for the preconditions. The use of the name "precondition" would be somewhat misleading given that the conditions described can include constraints on what must hold after the action has begun. This has motivated the adoption of `:condition` to describe the collection of constraints that must hold in order to successfully apply a durative action. The logical form of conditions for durative actions has been restricted to conjunctions of temporally annotated expressions, but there is clearly scope for future extension to allow more complex formulae.

```
<da-GD>                 ::= <timed-GD>
<da-GD>                 ::= (and <timed-GD>+)
<timed-GD>              ::= (at <time-specifier> <GD>)
<timed-GD>              ::= (over <interval> <GD>)
<time-specifier>        ::= start
<time-specifier>        ::= end
<interval>              ::= all
```

The duration (`?duration`) of a durative action can be specified to be equal to a given expression (which can be a function of numeric expressions), or else it can be constrained with inequalities. This latter allows for actions where the conclusion of the action can be freely determined by the executive without necessarily having further side-effects. For example, a walk between two locations could be made to take as long as the executive considered convenient, provided it was at least as long as the time taken to walk between the locations at the fastest walking speed possible. Constraints that do not specify the exact duration of a durative action might prove harder to handle, so we have introduced a label (`:duration-inequalities`) to signal that a domain makes use of them. A duration constraint is supplied to dictate or limit the temporal extent of the durative action. The duration is an implicit parameter of the durative action and must be supplied in a plan that uses durative actions. To denote this, a durative action is denoted in a plan by `t:(name arg1...argn)[d]` where `d` is the (rational valued) duration in floating point format ($n.n$).

```
<duration-constraint>   ::= :duration-inequalities
                            (and <duration-constraint>+)
<duration-constraint>   ::= (<d-op> ?duration <d-value>)
<d-op>                  ::=:duration-inequalities <=
<d-op>                  ::=:duration-inequalities >=
<d-op>                  ::= =
<d-value>               ::= <number>
```

```
<d-value>                  ::=:fluents <f-exp>
```

In addition to logical effects, which can occur at the start or end of a durative action, durative actions can have numeric effects that refer to the literal `?duration`. More sophisticated durative actions can also make use of functional expressions describing effects that occur over the duration of the action. This allows functional expressions to be updated by a continuous function of time, rather than only step functions.

```
<da-effect>      ::= (and <da-effect>*)
<da-effect>      ::= <timed-effect>
<da-effect>      ::=:conditional-effects (forall (<variable>*) <da-effect>)
<da-effect>      ::=:conditional-effects (when <da-GD> <timed-effect>)
<da-effect>      ::=:fluents (<assign-op> <f-head> <f-exp-da>)
<timed-effect>   ::= (at <time-specifier> <a-effect>)
<timed-effect>   ::= (at <time-specifier> <f-assign-da>)
<timed-effect>   ::=:continuous-effects (<assign-op-t> <f-head> <f-exp-t>)
<f-assign-da>    ::= (<assign-op> <f-head> <f-exp-da>)
<f-exp-da>       ::= (<binary-op> <f-exp-da> <f-exp-da>)
<f-exp-da>       ::=:duration-inequalities ?duration
<f-exp-da>       ::= <f-exp>
```

Note that the `?duration` term can only be used to define functional expression updating effects if the duration constraints requirement is set. This is because in other cases the duration value is available as an expression in any case, where as in the case where duration constraints are offered the duration can, in some cases, be freely selected within constrained boundaries.

```
<assign-op-t>      ::= increase
<assign-op-t>      ::= decrease
<f-exp-t>          ::= (* <f-exp> #t)
<f-exp-t>          ::= (* #t <f-exp>)
<f-exp-t>          ::= #t
```

The symbol $\#t$ is used to represent the period of time that a given durative action has been active. It is therefore a *local* clock value for each duration, independent of similar clocks for each other duration. There has been discussion with members of the committee about the use of the expression using $\#t$: it was proposed that an expression declaring the rate of change alone could be used. This was decided against on the grounds that the assertion of a rate of change suggests that the rate of change is determined by one process effect alone. In fact, it is intended that if multiple active processes affect the same fluent then these effects are accumulated. Using the expression that directly defines the amount by which each process contributes to the change in a fluent value over time we do not appear to assert (inconsistently) that a fluent has multiple simultaneous rates of change.

A.3 Problems

Planning problems specifications have been modified to exclude several generally unused constructs (named initial situations and expansion information). We have deprecated the length specification because it is at odds with the intention to supply physics, not advice.

Furthermore, the advice this field offers over-emphasises a very coarse plan metric. Instead, we have introduced an optional metric field, which can be used to supply an expression that should be optimized in the construction of a plan. The field states whether the metric is to be minimized or maximized. Of course, a planner is free to ignore this field and make the assumption that plans with fewest steps will be considered a good plan. However, we consider that this extension is a crucial one in the development of a more widely applicable planning language. We have provided the variable `total-time` that takes the value of the total execution time for the plan. This allows us to conveniently express the intention to minimize total execution time. We anticipate that extensions of the plan metric syntax will prove necessary in the longer term, but believe that this version already provides a significant new challenge to the community. Problem specifications are still somewhat impoverished in terms of the ability to easily specify temporal constraints on goals and other non-standard features of initial and goal states. Again, we anticipate the need for extension, but have chosen to leave a clean sheet for future developments.

```
<problem>            ::= (define (problem <name>)
                                (:domain <name>)
                                [<require-def>]
                                [<object declaration> ]
                                <init>
                                <goal>+
                                [<metric-spec>]
                                [<length-spec> ])
<object declaration> ::= (:objects <typed list (name)>)
<init>               ::= (:init <init-el>+)
<init-el>            ::= <literal(name)>
<init-el>            ::=:fluents (= <f-head> <number>)
<goal>               ::= (:goal <GD>)
<metric-spec>        ::= (:metric <optimization> <ground-f-exp>)
<optimization>       ::= minimize
<optimization>       ::= maximize
<ground-f-exp>       ::= (<binary-op> <ground-f-exp> <ground-f-exp>)
<ground-f-exp>       ::= <number>
<ground-f-exp>       ::= (<functor> <name>*)
<ground-f-exp>       ::= total-time
<length-spec>        ::= (:length [(:serial <integer>)]
                              [(:parallel <integer>)])
                            The length-spec is deprecated.
```

A.4 Requirements

Here is a table of all requirements in PDDL2.1. Some requirements imply others; some are abbreviations for common sets of requirements. If a domain stipulates no requirements, it is assumed to declare a requirement for `:strips`.

Requirement	*Description*
`:strips`	Basic STRIPS-style adds and deletes
`:typing`	Allow type names in declarations of variables
`:negative-preconditions`	Allow `not` in goal descriptions
`:disjunctive-preconditions`	Allow `or` in goal descriptions
`:equality`	Support `=` as built-in predicate
`:existential-preconditions`	Allow `exists` in goal descriptions
`:universal-preconditions`	Allow `forall` in goal descriptions
`:quantified-preconditions`	= `:existential-preconditions` + `:universal-preconditions`
`:conditional-effects`	Allow `when` in action effects
`:fluents`	Allow function definitions and use of effects using assignment operators and arithmetic preconditions.
`:adl`	= `:strips` + `:typing` + `:negative-preconditions` + `:disjunctive-preconditions` + `:equality` + `:quantified-preconditions` + `:conditional-effects`
`:durations`	Allows durative actions. Note that this does not imply `:fluents`.
`:duration-inequalities`	Allows duration constraints in durative actions using inequalities.
`:continuous-effects`	Allows durative actions to affect fluents continuously over the duration of the actions.

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] A. Blum and M. Furst. Fast Planning through Plan-graph Analysis. In *IJCAI*, 1995.

[3] Maria Fox and Derek Long. PDDL+ : Planning with time and metric resources. Technical report, University of Durham, UK, 2001.

[4] B. Gazen and C. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *ECP-97*, pages 221–233, 1997.

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

[6] D. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2), 2000.

[7] D. McDermott and the AIPS'98 Planning Competition Committee. PDDL–the planning domain definition language. Technical report, Available at: www.cs.yale.edu/homes/dvm, 1998.

[8] Edwin Peter Dawson Pednault. Adl: Exploring the middle ground between Strips and the situation calculus. In *Proc. Conf. on Knowledge Representation and Reasoning* **1**, pages 324–332, 1989.

[9] J. Scott Penberthy. Planning with Continuous Change. Technical Report 93-12-01, University of Washington Department of Computer Science & Engineering, 1993.

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