The Formal Semantics of Processes in PDDL

Drew V. McDermott
Yale University
Computer Science Department
drew.mcdermott@yale.edu

Abstract

One reason why autonomous processes have not been officially incorporated into PDDL is that there is no agreed-upon semantics for them. I propose a semantics with interpretations grounded on a branching time structure. A model for a PDDL domain is then simply an interpretation that makes all its axioms, action definitions, and process definitions true. In particular, a process definition is true if and only if over every interval in which its condition is true, the process is active and all its effects occur at the right times.

The Semantics Problem for PDDL

In (Fox & Long 2001a), Maria Fox and Derek Long proposed extensions to PDDL for describing *processes*, that is, activities that could go on independent of what the executor of plans did. They also described a semantics for processes in terms of hybrid automata (Henzinger 1996). Their proposal was not adopted by the rules committee for the third International Planning Competition, mainly because most of the members preferred to focus on the narrower extension to "durative actions," actions that took a nonzero amount of time. Another potential reason is that the semantics of processes were so complex that no one really understood them.

A semantics for all of PDDL is hard to lay out, for a couple of reasons. One is that the requirement-flag system turns it into a family of languages with quite different behavior. For instance, the `:open-world` flag transforms the language from one in which `not` is handled via "negation as failure" into god-knows-what. To sidestep this complexity, I will make a variety of simplifying omissions and deliberate oversights:

- ◇ I will ignore the possibility of action `:expansion`s.
- ◇ There will be no formalization of the fact that propositions persist in truth value until some event changes them.
- ◇ `:vars` fields in action and process definitions will be left out.
- ◇ I will assume effects are simple conjunctions of literals. There are no secondary preconditions ("`:when`s") or universally quantified effects.
- ◇ I won't discuss durative actions. They can be defined in terms of processes, as discussed by (Fox & Long 2001b) and (McDermott 2003).
- ◇ Fox and Long (Fox & Long 2001b) allow for "events," which occur autonomously, like processes, but are instantaneous, like primitive actions. I won't discuss these, although they present no particular problem.
- ◇ Except for the presence of autonomous processes, we'll stay in the "classical milieu," and in particular assume that the planner knows everything about the initial situation and the consequences of events and processes. There's no nondeterminism, and no reason to use sensors to query the world.

The Structure of Time

It's fairly traditional (McDermott 1982; 1985) to think of an action term as denoting a set of intervals; intuitively, these are the intervals over which the action occurs. To make this precise, we have to specify what we mean by "interval." A *date* is a pair $\langle r, i\rangle$, where r is a nonnegative real number and i is a natural number. A *situation* is a mapping from propositions to truth values. A *date range function* is a function from real numbers to natural numbers. A *situation continuum* is a pair $\langle C, h\rangle$, where h is a date range function and C is a *timeline*, that is, a mapping from

$$\{\langle r, i\rangle \mid r \in \textit{nonnegative reals and } 0 \leq i < h(r)\}$$

to situations. If $d_1 = \langle r_1, i_1\rangle$ and $d_2 = \langle r_2, i_2\rangle$, then $d_1 < d_2$ if $r_1 < r_2$ or $r_1 = r_2$ and $i_1 < i_2$. A date $\langle r, d\rangle$ is *in* $\langle C, h\rangle$ if $r \geq 0$ and $0 \leq d < h(r)$. A situation s is *in* $\langle C, h\rangle$ if there exists a date d in $\langle C, h\rangle$ such that $C(d) = s$.

The intuitive meaning of these definitions is that at any point r in time there can be a series of zero or more actions taken by the *target agent* (i.e., the hypothetical agent that executes plans). The range function says how many actions are actually taken at point r in that continuum. Each action takes an infinitesimal amount of time, so that "just after" time r there can be an arbitrary number $h(r)$ of actions that precede all time points with times $r' > r$. (This picture is reminiscent of nonstandard analysis (Robinson 1979).)

Using this framework, a purely classical plan might be specified by giving an action $A(i)$ for all dates $\langle 0, i\rangle$ where $i < h(0)$, and $h(0)$ is the length of the plan. The plan is feasible if $A(0)$ is feasible in the initial situation, and $A(i+1)$ is feasible in $C(0, i)$. The plan achieves a goal G if G is true in $C(0, h(0))$. It may sound odd to imagine the entire

plan being executed in zero time, but time is not a factor in classical planning, so a plan might as well take no time at all to execute, or $h(0) \times dt$ if you prefer. In what follows, I will use the dt notation to denote an infinitesimal time interval, i.e., the time from $\langle r, i\rangle$ to $\langle r, i+1\rangle$. If date $d = \langle r, i\rangle$, then $d + dt = \langle r, i+1\rangle$.

If we broaden our ontology to allow for autonomous processes, then plans can have steps such as "turn on the faucet," and "wait until the bucket is full." We can use situation continua to model bursts of instantaneous "classical" actions separated by periods of waiting while processes run their course. A plan may even call for the target agent to do nothing, but simply wait for processes that are active in the initial situation to solve a problem. In that case $h(r) = 0$ for all r.

A *closed situation interval* in continuum $\langle C, h\rangle$ is a pair of dates $[d_1, d_2]_{\langle C,h\rangle}$ with d_1 and d_2 both in $\langle C, h\rangle$. (The subscript will be omitted when it is obvious which continuum we're talking about.) Informally, the interval denotes the set of all situations in $\langle C, h\rangle$ with dates d such that $d_1 \leq d \leq d_2$. The interval is *nontrivial* if $d_1 \leq d_2$. We also have *open* and *half-open* situation intervals defined and written in exact analogy to the usual concepts of open and half-open intervals on the reals.

A Notation for Processes

Fox and Long, in (Fox & Long 2001a), suggested a notation for autonomous processes. I propose a slightly different one, which I believe is clearer and somewhat easier to formalize. A process is declared using a syntax similar to that for actions. (This notation is part of an overall reform and extension of PDDL called Opt. The Opt Manual (McDermott 2003) lays out the whole language and describes other features of processes that I've omitted in this discussion.)

```
<process-def>
         ::= (:process <name>
         :parameters
             <typed list (variable)>
         <process-def body>)
<process-def body>
         ::=
         [:condition <goal proposition>]
         [:start-effect <effect proposition>]
         [:effect <effect proposition>]
         [:stop-effect <effect proposition>]
```

The syntax `<typed list (variable)>` refers to the PDDL notation exemplified by `(x y - Location r - Truck)`, although here again the full Opt language allows several extensions.

The informal semantics of a process are simple: Whenever the `:condition` of the process is true, the process becomes active. It has immediate effects spelled out by its `:start-effect` field and the `:effect` field. These take time dt. The start-effects occur once, but the *through-effects*, i.e., those specified by the `:effect` field, remain true as time passes. As soon as the process's condition becomes false, the through-effects become false and the stop-effects (described by the `:stop-effects` field) occur, again taking time dt.

The contents of `:effect` field of a process definition is a conjunction of one or more propositions of the form

```
(derivative $x$ $d$)
```

where x and d are numerical fluents, that is, objects of type `(Fluent Number)`.[1] So for a process's effects to remain true is for various quantities to change at the given rates. The rates are not constants in general, so we can express any set of differential equations in this notation.

A formal specification of the semantics of processes is given by detailing the *truth conditions* of a process definition, that is, the constraints mandated by the definition on situation intervals over which the process is *active*. This specification must fit with an overall semantics for plans and facts.

I said above that a plan might be described by listing the times when actions occur, but I'm actually going to describe plans in terms of the method language of Opt. The full method language is rather different from PDDL's, but I'm going to stick to a subset that is close to what PDDL allows. The grammar of plans is simply

```
P ::= A
      | (seq P1 ...Pn)
      | (parallel P1 ...Pn)
A ::= action term
      | (wait-while p)
      | (wait-for q p)
```

where p is a process term and q is an inequality that p presumably affects. An example plan is

```
Plan A =
   (seq (parallel (turn-on faucet1)
                  (plug-up outlet1))
        (wait-for (>= (level tub1)
                      (cm 30))
                  (filling tub1)))
```

Here `filling` is defined as a process thus:

```
(:process filling
    :parameters (b - Tub)
    :vars (f - Faucet l - Outlet)
    :condition (and (faucet-of f b)
                    (outlet-of l b)
                    (faucet-on f))
    :effect
       (and (when (plugged-up l)
                  (derivative
                     (level b)
                     (constant (cm/sec 2))))
            (when (not (plugged-up l))
                  (derivative
                     (level b)
                     (constant (cm/sec 1))))))
```

and the actions are defined thus:

[1] I capitalize types such as `Number` and type functions such as `Fluent`.

```
(:action turn-on
    :parameters (f - Faucet)
    :effect (faucet-on f))

(:action plug-up
    :parameters (l - outlet)
    :effect (plugged-up l))
```

Obviously, these are all simplified for expository purposes. One apparent simplification is that we provide no method for the target agent to test whether the level in `tub1` has reached 30 cm. But in our classical framework, there is no need for such a method; the agent knows exactly when the level will get to that point.

What we want our formal semantics to tell us is that Plan A is executed over any situation interval $[\langle r_1, i_1\rangle, \langle r_2, 0\rangle]$ in which `(turn-on faucet1)` and `(plug-up outlet1)` occur at times $\langle r_1, i_1\rangle$ and $\langle r_1, i_1 + 1\rangle$ (in either order), and the level of `tub1` is < 30 cm at time r_1 and reaches 30 cm at time r_2. It is also executed over any situation interval $[\langle r_1, i_1\rangle, \langle r_1, i_1 + 2\rangle]$, where the `turn-on` and `plug-up` actions are executed as before, and where `(level tub1)` $\geq$ 30 cm at r_1.

The Formal Semantics

Interpretations of Domains with Processes

An *interpretation* of a domain D is a tuple $\langle U_0, T, I_0\rangle$, where U_0 is a function from type symbols in D to sets of objects called *subuniverses*; T is a set of situation continua; and I_0 is a function from the non-type symbols of D to objects in the subuniverses. If $I_0(s) = v$, and s has type τ, then it must be the case that $v \in U_0(\tau)$.

U_0 must obey the following constraints:

$U_0(\texttt{Void}) = \emptyset$

$U_0(\texttt{Boolean}) = \{\textit{true}, \textit{false}\}$

$U_0(\texttt{Integer})$ = the set of all integers

$U_0(\texttt{Number})$ = the set of all real numbers. I treat integers as a subset of the reals, not an alternative data type

$U_0(\texttt{Situation}) =$ $\{s \mid$ *For some* $\langle C, h\rangle \in T$ *and date* $d, C(d) = s\}$

We want to extend U_0 to a function U that gives the meaning of all type expressions, and I_0 to a function I giving the meaning of all formulas and terms. To extend U_0, we need the notion of a *tuple*, of which Opt distinguishes two varieties, `Tup`-tuples and `Arg`-tuples. The former are like lists in Lisp, the latter like ordered n-tuples in mathematics. The difference is that `Tup`-tuples can be of any length, including 0 and 1, while `Arg`-tuples have to have length at least 2. An `Arg`-tuple $\langle x\rangle$ of length 1 is identical to x. An empty `Arg`-tuple is impossible, so we identify the type `(Arg)` with `Void`, the empty type. In this paper we need only `Arg`-tuples, so I concentrate on those.

Both kinds of tuples have designators with named fields, as in

```
(Arg num - Integer &rest strings - String)
```

but in the `Arg` case the names are there only because an `Arg` expression often does double duty as defining a tuple and declaring local variables in an action or process. The subuniverse denoted by `(Arg num - Integer &rest strings - String)` is

$\{\langle i, s_1, \ldots, s_n\rangle \mid i$ *is an integer and each* s_j *is a string*$\}$

which, not surprisingly, is the type of arguments to a an action declared thus:

```
(:action name
    :parameters (num -  Integer
                 &rest strings - String)
    ...)
```

The labels are simply ignored when determining the denotation of the type. We can replace each label with a "don't care" symbol ("_") or omit them entirely. Note that if τ is a type, $U(\texttt{(Arg } \tau\texttt{)}) = U(\tau)$. (I will use the term `Arg`-type for expressions such as the one following the keyword `:parameters` even though the `Arg` flag is missing.)

We extend U to tuples by making $U(\texttt{(Tup ...)})$ denote a `Tup`-tuple and $U(\texttt{(Arg ...)})$ denote an `Arg`-tuple as suggested by these examples. I won't try to fill in the messy details.

Using `Arg`-tuples, we can give a meaning to the type notation `(Fun` τ_r `<-` τ_d`)` of functions from domain type τ_d to range type τ_r. The domain type τ_d is in general an `Arg`-tuple, and $U(\texttt{(Fun } \tau_r \texttt{ <- } \tau_d\texttt{)}) = U(\tau_d) \otimes U(\tau_r)$.

The type `(Fluent` τ`)` is an abbreviation for `(Fun` τ `<- Situation)`. `Prop`, for "proposition," is an abbreviation for `(Fluent Boolean)`. Predicates have types of the form `(Fun Prop <- ...)`.

Recall our intuition that action and process terms denote sets of intervals. For this to be the case, the *subuniverse* that the denotation resides in must be the *powerset* of a set of intervals, written *pow*(S).

There are four kinds of event type, and hence four types of sets of interval sets to contemplate: no-ops, skips, hops, and slides:[2]

1. `Skip`: The type of all actions that take one infinitesimal time unit:

$$U(\texttt{Skip}) = \textit{pow}(\{[\langle r, d\rangle, \langle r, d+1\rangle]_{\langle C,h\rangle} \mid \langle C, h\rangle \in T \textit{ and } d + 1 \leq h(r)\}) = \textit{pow}(\{[d, d + dt]_{\langle C,h\rangle} \mid \langle C, h\rangle \in T\})$$

2. `Hop`: The type of all actions that take more time than a `Skip`:

$$U(\texttt{Hop}) = \textit{pow}(\{[\langle d_1, d_2\rangle]_{\langle C,h\rangle} \mid \langle C, h\rangle \in T,\ d_1, d_2 \textit{ are in } \langle C, h\rangle, \textit{ and there is a } d \textit{ in } \langle C, h\rangle \textit{ such that } d_1 < d < d_2\})$$

[2] In the full Opt implementation, events can have values as well as effects, so we can distinguish, say, `(Skip Integer)` from `(Skip String)`. What I write as `Skip` in this paper would actually be written `(Skip Void)` in Opt, meaning a `Skip` that returns no value.

3. `no-op`: A constant whose value is the only element of subuniverse

   $$U(\texttt{(Con no-op)}) = \{\{[\texttt{d},\texttt{d}]_{\langle \texttt{C},\texttt{h}\rangle} \mid \texttt{d} \textit{ is in } \langle \texttt{C},\texttt{h}\rangle \in \texttt{T}\}\}$$

   that is, the singleton set whose only element is the set of all zero-duration closed situation intervals in T. The name of this type is `(Con no-op)`. Unlike the others, this subuniverse is not the powerset of anything.

   I define $U(\texttt{Step})$ to be $U(\texttt{(Con no-op)}) \cup U(\texttt{Skip}) \cup U(\texttt{Hop})$.

4. `Slide`: The type of autonomous processes. A process must take noninfinitesimal time, so

   $$U(\texttt{Slide}) = \textit{pow}(\{[\langle r_1, i_1\rangle, \langle r_2, i_2\rangle]_{\langle C,h\rangle} \mid \langle C,h\rangle \in T \textit{ and } r_1 < r_2\})$$

We also assume that every expression in Opt is typed. In all our examples imagine a superscript giving the type of the expression, with the proviso that all the type labels are consistent. For instance, the formula

```
(parallel (turn-on faucet1)
          (plug-up outlet1))
```

with type annotations would be

```
(parallel(Fun Action <- (Arg &rest Action))
    (turn-on(Fun Action <- Faucet)
        faucet1Faucet)Action
    (plug-up(Fun Action <- Outlet)
        outlet1Outlet)Action)Action
```

The annotations in this example are consistent because whenever f is labeled `(Fun` τ `<-` α`)`, then in `(`f a`)` a is of type α and `(`f a`)` is of type τ. The proper-typing requirementavoids absurd formulas like `(if 3 (= (not "a")))`. In the implementation, finding a consistent typing is a mostly automatic process, which is not relevant to this paper.

We can now state very simply how to extend I_0 to I, which assigns a denotation to every (properly typed) term of the language. I takes two arguments, a term and an *environment*, which is a total function from variables to ordered pairs $\langle v, y\rangle$, where y is a subuniverse.

- If s is a symbol, $I(s^\tau, E) = I_0(s) \in U(\tau)$.
- If x is a variable, $I(x^\tau, E) = v$ iff $E(x) = \langle v, U(\tau)\rangle$.
- For a functional term,

  $I((f^{\texttt{(Fun } \tau \texttt{ <- } \alpha\texttt{)}} a_1^{\alpha_1} \ldots a_n^{\alpha_n})^\tau, E) = v$
  iff
  $\langle\langle I(a_1^{\alpha_1}, E), \ldots I(a_n^{\alpha_n}, E)\rangle, v\rangle$
  $\in I(f^{\texttt{(Fun } \tau \texttt{ <- } \alpha\texttt{)}}, E)$
  (which is possible only if
  $U(\alpha_1) \otimes \ldots \otimes U(\alpha_n) \subseteq U(\alpha)$)

Because f can be an arbitrary function symbol, we don't need special rules to give the meanings of `(if` p q`)`, `(=` a b`)`, and such. We just need to stipulate that some symbols have the same meaning in all interpretations:

- $I_0(\texttt{if}) = (\lambda\,(x,y)(\lambda\,(s)\; x(s) = \textit{false or } y(s) = \textit{true}))$
- $I_0(\texttt{=}) = (\lambda\,(x,y)(\lambda(s)\; x = y))$
- ... and so forth

In general, the type of a predicate symbol P is `(Fun Prop <-` α`)` for some α, recalling that `Prop` is the type of functions from situations to booleans. We can take `if` to be just another predicate, whose argument type is `(Arg Prop Prop)`. $I(\texttt{if})$ must then be a function from situations to Booleans, which yields true when its first argument yields *false* or its second yields *true* in that situation. Although I use λ here, it's just a shorthand for a set of ordered pairs. I could have said $\{\langle\langle x, y\rangle, \{\langle s, b\rangle \mid b = \textit{true iff } x(s) = \textit{false or } y(s) = \textit{true}\}\rangle\}$

The denotation of "`=`" is a *constant* function on situations; two objects are equal only if they are always equal. The equality tester that tests whether two fluents have the same value in a situation is called `fl=`, with denotation

- $I_0(\texttt{fl=}) = (\lambda\,(f_1, f_2)(\lambda\,(s)\; f_1(s) = f_2(s)))$

The denotation of an action term or a process term must be a set of intervals:

For every primitive action function f, that is, every symbol f defined by an `:action` definition $I_0(f)$ must be of type `(Fun Skip <-` α`)`.
For every process function f, that is, every symbol f defined by a `:process` definition, $I_0(f)$ must be of type `(Fun Slide <-` α`)`.

In both cases, α is the `Arg` type from the `:parameters` of the definition.

I'll deal with domain-dependent functions in a later section. The meanings of `seq` and `parallel` are defined in every domain thus:

- $I_0(\texttt{seq})$

  $= (\lambda\,(a_1, \ldots, a_n)$
  $\{[d_0, d_e]_{\langle C,h\rangle}$
  $\mid \langle C,h\rangle \in T$
  and there exist dates $d_1, \ldots, d_n$
  such that for $j = 1, \ldots, n$,
  $[d_{j-1}, d_j] \in a_j$
  and $d_n = d_e\})$

- $I_0(\texttt{parallel})$

  $= (\lambda\,(a_1, \ldots, a_n)$
  $\{[d_b, d_e]_{\langle C,h\rangle}$
  $\mid \langle C,h\rangle \in T$
  there is a function $s : [1, \ldots, n]$
  $\rightarrow$ *situation intervals*,
  such that for $j = 1, \ldots, n$,
  $s(j) = [d_{j1}, d_{j2}]_{\langle C,h\rangle} \in a_j$
  and $d_b \le d_{j1} \le d_{j2} \le d_e$
  and for some $j_b, j_e \in [1, \ldots, n]$,
  $s(j_b) = [d_b, d_{j_b 2}]$
  and $s(j_e) = [d_{j_e 1}, d_e]$)

Another symbol that must have a standard meaning is `derivative`:

- $U_0(\texttt{derivative})$

  $= U(\texttt{(Fun (Fluent Number) <- (Fluent Number))})$

- I_0(`derivative`) =

  $(\lambda\,(f)$
  $\quad(\lambda\,(s)$ *if there is a* d *such that*
  $\qquad$(*for all* (r, i, C, h)
  $\qquad\quad$ *if* $\langle C, h\rangle \in T,\ s = C(r, i),\ h(r) = i$
  $\qquad\qquad$ *and there is an* r'
  $\qquad\qquad\quad$ *such that* $r < r'$
  $\qquad\qquad\qquad$ *and for all* $r'',\ r < r'' < r'$
  $\qquad\qquad\qquad\quad h(r'') = 0$
  $\qquad\qquad$ *then* $f^+(C)(r) = d$),
  $\qquad$ *then* d
  $\qquad$ *else* 0))

  where $f^+(C)$ is the "right derivative" of f in timeline C measured at time r, that is, the function of time whose value at t is the limit as $\Delta t \to 0$ of

$$\frac{f(C)(r + \Delta t) - f(C)(r)}{\Delta t}$$

This will require a bit of explanation. The same situation can occur in more than one continuum of T. So we first define the derivative at a "situation occurrence," that is, a date $\langle r, i\rangle_{\langle C,h\rangle}$. The derivative can be meaningfully defined only if there is an open interval after $\langle r, i\rangle$ such that no discrete events occur during that interval — i.e., $h(r) = i$ and $h(r'') = 0$ for all times r'' in that interval. In that case $(\lambda\,(t)\ f(C(t, 0)))$ is an ordinary function of time over that interval, which may have a derivative. The derivative of that function we denote by $f^+(C)$.

The remaining detail to fill in is to switch from situation occurrences to situations by requiring $f^+(C)(r)$ to have the same value for all occurrences of $C(r, i)$. Actually, it might be reasonable simply to require that this be the case, because it follows from the *Markov property,* that what happens starting in a situation depends only on what's true in that situation, assuming the target agent doesn't interfere. For now, I don't impose that requirement, but it will usually follow trivially from the axioms of a domain.

Important note: Even though the numerical value of the derivative in s is defined in terms of timelines in which no actions happen, the derivative still has a value at a date $\langle r, i\rangle$ in a timeline $\langle C, h\rangle$ in which an action, or even a series of actions, occurs at $\langle r, i\rangle$, so long as $C(r, i) = s$. You can think of (`derivative` f) as the rate at which the quantity f would change starting at s if it were undisturbed. In some timelines, the disturbance may be sufficient to cause the derivative to disappear Δt after the current situation, that is, at $C(r, i + 1)$, but it is still well defined at s.

Truth and Models

As usual, we want to define a model to be an interpretation of a domain's language that makes all its axioms true. Because propositions change truth value from situation to situation, we must amend that to: A *model* of a PDDL domain is an interpretation $\langle U_0, T, I_0\rangle$ that makes all axioms true in all situations and environments.

For ordinary axioms, we simply translate an axiom A from the `:axiom` syntax of PDDL to the traditional syntax,[3] yielding A', and then test whether $I(A', E)(s)$ is true for all environments E and situtations s in $\langle C, h\rangle$. We use the traditional specification of the interpretation of quantified formulas:

- I((`forall` $(x$ `-` $\alpha)\ P^{\texttt{Prop}}), E)$ = p, an object from subuniverse

  situations $(T) \otimes \{$*true,false*$\}$

  such that $p(s)$ = *true* if and only if $I(P, E')(s)$ = *true* for every environment E' that differs from E, if at all, only in assigning a different value, drawn from $U(\alpha)$, to variable x. (That is, $E'(x^\alpha) = \langle v, U(\alpha)\rangle$ for some $v \in U(\alpha)$.)

- Dual formula for `exists` left as an exercise for the reader.

In addition to axioms, we must require that an interpretation make action and process definitions true. I is extended to an action definition thus:

- I((`:action` $a^{\texttt{Fun Skip <-}\ \alpha}$
  `:parameters` r^α
  `:precondition` $p^{\texttt{Prop}}$
  `:effect` $e^{\texttt{Prop}}$),
  E)
  = *true*
  if and only if
  for all $\langle C, h\rangle \in T$,
  and every E' that differs from E,
  if at all, only in the assignments
  of the variables in r,
  and all $d_1 = \langle r, i\rangle$ and $d_2 = \langle r, i + 1\rangle$ in $\langle C, h\rangle$,
  if $I(p^{\texttt{Prop}}, E')(C(r, i))$ = *true*
  and $\langle I(r^\alpha, E'), [d_1, d_2]_{\langle C,h\rangle}\rangle \in I_0(a)$
  then $I(e^{\texttt{Prop}}, E')(C(r, i + 1))$ = *true*

In other words, the action definition is true if whenever I says the action occurs and that its preconditions are true, the effect is true. In this definition, α is an `Arg` type, and $I(r^\alpha, E')$ refers to an instance of the `Arg`-tuple obtained by substituting its variables as specified by E'.

We also need the following condition

- For all pairs of action terms $a_1^{\texttt{Skip}} \neq a_2^{\texttt{Skip}}$, all timelines $\langle C, h\rangle \in T$, and all environments E_1 and E_2, if $[d_1, d_1 + dt]_{\langle C,h\rangle} \in I(a_1, E_1)$ and $[d_2, d_2 + dt]_{\langle C,h\rangle} \in I(a_2, E_2)$, then $d_1 \neq d_2$. In other words, no two actions occur over precisely the same infinitesimal ("skip") interval.

This condition enforces an *interleaving* interpretation of concurrency. Two actions can occur at the same time, but they must still be ordered.

The truth condition on process definitions relies on the following definition:

With respect to an interpretation $\langle U, T, I\rangle$
and an environment E,
an interval $[d_1, d_2]_{\langle C,h\rangle}$ is a
maximal slide in $\langle C, h\rangle$ *over which* p *is true*

[3] Opt allows you to use the traditional syntax for axioms, which is a lot less cumbersome that PDDL's.

if and only if

$[d_1, d_2]$ is a slide,
and for all d, $d_1 < d < d_2$
$I(p, E)(C(d))$ = *true*,
and there is a $d_0 < d_1$ such that
for all d, $d_0 < d < d_1$
$I(p, E)(C(d))$ = *false*,
and there is a $d_3 > d_2$ such that
for all d, $d_2 < d < d_3$
$I(p, E)(C(d))$ = *false*

To preserve flexibility, the definition doesn't say what the truth value of p is at d_1 or d_2. So the interval over which p is true can be open or closed.

Finally, figure 1 shows truth condition for processes. This condition may appear more complex than one would expect, because a conceptual "if and only if":

> A process occurs over an interval if and only if that interval is a maximal slide over which its `:condition` is true.

has had to be broken into an "if" clause and an "only if" clause, to allow the process's boundaries to differ infinitesimally from the boundaries of the maximal slide. The reason for this slop is to enforce another interleaved-concurrency constraint:

- For all pairs of process terms $p_1 \neq p_2$, all timelines $\langle C, h\rangle \in T$ and all environments E_1 and E_2, if $I(p_1, E) = [d_{b1}, d_{e1}]_{\langle C,h\rangle}$ and $I(p_2, E) = [d_{b2}, d_{e2}]_{\langle C,h\rangle}$ then $d_{b1} \neq d_{b2}$ and $d_{e1} \neq d_{e2}$. In other words, no two processes begin or end at precisely the same moment.

The interpretation of the "wait" actions can now be specified:

- $I_0($`wait-while`$) =$

  $(\lambda\,(p)\ \{[d_1, d_2]_{\langle C,h\rangle}$
  $\mid$ *either there is a slide* $[p_b, p_e] \in p$
  such that $p_b \leq d_1 < p_e$ *and* $d_2 = p_e$
  or there is no such slide and $d_1 = d_2\}$*)*

- $I_0($`wait-for`$) =$

  $(\lambda\,(q, p)\ \{[d_1, d_2]_{\langle C,h\rangle}$
  $\mid$ *there is a slide* $[p_b, p_e] \in p$
  such that $p_b \leq d_1 \leq d_2 < p_e$
  and d_2 *is the first date in* $[d_1, p_e]_{\langle C,h\rangle}$
  such that $q(C(d)) =$ *true*$\}$*)*

Although there are many details to be fleshed out, we can summarize with this definition:

> A *model* of a PDDL domain D is an interpretation $\langle U_0, T, I_0\rangle$ of the symbols in D such that for every variable-binding environment E
>
> ⋆ For every axiom A and every date $\langle r, i\rangle$ in $\langle C, h\rangle \in T$, $I(A, E)(s) =$ *true*.
>
> ⋆ and For every process or action definition P, $I(P, E)$ = true.
>
> where I is the extension of I_0 outlined above, respecting the interleaving constraints and the required definitions of symbols such as `derivative` and `seq`.

Assessment

I have been thinking about the logic of processes for a long time (McDermott 1982). The contents of this paper are basically a further refinement of those ideas.

The only other attempt I know of to add autonomous processes to PDDL is the proposal by Fox and Long (Fox & Long 2001a) for the AIPS 2002 Planning Competition. The syntax of their notation differs from mine only in detail.[4] The semantics they propose is very different. They base it on the theory of hybrid automata (Henzinger 1996), so that to every domain and initial situation there corresponds an automaton. Finding a plan is finding a path through the states of the automaton. The main virtue of their approach is also its main defect: They are determined to preserve finiteness properties exploited by many planning algorithms, especially that there are only a finite number of plan states, and that the branching factor at each plan state is finite. The states of the hybrid automaton are all possible sets of ground atomic formulas in the language. For the state set to be finite, atomic formulas with numeric arguments must be separated out and treated in a special way. The whole apparatus becomes quite unwieldy.[5]

I believe that maintaining finiteness properties required by some current planners should be rejected as a constraint on domain-description languages. If some problems cannot be solved by some planners, so be it. It would be better to make it the responsibility of any planning system to decide whether a problem is beyond its scope. Of course, requirements flags can provide a broad-brush portrait of what a planner must be able to handle in order to solve a problem, so it's important to keep the set of flags up to date as the language evolves. But it's asking too much for PDDL to fit the capabilities of some set of existing planners exactly. In the first planning competition, there were loud votes for a `:length` field to be included in every problem description, specifying a bound on the length of a solution, because several systems had to guess a length in order to get started. We reluctantly acceded to that demand, but the `:length` field has since been taken out on the grounds that it's an arbitrary hint to a special class of planners. Trying to base the semantics of PDDL processes on finite-state hybrid automata strikes me as an even worse accommodation to the needs of a "special-interest group."

There is an important issue raised by (Fox & Long 2001a), namely, how do you check the correctness of a problem solution when real numbers are involved? A complete answer is beyond the scope of this paper, but I believe the semantic framework laid out here lends itself to the idea of "approxi-

[4] Which means it will be the subject of bitter and unending debate before the next competition!

[5] I should acknowledge the regrettable fact that a specification of the formal semantics of anything is generally clear only to the person that wrote it, so my proposal is probably as opaque to Maria and Derek as theirs is to me.

- $I($(:process a (Fun Slide <- α)
:parameters r^{α}
:condition $p^{\texttt{Prop}}$
:start-effect $b^{\texttt{Prop}}$
:effect $e^{\texttt{Prop}}$
:stop-effect $w^{\texttt{Prop}}$),
E)
= *true*
if and only if
for all $\langle C, h\rangle \in T$,
and every E' that differs from E,
if at all, only in the assignments
of the variables in r,
and every interval $[d_1, d_2]$ in $\langle C, h\rangle$
where $d_1 = \langle r_1, i_1\rangle$ and $d_2 = \langle r_2, i_2\rangle$,
(a) If $\langle I(r^{\alpha}, E'), [d_1, d_2]_{\langle C,h\rangle}\rangle \in I_0(a)$
then there is a maximal slide $[\langle r_1, i_{s1}\rangle, \langle r_2, i_{s2}\rangle]_{\langle C,h\rangle}$ over which p is true
with respect to $\langle U, T, I\rangle$ and E'
such that $i_{s1} \leq i_1$ and $i_{s2} \leq i_2$
and $I(b^{\texttt{Prop}}, E')(C(r_1, i_1 + 1)) =$ *true*
and $I(w^{\texttt{Prop}}, E')(C(r_2, i_2 + 1)) =$ *true*
and
(b) If $[d_1, d_2]$ is a maximal slide over which p is true
with respect to $\langle U, T, I\rangle$ and E'
then there are dates $d_1' = \langle r_1, i_{p1}\rangle$ and $d_2' = \langle r_2, i_{p2}\rangle$,
such that $\langle I(r^{\alpha}, E'), [d_1', d_2']_{\langle C,h\rangle}\rangle \in I_0(a)$
and $i_{p1} \geq i_1$ and $i_{p2} \geq i_2$
and $\langle I(r^{\alpha}, E'), [d_1', d_2']\rangle \in I_0(a)$
and $I(b^{\texttt{Prop}}, E')(C(r_1, i_{p1} + 1)) =$ *true*
and $I(w^{\texttt{Prop}}, E')(C(r_2, i_{p2} + 1)) =$ *true*

Figure 1: Truth Condition for Process Definitions

mate simulation" of a plan. The exact dates at which events occur is not important as long as every step the planner takes is feasible when it takes it, and the plan simulator stops in a state where the goal condition is true. If a problem includes an objective function, we evaluate it in that final state. The value may be slightly different from the true mathematically attainable minimum, but all we require is that it be comparable to the results obtained by other planners. The only tricky part is how to deal with equalities in process specifications. If the `:condition` of a process includes a formula `(not (= x y))`, then the process stops when x equals y. Finding the precise instant when that occurs is usually impossible. To fix this problem, all the planner has to do is convert the "$=$" goal relationship to a "$\geq$" or "$\leq$" by observing whether $x < y$ or $x > y$ when the process starts, and do the best job it can of finding the earliest time when the new inequality becomes true. A similar trick will work for determining when the action `(wait-for (= x y) p)` ends.

There is also the easily overlooked issue of plan execution. The assumption that actions take infinitesimal time is of course absurd when applied to a physically possible action. Obviously, there must be some temporal grain size ϵ specified to the executor such that any action must take less than ϵ, and every process must take more than ϵ. Otherwise, the domain model is simply inappropriate.

Finally, it should be noted that, because action and process definitions have truth conditions, they can be "reverse engineered" to yield formulas with the same truth conditions. These would not be legal PDDL axioms, because they would have to mention multiple situations explicitly, whereas PDDL axioms refer implicitly to one and only one situation. However, to translate PDDL to Kif (**?**), which has no built-in action-definition syntax, these axioms could be very useful. There existence also makes it clear that PDDL is not "predicate calculus + actions"; it's just "predicate calculus + useful macros for writing action-definition axioms."

References

Fox, M., and Long, D. 2001a. Pddl. + level 5: An Extension to PDDL2.1 for Modeling Planning Domain with Continuous Time-dependent Effects available at `http://www.dur.ac.uk/d.p.long/pddllevel5.ps.gz`.

Fox, M., and Long, D. 2001b. Pddl. 2.1: An Extension to PDDL for Expressing Temporal Planning Domains available at `http://www.dur.ac.uk/d.p.long/pddl2.ps.gz`.

Henzinger, T. 1996. The theory of hybrid automata. In *Proc. Annual Symposium on Logic in Computer Science*, 278–292.

McDermott, D. 1982. A temporal logic for reasoning about processes and plans. *Cognitive Science* 6:101–155.

McDermott, D. 1985. Reasoning about plans. In Hobbs, J., and Moore, R. C., eds., *Formal Theories of the Commonsense World*. Ablex Publishing Corporation. 269–317.

McDermott, D. 2003. Draft opt manual. Available at `http://www.cs.yale.edu/~dvm/papers/opt-manual.ps`.

Robinson, A. 1979. *Selected papers of Abraham Robinson: Vol. II. Nonstandard analysis and philosophy*. Yale University Press.