PDDL2.2: The Language for the Classical Part of IPC-4 — extended abstract —

Stefan Edelkamp
Fachbereich Informatik
Baroper Str. 301, GB IV
44221 Dortmund, Germany
stefan.edelkamp@cs.uni-dortmund.de

Jörg Hoffmann
Institut für Informatik
Georges-Köhler-Allee, Geb. 52
79110 Freiburg, Germany
hoffmann@informatik.uni-freiburg.de

Introduction

The 3rd International Planning Competition, IPC-3, was run by Derek Long and Maria Fox. The competition focussed on planning in temporal and metric domains. For that purpose, Fox and Long developed the PDDL2.1 language (Fox & Long 2003), of which the first three *levels* were used in IPC-3. Level 1 was the usual STRIPS and ADL planning, level 2 added numeric variables, level 3 added durational constructs.

In this document, we describe the language, named *PDDL2.2*, used for formulating the domains used in the classical part of IPC-4. As the language extensions made for IPC-3 still provide major challenges to the planning community, the language extensions for IPC-4 are relatively moderate. The first three levels of PDDL2.1 are interpreted as an agreed fundament, and kept as the basis of PDDL2.2. PDDL2.2 also inherits the separation into the three levels. The language features added on top of PDDL2.1 are *derived predicates* (into levels 1,2, and 3) and *timed initial literals* (into level 3 only). Both of these constructs are practically motivated, and are put to use in some of the competition domains. Details on the constructs are in the respective sections.

The next section discusses derived predicates, including a brief description of their syntax, and the definition of their semantics. The section after that does the same for timed initial literals. Full details, including a BNF description of PDDL2.2, can be found in a technical report (Edelkamp & Hoffmann 2004).

Derived Predicates

Derived predicates have been implemented in several planning systems in the past, including e.g. UCPOP (Penberthy & Weld 1992). They are predicates that are not affected by any of the actions available to the planner. Instead, the predicate's truth values are derived by a set of rules of the form **if** $\phi(\overline{x})$ **then** $P(\overline{x})$. The semantics are, roughly, that an instance of a derived predicate (a derived predicate whose arguments are instantiated with constants; a *fact*, for short) is TRUE iff it can be derived using the available rules (more details below). Under the name "axioms", derived predicates were a part of the original PDDL language defined by McDermott (McDermott & others 1998) for the first planning competition, but they have never been put to use in a competition benchmark (we use the name "derived predicates" instead of "axioms" in order to avoid confusion with safety conditions).

Syntax

The BNF definition of derived predicates involves just two small modifications to the BNF definition of PDDL2.1:

```
<structure-def> ::=:derived−predicates
        <derived-def>
```

The domain file specifies a list of "structures". In PDDL2.1 these were either actions or durational actions. Now we also allow "derived" definitions at these points.

```
<derived-def> ::= (:derived <atomic
        formula(term)> <GD>)
```

The "derived" definitions are the "rules" mentioned above. They simply specify the predicate P to be derived (with variable vector $\overline{x}$), and the formula $\phi(\overline{x})$ from which instances of P can be concluded to be true. Syntactically, the predicate and variables are given by the `<atomic formula(term)>` expression, and the formula is given by `<GD>` (a "goal descrption", i.e. a formula).

The BNF is more generous than what we actually allow in PDDL2.2, respectively in IPC-4. We make a number of restrictions to ensure that the definitions make sense and are easy to treat algorithmically. We call a predicate P *derived* if there is a rule that has a predicate P in its head; otherwise we call P *basic*. The restrictions we make are the following.

1. The actions available to the planner do not affect the derived predicates: no derived predicate occurs on any of the effect lists of the domain actions.
2. If a rule defines that $P(\overline{x})$ can be derived from $\phi(\overline{x})$, then the variables in $\overline{x}$ are pairwise different (and, as the notation suggests, the free variables of $\phi(\overline{x})$ are exactly the variables in $\overline{x}$).
3. If a rule defines that $P(\overline{x})$ can be derived from ϕ, then the Negation Normal Form (NNF) of $\phi(\overline{x})$ does not contain any derived predicates in negated form.

The first restriction ensures that there is a separation between the predicates that the planner can affect (the basic predicates) and those (the derived predicates) whose truth values follow from the basic predicates. The second restriction ensures that the rule right hand sides match the rule left hand sides. Let us explain the third restriction. The NNF of a formula is obtained by "pushing the negations downwards", i.e. transforming $\neg\forall x : \phi$ into $\exists x : (\neg\phi)$, $\neg\exists x : \phi$ into $\forall x : (\neg\phi)$, $\neg \bigvee \phi_i$ into $\bigwedge(\neg\phi_i)$, and $\neg \bigwedge \phi_i$ into $\bigvee(\neg\phi_i)$. Iterating these transformation steps, one ends up with a formula where negations occur only in front of atomic formulas – predicates with variable vectors, in our case. The formula contains a predicate P *in negated form* iff there is an occurence of P that is negated. By requiring that the formulas in the rules (that derive predicate values) do not contain any derived predicates in negated form, we ensure that there can not be any negative interactions between applications of the rules (see the semantics below).

An example of a derived predicate is the "above" predicate in the *Blocksworld*, which is true between blocks x and y whenever x is transitively (possibly with some blocks in between) on y. Using the derived predicates syntax, this predicate can be defined as follows.

```
(:derived (above ?x ?y)
  (or (on ?x ?y)
      (exists (?z) (and (on ?x ?z)
                        (above ?z ?y)))))
```

Note that formulating the truth value of "above" in terms of the effects of the normal *Blocksworld* actions is very awkward (the unconvinced reader is invited to try). The predicate is the transitive closure of the "on" relation.

Semantics

We now describe the updates that need to be made to the PDDL2.1 semantics definitions given by Fox and Long in (Fox & Long 2003). We introduce formal notations to capture the semantics of derived predicates. We then "hook" these semantics into the PDDL2.1 language by modifying two of the definitions in (Fox & Long 2003).

Say we are given the truth values of all (instances of the) basic predicates, and want to compute the truth values of the (instances of the) derived predicates from that. We are in this situation every time we have applied an action, or parallel action set. (In the durational context, we are in this situation at the "happenings" in our current plan, that is every time a durative action starts or finishes.) Formally, what we want to have is a function $\mathcal{D}$ that maps a set of basic facts (instances of basic predicates) to the same set but enriched with derived facts (the derivable instances of the derived predicates). Assume we are given the set R of rules for the derived predicates, where the elements of R have the form $(P(\overline{x}), \phi(\overline{x}))$ – **if** $\phi(\overline{x})$ **then** $P(\overline{x})$. Then $\mathcal{D}(s)$, for a set of basic facts s, is defined as follows.

$$\mathcal{D}(s) := \bigcap\{s' \mid s \subseteq s', \forall(P(\overline{x}), \phi(\overline{x})) \in R : \forall\overline{c}, |\overline{c}| = |\overline{x}| : (s' \models \phi(\overline{c}) \Rightarrow P(\overline{c}) \in s')\}$$

This definition uses the standard notations of the modelling relation $\models$ between states (represented as sets of facts in our case) and formulas, and of the substitution $\phi(\overline{c})$ of the free variables in formula $\phi(\overline{x})$ with a constant vector $\overline{c}$. In words, $\mathcal{D}(s)$ is the intersection of all supersets of s that are closed under application of the rules R.

Remember that we restrict the rules to not contain any derived predicates in negated form. This implies that the order in which the rules are applied to a state does not matter (we can not "lose" any derived facts by deriving other facts first). This, in turn, implies that $\mathcal{D}(s)$ is itself closed under application of the rules R. In other words, $\mathcal{D}(s)$ is the least fixed point over the possible applications of the rules R to the state where all derived facts are assumed to be FALSE (represented by their not being contained in s).

More constructively, $\mathcal{D}(s)$ can be computed by the following simple process.

```
$s' := s$
do
    select a rule $(P(\overline{x}), \phi(\overline{x}))$ and a vector $\overline{c}$ of constants,
        $|\overline{c}| = |\overline{x}|$, such that $s' \models \phi(\overline{c})$
    let $s' := s' \cup \{P(\overline{c})\}$
until no rule and constant vector could be selected
let $\mathcal{D}(s) := s'$
```

In words, apply the applicable rules in an arbitrary order until no new facts can be derived anymore.

We can now specify what an executable plan is in PDDL2.1 with derived predicates. All we need to do is to hook the function $\mathcal{D}$ into Definition 13, "Happening Execution", in (Fox & Long 2003). By this definition, Fox and Long define the state transitions in a plan. The happenings in a (temporal or non-temporal) plan are all time points at which at least one action effect occurs. Fox and Long's definition is this:

Definition 13 Happening Execution (Fox and Long (2003))
Given a state, $(t, s, \mathbf{x})$ and a happening, H, the activity *for H is the set of grounded actions*

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

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*

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

and $\mathbf{x}'$ is the result of applying the composition of the functions $\{\text{NPF}_a \mid a \in A_H\}$ to $\mathbf{x}$.

Note that the happenings consist of grounded actions, i.e. all operator parameters are instantiated with constants. To introduce the semantics of derived predicates, we now modify the result of executing the happening. (We will also adapt the definition of mutex actions, see below.) The result of executing the happening is now obtained by applying the actions to s, then subtracting all derived facts from this, then applying the function $\mathcal{D}$. That is, in the above definition we replace $(***)$ with the following:

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

where D denotes the set of all derived facts. If there are no derived predicates, D is the empty set and $\mathcal{D}$ is the identity function.

As an example, say we have a *Blocksworld* instance where A is on B is on C, $s = \{clear(A), on(A,B), on(B,C), ontable(C), above(A,B), above(B,C), above(A,C)\}$, and our happening applies an action that moves A to the table. Then the happening execution result will be computed by removing $on(A,B)$ from s, adding $clear(B)$ and $ontable(A)$ into s, removing all of $above(A,B)$, $above(B,C)$, and $above(A,C)$ from s, and applying $\mathcal{D}$ to this, which will re-introduce (only) $above(B,C)$. So s' will be $s' = \{clear(A), ontable(A), clear(B), on(B,C), ontable(C), above(B,C)\}$.

By the definition of happening execution, Fox and Long (Fox & Long 2003) define the state transitions in a plan. The definitions of what an executable plan is, and when a plan achieves the goal, are then standard. The plan is *executable* if the result of all happenings in the plan is defined. This means that all action preconditions have to be fulfilled in the state of execution, and that no two pairs of actions in a happening are *mutex*. The plan *achieves the goal* if the goal holds true in the state that results after the execution of all actions in the plan.

With our above extension of the definition of happening executions, the definitions of plan executability and goal achievement need not be changed. We do, however, need to adapt the definition of when a pair of actions is mutex. This is important if the happenings can contain more than one action, i.e. if we consider parallel (e.g. Graphplan-style) or concurrent (durational) planning. Fox and Long (Fox & Long 2003) give a conservative definition that forbids the actions to interact in any possible way. The definition is the following.

Definition 12 Mutex Actions (Fox and Long (2003)) *Two grounded actions, a and b are* non-interfering *if*

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

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

Note that the definition talks about grounded actions where all operator parameters are instantiated with constants. L_a, L_b, R_a, and R_b refer to the left and right hand side of a's and b's numeric effects. Add_a/Add_b and Del_a/Del_b are a's and b's positive (add) respectively negative (delete) effects. $GPre_a/Gpre_b$ denotes all (ground) facts that occur in a's/b's precondition. If a precondition contains quantifiers then these are grounded out ($\forall x$ transforms to $\bigwedge c_i$, $\exists x$ transforms to $\bigvee c_i$ where the c_i are all objects in the given instance), and $GPre$ is defined over the resulting quantifier-free (and thus variable-free) formula. Note that this definition of mutex actions is very conservative – if, e.g., fact F occurs only positively in a's precondition, then it does not matter if F is among the add effects of b. The conservative definition has the advantage that it makes it algorithmically very easy to figure out if or if not a and b are mutex.

In the presence of derived predicates, the above definition needs to be extended to exclude possible interactions that can arise indirectly due to derived facts, in the precondition of the one action, whose truth value depends on the truth value of (basic) facts affected by the effects of the other action. In the same spirit in that Fox and Long forbid any possibility of direct interaction, we now forbid any possibility of indirect interaction. Assume we ground out all rules $(P(\overline{x}), \phi(\overline{x}))$ for the derived predicates, i.e. we insert all possible vectors $\overline{c}$ of constants; we also ground out the quantifiers in the formulas $\phi(\overline{c})$, ending up with variable free rules. We define a directed graph where the nodes are (ground) facts, and an edge from fact F to fact F' is inserted iff there is a grounded rule $(P(\overline{c}), \phi(\overline{c}))$ such that $F' = P(\overline{c})$, and F occurs in $\phi(\overline{c})$. Now say we have an action a, where all ground facts occuring in a's precondition are, see above, denoted by $GPre_a$. By $DPre_a$ we denote all ground facts that can possibly influence the truth values of the derived facts in $GPre_a$:

$$DPre_a := \{F \mid \text{there is a path from } F \text{ to an } F' \in GPre_a\}$$

The definition of mutex actions is now updated simply by replacing, in the above definition, $(***)$ with:

$$\begin{gathered} (DPre_a \cup GPre_a) \cap (Add_b \cup Del_b) = \\ (DPre_b \cup GPre_b) \cap (Add_a \cup Del_a) = \emptyset \end{gathered}$$

As an example, reconsider the *Blocksworld* and the "above" predicate. Assume that the action that moves a block A to the table requires as an additional, derived, precondition, that A is above some third block. Then, in principle, two actions that move two different blocks A and B to the table can be executed in parallel. Which block A (B) is on can influence the $above$ relations in that B (A) participates; however, this does not matter because if A and B can be both moved then this implies that they are both clear, which implies that they are on top of different stacks anyway. We observe that the latter is a statement about the domain semantics that either requires non-trivial reasoning, or access to the world state in which the actions are executed. In order to avoid the need to either do non-trivial reasoning about domain semantics, or resort to a forward search, our definition is the conservative one given above. The definition makes the actions moving A and B mutex on the grounds that they can possibly influence each other's derived preconditions.

The definition adaptions described above suffice to define the semantics of derived predicates for the whole of PDDL2.2. Fox and Long reduce the temporal case to the case of simple plans above, so by adapting the simple-plan definitions we have automatically adapted the definitions of the more complex cases. In the temporal setting, PDDL2.2 level 3, the derived predicates semantics are that their values are computed anew at each happening in the plan where an action effect occurs.

Timed Initial Literals

Timed initial literals are a syntactically very simple way of expressing a certain restricted form of exogenous events: facts that will become TRUE or FALSE at time points that are known to the planner in advance, independently of the actions that the planner chooses to execute. Timed initial literals are thus deterministic unconditional exogenous events. Syntactically, we simply allow the initial state to specify – beside the usual facts that are true at time point 0 – literals that will become true at time points greater than 0.

Timed initial literals are practically very relevant: in the real world, deterministic unconditional exogenous events are very common, typically in the form of time windows (within which a shop has opened, within which humans work, within which traffic is slow, within which there is daylight, within which a seminar room is occupied, within which nobody answers their mail because they are all at conferences, etc.).

Syntax

As said, the syntax simply allows literals with time points in the initial state.

```
        <init> ::= (:init <init-el>*)
  <init-el> ::=:timed-initial-literals (at <number>
              <literal(name)>)
```

The requirement flag for timed initial literals implies the requirement flag for durational actions, i.e. as said the language construct is only available in PDDL2.2 level 3. The times `<number>` at which the timed literals occur are restricted to be greater than 0. If there are also derived predicates in the domain, then the timed literals are restricted to not influence any of these, i.e., like action effects they are only allowed to affect the truth values of the basic (non-derived) predicates (IPC-4 will not use both derived predicates and timed initial literals within the same domain).

As an illustrative example, consider a planning task where the goal is to be done with the shopping. There is a single action *go-shopping* that achieves the goal, and requires the (single) shop to be open as the precondition. The shop opens at time 9 relative to the initial state, and closes at time 20. We can express the shop opening times by two timed initial literals:

```
(:init
  (at 9 (shop-open))
  (at 20 (not (shop-open)))
)
```

Semantics

We now describe the updates that need to be made to the PDDL2.1 semantics definitions given by Fox and Long in (Fox & Long 2003). Adapting two of the definitions suffices.

The first definition we need to adapt is the one that defines what a "simple plan", and its happening sequence, is. The original definition by Fox and Long is this.

Definition 11 Simple Plan (Fox and Long (2003))
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.*

In the STRIPS case, the time stamps are the natural numbers $1, \dots, n$ when there are n actions/parallel action sets in the plan. The happenings then are the actions/parallel action sets at the respective time steps. Fox and Long reduce the temporal planning case to the simple plan case defined here by splitting each durational action up into at least two simple actions – the start action, the end action, and possibly several actions in between that guard the durational action's invariants at the points where other action effects occur. So in the temporal case, the happening sequence is comprised of all time points at which "something happens", i.e. at which some action effect occurs.

To introduce our intended semantics of timed initial literals, all we need to do to this definition is to introduce additional happenings into the temporal plan, namely the time points at which some timed initial literal occurs. The timed initial literals can be interpreted as simple actions that are forced into the respective happenings (rather than selected into them by the planner), whose precondition is true, and whose only effect is the respective literal. The rest of Fox and Long's definitions then carry over directly (except goal achievement, which involves a little care, see below). The PDDL2.2 definition of simple plans is this here.

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. By t_{end} we denote the largest time t in SP, or 0 if SP is empty.*

Let TL be the (finite) set of all timed initial literals, given as pairs (t, l) where t is the rational-valued time of occurence of the literal l. We identify each timed initial literal (t, l) in TL with a uniquely named simple action that is associated with time t, whose precondition is TRUE, and whose only effect is l.

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 and TL. 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 or TL.*

Thus the happenings in a temporal plan are all points in time where either an action effect, or a timed literal, occurs. The timed literals are simple actions forced into the plan. With this construction, Fox and Long's Definitions 12 (Mutex Actions) and 13 (Happening Execution), as described (and adapted to derived predicates) in Section , can be kept unchanged. They state that no action effect is allowed to interfere with a timed initial literal, and that the timed initial literals are true in the state that results from the execution of the happening they are contained in. Fox and Long's Definition 14 (Executability of a plan) can also be kept unchanged – the timed initial literals change the happenings in the plan, but not the conditions under which a happening can be executed.

The only definition we need to re-think is that of what the *makespan* of a valid plan is. In Fox and Long's original definition, this is implicit in the definition of vaild plans. The definition is this.

Definition 15 Validity of a Simple Plan (Fox and Long (2003))
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.*

The makespan of the valid plan is accessible in PDDL2.1 and PDDL2.2 by the "total-time" variable that can be used in the optimization expression. Naturally, Fox and Long take the makespan to be the end of the plan, the time point of the plan's final state.

In the presence of timed initial literals, the question of what the plan's makespan is becomes a little more subtle. With Fox and Long's above original definition, the makespan would be the end of all happenings in the simple plan, which *include* all timed initial literals (see the revised Definition 11 above). So the plan would at least take as long as it takes until no more timed literals occur. But a plan might be finished long before that – imagine something that needs to be done while there is daylight; certainly the plan does not need to wait until sunset. We therefore define the makespan to be the earliest point in time at which the goal condition becomes (and remains) true. Formally this reads as follows.

Definition 15 Validity and Makespan 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. The plan's* makespan *is the smallest $t \geq t_{end}$ such that, for all happenings at times $t' \geq t$ in the plan's happening sequence, the goal specification is satisfied after execution of the happening.*

Remember that t_{end} denotes the time of the last happening in the plan that contains an effect caused by the plan's *actions* – in simpler terms, t_{end} is the end point of the plan. What the definition says is that the plan is valid if, at some time point t after the plan's end, the goal condition is achieved and remains true until after the last timed literal has occured. The plan's makespan is the first such time point t. Note that the planner can "use" the events to achieve the goal, by doing nothing until a timed literal occurs that makes the goal condition true – but then the waiting time until the nearest such timed literal is counted into the plan's makespan. (The latter is done to avoid situations where the planner could prefer to wait millions of years rather than just applying a single action itself.) Remember that the makespan of the plan, defined as above, is what can be denoted by `total-time` in the optimization expression defined with the problem instance.

Acknowledgements. We would like to thank the IPC-4 organizing committee for their help in taking the decision about the language for the classical part of IPC-4, and in ironing out the details about syntax and semantics. The people contributing to this discussion were Drew McDermott, Daniel Weld, David Smith, Hakan Younes, Jussi Rintanen, Sylvie Thiebaux, Maria Fox, and Derek Long. We especially thank Maria Fox and Derek Long for giving us the latex sources of their PDDL2.1 article, and for discussing the modifications of this document needed to introduce the semantics of derived predicates and timed initial literals.

References

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*. Special issue on the 3rd International Planning Competition, to appear.

McDermott, D., et al. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee.

Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B.; Swartout, W.; and Rich, C., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference (KR-92)*, 103–114. Cambridge, MA: Morgan Kaufmann.

Thiebaux, S.; Hoffmann, J.; and Nebel, B. 2003. In defense of PDDL axioms. In Gottlob, G., ed., *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*. Acapulco, Mexico: Morgan Kaufmann. accepted for publication.