PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition

Stefan Edelkamp
Fachbereich Informatik
Baroper Str. 301, GB IV
44221 Dortmund, Germany
stefan.edelkamp@cs.uni-dortmund.de
+49 (231) 755-5809

Jörg Hoffmann
Institut für Informatik
Georges-Köhler-Allee, Geb. 52
79110 Freiburg, Germany
hoffmann@informatik.uni-freiburg.de
+49 (761) 203-8229

Abstract

This document defines the language to be used in the classical part of the 4th International Planning Competition, IPC-4. The language comprises all of PDDL2.1 levels 1, 2, and 3, as defined by Maria Fox and Derek Long in [1]; parts of this document have been copied from that source. On top of this language, for IPC-4 *derived predicates* are re-introduced, and *timed initial literals* are (newly) introduced into the competition language. We give the syntax and semantics of these constructs.

Technical Report No. 195

Report Date January 21, 2004

1 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 [1], 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.

The 4th International Planning Competition, IPC-4, will take place alongside ICAPS-2004. IPC-4 will feature separate parts for *classical* and *probabilistic* planning. **In the document at hand, we are only concerned with the classical part**. In this part, it will be tried to provide a useful range of benchmark domains, motivated by applications and interesting in structure.

In this document, we focus on the language that will be used for formulating the IPC-4 domains, named *PDDL2.2*. 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 will be put to use in some of the competition domains. Details on the constructs are in the respective sections. Here, let us note that of all domains in IPC-4 that make use of these constructs there will also be formulations of these domains were the constructs have been compiled away (i.e., translated into artificial facts and operators). **So every planner that can handle the first three levels of PDDL2.1, or a subset thereof, will have no difficulty in participating in IPC-4.** There will be less stress on language coverage than there was at IPC-3 (see also Section 4).

Section 2 discusses derived predicates, including a brief description of their syntax, and the definition of their semantics. Section 3 does the same for timed initial literals. Section 4 gives a brief overview over the overall arrangements we envision for the (classical part of) IPC-4. Appendix A provides the full BNF syntax of the IPC-4 language, PDDL2.2. Please note that this document is only a draft that has been put together for the use of potential IPC-4 participants. We did not have much time to polish the presentation. The semantic definitions we give are done by adapting parts of the definitions given for PDDL2.1 by Fox and Long in [1]. For full background information, we refer the reader to this latter article. If you have any questions/suggestions/comments, please contact the authors by e-mail or phone.

2 Derived Predicates

Derived predicates have been implemented in several planning systems in the past, including e.g. UCPOP [3]. 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 [2] 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). Derived predicates combine several key aspects that make them a useful language extension for IPC-4:

- They are practically motivated: in particular, they provide a concise and convenient means to express updates on the transitive closure of a relation. Such updates occur in domains that include structures such as paths or flows (electricity flows, chemical flows, etc.); in particular some interesting domains that will be used in IPC-4 include this kind of structure.

- They are also theoretically justified in that compiling them away can be infeasible: Bernhard Nebel proved that, in the worst case, compiling derived predicates away results in an exponential blow up of either the problem description, or the plan length [4].

- Last but not least, derived predicates do not cause a significant implementation overhead in, at least, forward search planners: when the world state – the truth values of all non-derived, *basic*, predicates – is known, computing the truth values of the derived predicates is conceptually trivial.

IPC-4 will use derived predicates only in the non-durational context, PDDL2.2 level 1, maybe also level 2. In this context, derived predicates can be compiled away, by introducing artificial actions and facts. As said before, in IPC-4 we intend to provide the competitors that can not handle derived predicates with compiled formulations of the respective domains (see also Section 4).

Below we define the syntax and semantics of derived predicates as will be used in IPC-4. (For completeness, the definition also covers the durational case, PDDL2.2 level 3, even though as said we do not intend to use this combination in IPC-4.)

2.1 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 full BNF definitions can be looked up in Appendix A. 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. Basically, this – encoding of transitive closures – is what we intend to use derived predicates for in IPC-4.

2.2 Semantics

We now describe the updates that need to be made to the PDDL2.1 semantics definitions given by Fox and Long in [1]. 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 [1].

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 appropriate 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 [1]. 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 [1])
Given a state, $(t, s, \mathbf{x})$ *and a happening,* H*, the* activity *for* H *is the set of grounded actions*

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

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*

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

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 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 [1] 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 [1] 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 [1])
Two grounded actions, a and b are non-interfering *if*

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

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 $GPre_a$:

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

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

$$(DPre_a \cup GPre_a) \cap (Add_b \cup Del_b) = (DPre_b \cup GPre_b) \cap (Add_a \cup Del_a) = \emptyset$$

Note that what we define are, as before, mutexes between actions, *not* (as some people speculated in individual discussions we had) mutexes between actions and rules (that derive predicates). 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.

Let us remark explicitly that 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. As said before, the IPC-4 domains will, however, not use derived predicates in the durational context. (Partly because we don't have a domain where we need both durations and derived predicates, and partly because compiling derived predicates away is not easily possibly in the presence of durations, where the actions that the compilation introduces take time, and time can be critical.)

3 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.).

IPC-4 will use timed initial literals only in the durational context, PDDL2.2

level 3. As said before, we intend to provide the competitors that can not handle timed initial literals with compiled formulations of the respective domains (see also Section 4). Below we define the syntax and semantics of derived predicates as will be used in IPC-4.

3.1 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 (see also Section A.6), 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)))
)
```

Note that what we express this way is basically a time window. This – expressing time windows – is what timed initial literals will be used for in IPC-4.

3.2 Semantics

We now describe the updates that need to be made to the PDDL2.1 semantics definitions given by Fox and Long in [1]. 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 [1])
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 $1, \ldots, n$ 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 2.2, 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 [1])
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.) Please 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**.

In conclusion of our description of the new language features in PDDL2.2, let us reiterate that the full BNF syntax for PDDL2.2 is contained in Appendix A, and that the full definition of the PDDL2.2 semantics results from the definitions given by Fox and Long for PDDL2.1 in [1], when extending Fox and Long's definitions as described above in this document. We now close the document with an outline of our plans for the classical part of IPC-4.

4 IPC-4 (Classical Part) Overview

The main effort made in the preparation of the classical part of IPC-4 is an attempt to come up with an interesting range of benchmark domains: domains that are motivated by applications, and that are interesting in structure. We expect to use between five and eight domains in the competition. Of each domain there will be different **versions**, differing in terms of the (number of) problem constraints they consider. Of each domain version, there will be different **formulations**, differing in terms of the language used to formulate the (same) problem constraints. (Formulations of complex problem constraints in more primitive languages such as STRIPS will be made by compiling down from the instances formulated in richer languages such as ADL.) We intend to let the competitors choose whichever formulation of a domain version they can handle best/handle at all. The results for one domain *version* will then be evaluated together (irrespective of the *formulation* used by the individual planners). We expect to use two domains whose most natural (and concise) formulation features derived predicates, and two domains where the most natural formulation (of the most complex domain version) uses timed initial literals. As said before, **of all domain versions that use one of the new language features, we will also provide formulations where these new features have**

been compiled away. As an example, of a domain we might have the three different versions *non-temporal*, *temporal*, *temporal with time windows*. In each of the versions we might have a formulation that uses ADL, and a formulation where ADL has been compiled into STRIPS. In the temporal version with time windows, one formulation would use timed initial literals to encode the time windows, while another formulation would encode the time windows by artificial facts and actions.

A bulleted list of the main points we will be trying to make is this here:

- A structurally interesting range of benchmark domains that, in their most complex versions, come close (that is, as close as feasible in the context of the competition) to existing applications of planning.

- Make the parsing/language (implementation) barrier less prohibitive by providing compiled formulations of the domains.

- Less stress on coverage than at IPC-3: when using compiled domain formulations (rather than simplified domain formulations, as has traditionally been done) in more primitve languages, even the STRIPS domains will constitute an interesting range of benchmark problems, and good performance across them will be honoured.

- Reviving optimal planning: we will make sure that in all domain versions there are enough smaller instances to obtain meaningful data to compare optimal planners – planners that prove a guarantee on the quality of the found solution (such planners typically do not scale as far as the sub-optimal approaches). We intend to give a separate price to the best (most efficient) optimal planner in IPC-4.

For more, and up-to-date, details on the main aspects of (the classical part of) IPC-4, in particular for downloads of testing examples showing the language features and compilations we intend to use, please consider the IPC-4 web page at `http://ipc.icaps-conference.org/`

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 thank Maria Fox and Derek Long for giving us the latex sources for the BNF description of PDDL2.1 (they, in turn, started from the original sources provided by McDermott). We also thank Fox and

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

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

[2] Drew McDermott et al. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee, 1998.

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

[4] Sylvie Thiebaux, Jörg Hoffmann, and Bernhard Nebel. In defense of PDDL axioms. In G. Gottlob, editor, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, Acapulco, Mexico, August 2003. Morgan Kaufmann. accepted for publication.

A BNF Description of PDDL2.2

This appendix contains a complete BNF specification of the PDDL2.2 language. This is the same as the first three levels of PDDL2.1 (the language used in the IPC-3 domains), plus the possibilities to define derived predicates, and timed literals. Level 4 of PDDL2.1 (continuous effects) has been skipped as the IPC-4 committee considered it a too significant competition language extension at the current point in time. Note that this does not exclude the possibility to introduce PDDL2.1 level 4 into the languages of future competitions.

For readability, in the following we mark with $(***)$ the points in the BNF where the new language constructs of PDDL2.2 are inserted.

A.1 Domains

Domains are defined exactly as in PDDL2.2, except that we now also allow to define rules for derived predicates at the points where operators (actions) are allowed.

```
<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.6
<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 function skeleton)>)
<structure-def>      ::= <action-def>
<structure-def>      ::=^:durative-actions <durative-action-def>
(***) <structure-def> ::=^:derived-predicates <derived-def>
<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.2 Actions

The BNF for an action definition is the same as in PDDL2.2.

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

```
<GD>                  ::= ()
<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-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).
<effect>       ::= ()
<effect>       ::= (and <c-effect>*)
<effect>       ::= <c-effect>
<c-effect>     ::=:conditional-effects (forall (<variable>*) <effect>)
<c-effect>     ::=:conditional-effects (when <GD> <cond-effect>)
<c-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>)
```

```
<cond-effect> ::= (and <p-effect>*)
<cond-effect> ::= <p-effect>
<assign-op>   ::= assign
<assign-op>   ::= scale-up
<assign-op>   ::= scale-down
<assign-op>   ::= increase
<assign-op>   ::= decrease
```

A.3 Durative Actions

Durative actions are the same as in PDDL2.2, except that we restrict ourselves to level 3 actions, where the duration is given as the fixed value of a numeric expression (rather than as the possible values defined by a set of constraints). This slightly simplifies the BNF.

```
<durative-action-def> ::= (:durative-action <da-symbol>
                               :parameters ( <typed list (variable)> )
                               <da-def body>)
<da-symbol>           ::= <name>
<da-def body>         ::= :duration (= ?duration <f-exp>)
                          :condition <da-GD>
                          :effect <da-effect>
<da-GD>                  ::= ()
<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
```

A.4 Derived predicates

As said, rules for derived predicates can be given in the domain description at the points where actions are allowed. The BNF is:

```
(***) <derived-def>      ::= (:derived <typed list (variable)> <GD>)
```

Note that we allow the specification of types with the derived predicate arguments. This might seem redundant as the predicate types are already declared in the `:predicates` field. Allowing to specify types with the predicate (rule) "parameters" serves to give the language a more unified look-and-feel, and one might use the option to make the parameter ranges more restrictive. (Remember that the specification of types is optional, not manadatory.)

Repeating what has been said in Section 2.1, this BNF is more generous than what is considered a well-formed domain description in PDDL2.2. 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 apply are:

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.

A.5 Problems

The only change made to PDDL2.1 in the problem description is that we allow the specification of timed initial literals.

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

```
<ground-f-exp>       ::= <number>
<ground-f-exp>       ::= (<function-symbol> <name>*)
<ground-f-exp>       ::= total-time
<ground-f-exp>       ::= <function-symbol>
```

Repeating what has been said in Section 2.1, the requirement flag for timed initial literals implies the requirement flag for durational actions (see also Section A.6), i.e. the language construct is only available in PDDL2.2 level 3. Also, the above BNF is more generous than what is considered a well-formed problem description in PDDL2.2. 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).

A.6 Requirements

Here is a table of all requirements in PDDL2.2. 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`
`:durative-actions`	Allows durative actions. Note that this does not imply `:fluents`.
`:derived-predicates`	Allows predicates whose truth value is defined by a formula
`:timed-initial-literals`	Allows the initial state to specify literals that will become true at a specified time point **implies** `durative actions` **(i.e. applicable only in PDDL2.2 level 3)**