Subset of PDDL for the AIPS2000 Planning Competition
Draft 1

This document defines the subset of PDDL (planning domain definition language) that will be used in the AIPS-2000 planning Competition. The full PDDL is defined in [1], and entire sections of this document have been copied verbatim from that source.

Contact Fahiem Bacchus (fbacchus@cs.toronto.edu) with comments.

Modification History

Jan. 27th.	Original Version.
Feb. 14th.	(1) removed negative preconditions from STRIPS actions, (2) simplified parsing of ADL actions eliminating some unnecessary "nesting", (3) specified that problems will have all `:objects` explicitly listed.

1 Introduction

This document defines the subset of the PDDL language that will be utilized in the AIPS-2000 planning competition. The restrictions imposed here are particularly important for the first track of the competition (that involving fully automatic planning systems).

2 Syntactic Notation

We follow the notation used in the original PDDL definition. That is, we use an extended BNF (EBNF) with the following conventions:

1. Each rule is of the form <*syntactic element*> `::=` *expansion*.
2. Angle brackets delimit names of syntactic elements.
3. Square brackets (`[` and `]`) surround optional material.
4. An asterisk (`*`) means "zero or more of"; a plus (`+`) means "one or more of."
5. Some syntactic elements are parameterized. E.g., `<list (symbol)>` might denote a list of symbols, where there is an EBNF definition for `<list` x`>` and a definition for `<symbol>`. `<list` x`>` is defined to be:

   `<list` x`> ::= (`x`*)`

   so that a list of symbols is just `(<symbol>*)`.
6. Ordinary parenthesis are an essential part of the syntax we are defining and have no semantics in the EBNF meta language.
7. Optional material and expansion rules can both be superscripted with a requirement flag, such as:

   - `[(:types ...)]`$^{\texttt{:typing}}$, or
   - `<atomic formula skeleton> ::=`$^{\texttt{:typing}}$ `(<predicate> <typed list (variable)>)`

   it means that the optional material can only be included and the expansion rule can only be applied when the domain has declared a requirement for that flag.

3 Domains

We now describe the language more formally. The EBNF for defining a domain is given in Fig. 1.

All domains specified in the competition will also satisfy the following:

1. All keyword arguments (for `(define (domain ...))` and all similar constructs) will appear in the order specified in Fig. 1. (Arguments may be omitted.)
2. Just one PDDL definition (of a domain, problem, etc.) will appear per file. Furthermore the complete definitions will appear in the file (i.e., there is no facility for splitting the definition over multiple files).

Figure 1 Syntax of Domain Definition

```
<domain>             ::= (define (domain <name>)
                             [<require-def>]
                             [<types-def>]:typing
                             [<constants-def>]
                             [<predicates-def>]
                             <action-def>*)
<require-def>        ::= (:requirements <require-key>+)
<require-key>        ::= :strips
<require-key>        ::= :adl
<require-key>        ::= :typing
<types-def>          ::= (:types <typed list (name)>)
<typed?-list-of (t)> ::=:typing <typed list (t)>
<typed?-list-of (t)> ::= <list (t)>
<constants-def>      ::= (:constants <typed?-list-of (names)>)
<predicates-def>     ::= (:predicates <atomic formula skeleton>+)
<atomic formula skeleton>
                     ::= (<predicate> <typed?-list-of (variable)>)
<predicate>          ::= <name>
<variable>           ::= ?<name>
```

Names: the category `<name>` consists of strings of characters that beginning with a letter and contain only letters, digits, hyphens ("-"),and underscores ("_"). *Case is not significant*. `<name>`'s are required to be unique. That is, one cannot use the same name in two different definitions.

Requirements: we restrict ourselves to only two possible requirement flags

`:strips` STRIPS-style actions.

`:adl` ADL-style actions.

Note that, specifying `:adl` makes `:strips` redundant, as ADL-style actions are a superset of STRIPS-style actions. Furthermore, `:strips` is the default if no requirement flags appear.

Types: The `:types` argument uses the original PDDL syntax:

```
<typed list (x)> ::= x+
<typed list (x)> ::= x+- <type> <typed list(x)>
<type>           ::= <name>
<type>           ::= (either <type>+)
```

A typed list is used to declare the types of a list of entities; the types are preceded by a minus sign ("`-`"), and every other element of the list is declared to be of the first type that follows it, or `object` if there are no types that follow it.

`object` and `number` are both predefined types.

An example of a `<typed list(name)>` is

```
integer float - number physob
```

If this occurs as a `:types` argument to a domain, it declares three new types, `integer`, `float`, and `physob`. The first two are subclasses of `number`, the last a subclass of `object` (by default). That is, every integer is a number, every float is a number, and every physical object is an object.

An atomic type name is just a timeless unary predicate, and may be used wherever such a predicate makes sense. In addition to atomic type names, there are also union types: `(either` t_1 `...`t_k`)` is the union of types t_1 to t_k.

Constants: The `:constants` field is simply a list of names (these names can be typed if the `:typing` requirement flag has been specified). The names in the list are taken as new constants in this domain (perhaps with specified types). E.g., the declaration

```
(:constants sahara - theater
            division1 division2 - division)
```

indicates that in this domain there are three distinguished constants, `sahara` of type `theater` and two symbols of type `divisions`. If types are not required the following declaration can be made:

```
(:constants sahara division1 division2)
```

Predicates The `:predicates` field consists of a list of declarations of predicates. For each predicate we specify a list of variables (perhaps typed) to declare the arity of the predicate (and perhaps also the types of its arguments.)
Equality "`=`" is a predefined predicate taking two arguments of any type.

4 Actions

4.1 STRIPS actions

If the domain definition specifies STRIPS-style actions (i.e., the requirement flag `:adl` does not appear) then we can only expand `<action-def>` in the manner specified in Fig. 2.

The `:parameters` list is simply the list of variables on which the particular rule operates, *i.e.*, its arguments.

`:precondition` is a conjunction of atomic formulas, while `:effect` is a conjunctions of literals. In both cases, *all of the free variables in these components must appear among the* `:parameters`.

These actions have standard STRIPS semantics. In particular, every binding of the `:parameters`, σ, generates a particular instance of the action. This instance is applicable to a state $\mathcal{S}$ if and only if the :`precondition` is true in $\mathcal{S}$ under the bindings σ. The instance will then map $\mathcal{S}$ to a new state $\mathcal{S}'$ generated by adding to $\mathcal{S}$ all positive atomic formulas appearing in `:effect` (after applying the binding σ), deleting from $\mathcal{S}$ all negative atomic formulas appearing in `:effect` (after applying σ), and leaving unchanged all other ground atomic formulas true in $\mathcal{S}$.

Figure 2 Syntax of Strips Actions

```
<action-def>        ::= (:action <name>
                           :parameters (<typed?-list-of (variable)>)
                           <action-def body>)
<action-def body>   ::= [:precondition <POS-CONJUNCTION>]
                        :effect <CONJUNCTION>
<term>              ::= <name>
<term>              ::= <variable>
<atomic formula(t)> ::= (<predicate> t*)
<literal(t)>        ::= <atomic formula(t)>
<literal(t)>        ::= (not <atomic formula(t)>)
<POS-CONJUNCTION>   ::= <atomic formula(term)>
<POS-CONJUNCTION>   ::= (and <atomic formula(term)>
                           <atomic formula(term)>+)
<CONJUNCTION>       ::= <literal(term)>
<CONJUNCTION>       ::= (and <literal(term)> <literal(term)>+)
```

4.2 ADL actions

If the domain definition specifies ADL-style actions, then we can expand `<action-def>` in the more general manner specified in Fig. 3.

All of the free variables in `:precondition <FORMULA>` and in `:effect <EFF-FORMULAS>` must appear in `:parameters`.

The effects of an ADL action are specified a more complex manner so as to avoid certain constructions that would be more difficult to give semantics to. In particular, once we use a `when` (which is not the same as `implies`), we have an `<EFF-FORMULA>` which can no longer appear as the antecedent of another `when`. That is, we cannot nest `when`s.

ADL actions have a semantics similar to their STRIPS counter parts. Once again every binding of `:parameters`, σ, generates a particular instance of the action that is applicable to a state $\mathcal{S}$ if and only if $\mathcal{S}$ satisfies the operator's `:precondition` (in this case a general first-order formula) under σ. If $\mathcal{S}$ does satisfy the precondition under the bindings σ, the action instance will map $\mathcal{S}$ to a new state $\mathcal{S}'$. $\mathcal{S}'$ can be computed from $\mathcal{S}$ by applying the operator's `:effect` (leaving all other atomic formulas true in $\mathcal{S}$ unchanged):

Figure 3 Syntax of ADL Actions

```
<action-def>       ::= (:action <name>
                          :parameters (<typed?-list-of (variable)>)
                          <action-def body>)
<action-def body> ::= [:precondition <FORMULA>]
                       :effect <EFF-FORMULA>

<FORMULA>          ::= <literal(term)>
<FORMULA>          ::= (not <FORMULA>)
<FORMULA>          ::= (and <FORMULA> <FORMULA>+)
<FORMULA>          ::= (or <FORMULA> <FORMULA>+)
<FORMULA>          ::= (imply <FORMULA> <FORMULA>)
<FORMULA>          ::= (exists (<typed?-list-of (variable)>+)
                               <FORMULA>)
<FORMULA>          ::= (forall (<typed?-list-of (variable)>+)
                               <FORMULA>)


<ATOMIC-EFFS>      ::= <literal(term)>
<ATOMIC-EFFS>      ::= (and <literal(term)> <literal(term)>+)


<ONE-EFF-FORMULA> ::= <ATOMIC-EFFS>
<ONE-EFF-FORMULA> ::= (when <FORMULA> <ATOMIC-EFFS>)
<ONE-EFF-FORMULA> ::= (forall (<typed?-list-of (variable)>+)
                               <ATOMIC-EFFS>))
<ONE-EFF-FORMULA> ::= (forall (<typed?-list-of (variable)>+)
                               (when <FORMULA> <ATOMIC-EFFS>))
<EFF-FORMULA>      ::= <ONE-EFF-FORMULA>
<EFF-FORMULA>      ::= (and <ONE-EFF-FORMULA> <ONE-EFF-FORMULA>+)
```

1. Effects of the form `<ATOMIC-EFFS>` are applied by first applying the bindings σ to `<ATOMIC-EFFS>`, and then adding to $\mathcal{S}$ all positive atomic formulas in `<ATOMIC-EFFS>` and deleting from $\mathcal{S}$ all negative atomic formulas in `<ATOMIC-EFFS>`, just like the conjunctive effects of STRIPS operators.

2. Effects of the form `(when <FORMULA> <ATOMIC-EFFS>)` are applied by first determining if $\mathcal{S} \models$ `<FORMULA>`. If this is the case then `<ATOMIC-EFFS>` is applied to $\mathcal{S}$ as in the previous case.

3. Effects of the form `(forall (<typed?-list-of-variables>)`$^+$`<ATOMIC-EFFS>)` and `(forall (<typed?-list-of-variables>)`$^+$`(when <FORMULA> <ATOMIC-

Figure 4 Syntax of Problem Definitions

```
<problem>            ::= (define (problem <name>)
                            (:domain <name>)
                            [(:requirements :typing)]
                            <object declaration>
                            [<init>]
                            <goal>+
<object declaration> ::= (:objects <typed?-list-of (name)>)
<init>               ::= (:init <atomic formula(name)>+)
<goal>               ::= (:goal <FORMULA>)
```

EFFS>)) are applied by first finding all possible bindings for <typed?-list-of-variables>[1] augmenting σ with each such binding in turn, and then applying the internal <ONE-EFF-FORMULA> with the augmented set of bindings.

4. Effects of the form (and <ONE-EFF-FORMULA> <ONE-EFF-FORMULA>$^+$) are applied to $\mathcal{S}$ by applying each of the <ONE-EFF-FORMULA> to $\mathcal{S}$.

5 Problems

A problem is what a planner tries to solve. It is defined with respect to a domain. A problem specifies two things: an initial situation, and a goal to be achieved. The syntax for problem specifications is given in Fig. 4.

A problem definition must specify an initial situation by a list of initially true ground atomic formulas[2]

The :objects field is required to be present, lists objects that exist in this problem (which might be a superset of those appearing it the initial situation). If typing is defined, this will be a typed list of objects.

The :goal of a problem definition is a formula. A solution to a problem is a series of actions such that (a) the action sequence is feasible starting in the given initial situation situation; (b) the :goal, if any, is true in the situation resulting from executing the action sequence. It is likely that :goal will be restricted to conjunctions of atomic formulas during the competition.

6 Format of Solutions

To be announced (probably in a form suitable for checking by the PDDL solution checker developed by Drew Mcdermott).

[1]Typed variables have to be bound to constants of compatible type. Untyped variables can be bound to any constant.
[2]Negative atomic formulas are assumed to be true by the closed world assumption.

References

[1] Drew McDermott et al. *PDDL—The Planning Domain Definition Language*. Yale University, 1998.