BNF definition of HPDL

Ilche Georgievski

University of Groningen
e-mail: i.georgievski@rug.nl

March 31, 2014

We provide the syntax for HPDL using an extended BNF with the following conventions:

- Each production rule is of the form <*non-terminal*>::= *expansion*.
- A syntactic element surrounded by square brackets ("[" and "]") is optional.
- Expansions and optional syntactic elements that have a superscribed requirement flag are available only if the requirement flag is declared in the domain or problem being defined. For example, `[<types-def>]`$^{\texttt{:typing}}$ in the syntax of the domain definition means that `<types-def>` may only occur in domain definitions that include the `:typing` flag in the requirements declaration.
- An asterisk ("*") following a syntactic element x means 'zero or more' occurrences of x, while a plus ("+") following x means 'at least one' occurrence of x.
- Parametrised non-terminals, for example `<typed list (`x`)>`, represent separate production rules for each instantiation of the parameter x.
- Terminals represent symbols and words written in plain text format.
- The syntax is Lisp-like meaning that case is not important (e.g., `?x` and `?X` are equivalent), parenthesis are an essential part of the syntax and have no semantic meaning in the extended BNF definition, and any number of whitespace characters (e.g., space, newline, tab) may occur between tokens.

1 Domain

```
<domain>                  ::= (define (domain <name>)
                                  [<require-def>]
                                  [<types-def>]:typing
                                  [<predicates-def>]
                                  [<functions-def>]:fluents
                                  <structure-def>*)
<require-def>             ::= (:requirements <require-key>+)
<require-key>             ::= See Section 3

<types-def>               ::= (:types <typed list (<name>)>)
<typed list (x)>          ::= x*
<typed list (x)>          ::=:typing x+ - <type> <typed list (x)>
<type>                    ::= <name>
<type>                    ::= object

<predicates-def>          ::= (:predicates <atomic formula skeleton>+)
<atomic formula skeleton> ::= (<predicate> <typed list (<variable>)>)
<predicate>               ::= <name>
<variable>                ::= ?<name>
```

```
<functions-def>           ::=:fluents (:functions <function typed list (<atomic function skeleton>)>)

<function typed list (x)> ::= x+ - <function type> <function typed list (x)>
<function typed list (x)> ::=
<function typed list (x)> ::=:numeric−fluents x+
<function type>           ::=:numeric−fluents number
<function type>           ::=:typing+:object−fluents <type>
<atomic function skeleton>::= (<function-symbol> <typed list (<variable>))
<function-symbol>         ::= <name>

<structure-def>           ::= <action-def>
<structure-def>           ::= <task-def>
<structure-def>           ::=:derived−predicates <derived-def>

<emptyOr (x)>             ::= ()
<emptyOr (x)>             ::= x

<action-def>              ::= (:action <name>
                                       :parameters (<typed list (<variable>)>)
                                       <action-def body>)
<action-def body>         ::= :precondition <emptyOr (<pre>)>
                              :effect <emptyOr (<effect>)>

<pre>                     ::= <atomic formula (<term>)>
<pre>                     ::=:negative−preconditions <literal (<term>)>
<pre>                     ::= (and <pre>*)
<pre>                     ::=:disjunctive−preconditions (or <pre>*)
<pre>                     ::=:disjunctive−preconditions (not <pre>)
<pre>                     ::=:disjunctive−preconditions (imply <pre> <pre>)
<pre>                     ::=:universal−preconditions (forall (<typed list (<variable>)>) <pre>)
<pre>                     ::=:numeric−fluents <f-comp>

<atomic formula (t)>      ::= (<predicate> t*)
<atomic formula (t)>      ::=:equality (= t t)
<literal (t)>             ::= <atomic formula (t)>
<literal (t)>             ::=(not <atomic formula (t)>)
<term>                    ::= <name>
<term>                    ::= <variable>
<term>                    ::= <number>
<term>                    ::=:object−fluents <function-term>
<function-term>           ::= (<function-symbol> <term>*)

<f-comp>                  ::= (<binary-comp> <f-exp> <f-exp>)
<binary-comp>             ::= >
<binary-comp>             ::= <
<binary-comp>             ::= =
<binary-comp>             ::= <=
<binary-comp>             ::= >=
<f-exp>                   ::=:numeric−fluents <number>
<f-exp>                   ::=:numeric−fluents (<binary-op> <f-exp> <f-exp>)
<f-exp>                   ::=:numeric−fluents (<multi-op> <f-exp> <f-exp>+)
<f-exp>                   ::=:numeric−fluents (- <f-exp>)
<f-exp>                   ::=:numeric−fluents <f-head>
<binary-op>               ::= <multi-op>
<binary-op>               ::= -
<binary-op>               ::= /
<multi-op>                ::= *
```

```
<multi-op>                ::= +
<f-head>                  ::= (<function-symbol> <term>*)
<f-head>                  ::= <function-symbol>
<function-symbol>         ::= <name>

<effect>                  ::= (and <c-effect>*)
<effect>                  ::= <c-effect>
<c-effect>                ::=:conditional−effects (forall (<typed list (<variable>)>) <effect>)
<c-effect>                ::= <p-effect>
<p-effect>                ::= (not <atomic formula (<term>)>)
<p-effect>                ::= <atomic formula (<term>)>
<p-effect>                ::=:numeric−fluents (<assign-op> <f-head> <f-exp>)
<p-effect>                ::=:object−fluents (assign <function-term> <term>)
<p-effect>                ::=:object−fluents (assign <function-term> undefined)
<assign-op>               ::= assign
<assign-op>               ::= scale-up
<assign-op>               ::= scale-down
<assign-op>               ::= increase
<assign-op>               ::= decrease

<task-def>                ::= (:task <name>
                                  :parameters (<typed list (<variable>)>)
                                  <task-def body>)
<task-def body>           ::= <method>+
<method>                  ::= (:method <name>
                                    :precondition <emptyOr (<pre>)
                                    :tasks <emptyOr(<task-list>))
<task-list>               ::= (<ordering> <tl>*)
<ordering>                ::= sequence
<ordering>                ::= unordered
<ordering>                ::= parallel
<tl>                      ::= <task-atom>
<tl>                      ::= <task-list>
<task-atom>               ::= (<name> <term>*)

<derived-def>             ::= (:derived <atomic formula skeleton> <pre>)

<name>                    ::= <letter> <any char>*
<letter>                  ::= a..z | A..Z
<any char>                ::= <letter>
<any char>                ::= <digit>
<any char>                ::= -
<any char>                ::= _
<number>                  ::= <digit>+ [<decimal>]
<digit>                   ::= 0..9
<decimal>                 ::= .<digit>+
```

2 Problem

```
<problem>                 ::= (define (problem <name>)
                                  (:domain <name>)
                                  [<require-def>]
                                  [<object-declaration>]
                                  <init>
                                  <goal-tasks>)
<require-def>             ::= (:requirements <require-key>+)
<require-key>             ::= See Section 3
```

```
<object-declaration>      ::= (:objects <typed list (<name>)>)

<init>                    ::= (:init <init-el>*)
<init-el>                 ::= <literal (<name>)>
<init-el>                 ::=:numeric−fluents (= <basic-function-term> <number>)
<init-el>                 ::=:object−fluents (= <basic-function-term> <name>)
<literal (t)>             ::= <atomic formula (t)>
<atomic formula (t)>      ::= (<predicate> t*)
<predicate> = <name>
<basic-function-term>     ::= <function-symbol>
<basic-function-term>     ::= <function-symbol> <name>
<function-symbol>         ::= <name>

<goal-tasks>              ::= (:goal-tasks <task-list>)
<task-list>               ::= (<ordering> <tl>*)
<ordering>                ::= sequence
<ordering>                ::= unordered
<ordering>                ::= parallel
<tl>                      ::= <task-atom>
<tl>                      ::= <task-list>
<task-atom>               ::= (<name> <term>*)

<name>                    ::= <letter> <any char>*
<letter>                  ::= a..z | A..Z
<any char>                ::= <letter>
<any char>                ::= <digit>
<any char>                ::= -
<any char>                ::= _
<number>                  ::= <digit>+ [<decimal>]
<digit>                   ::= 0..9
```

3 Requirements

Few planners will handle the entire PDDL language. The language is factored into subset of features, called requirements. Every domain defined using PDDL should declare which requirements it assumes. A planner that does not handle a given requirement can then skip over all definitions connected with a domain that declares that requirement, and won't even have to cope with its syntax. A domain's set of requirements allow a planner to quickly tell if it is likely to be able to handle the domain.

A keyword used in a :requirements field is called a requirement flag; the domain is said to declare a requirement for that flag. If a domain specifies no requirements, it is assumed a requirement `:strips`.

`:strips`	Basic STRIPS-style adds and deletes
`:typing`	Allow types in declarations of variables
`:negative-preconditions`	Allow `not` in precondition descriptions
`:disjunctive-preconditions`	Allow `or` in precondition descriptions
`:equality`	Support `=` as built-in predicate
`:universal-preconditions`	Allow `forall` in precondition descriptions
`:fluents`	`=:numeric-fluents + :object-fluents`
`:numeric-fluents`	Allow numeric function definitions and use of effects using assignment operators and arithmetic preconditions
`:conditional-effects`	Allow use of effects with `forall` operator
`:derived-predicates`	Allows predicates whose truth value is defined by a formula