Appendix: BNF Grammar of PDDL3.0

This is an appendix of the paper "Deterministic Planning in the 5th International Planning Competition: PDDL3 and Experimental Evaluation of the Planners" by Alfonso E. Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti and Yannis Dimopoulos. The parts marked in gray are the extensions of the language with respect to the previous versions of PDDL [1, 2, 3].

Domains

Domain descriptions have extended since PDDL2.1: firstly with derived predicated (PDDL2.2) and now with the option to add constraints. Note that both of these extensions have requirements flags.

```
<domain>                ::= (define (domain <name>)
                                [<require-def>]
                                [<types-def>]:typing
                                [<constants-def>]
                                [<predicates-def>]
                                [<functions-def>]:fluents
                                [<constraints>]
                                <structure-def>*)
<require-def>           ::= (:requirements <require-key>+)
<require-key>           ::= Any PDDL2.2 requirement (see [3])
<require-key>           ::= :preferences <require-key>*
<require-key>           ::= :constraints <require-key>*
<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)>)
<constraints>           ::= :constraints (:constraints <con-GD>)
<derived-def>           ::= (:derived <typed list (variable)> <GD>)
<structure-def>         ::= <action-def>
<structure-def>         ::=:durative−actions <durative-action-def>
<structure-def>         ::=:derived−predicates <derived-def>1
<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
<emptyOr (x)>             ::= ()
<emptyOr (x)>             ::= x
```

Actions

The BNF for an action definition is the same as in PDDL2.2, except that it has the additional option of attaching preferences to preconditions.

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

[1]This BNF is more permissive than is considered a well-formed description of derived predicates in PDDL2.2 and 3.0 [3].

```
<pre-GD>              ::= <pref-GD>
<pre-GD>              ::= (and <pre-GD>*)
<pre-GD>              ::= :universal-preconditions
                          (forall (<typed list(variable)>) <pre-GD>)
<pref-GD>             ::=:preferences (preference [<pref-name>] <GD>)
<pref-GD>             ::= <GD>
<pref-name>           ::= <name>
<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>           ::= <multi-op>
<binary-op>           ::= −
<binary-op>           ::= /
<multi-op>            ::= ∗
<multi-op>            ::= +
<binary-comp>         ::= >
<binary-comp>         ::= <
<binary-comp>         ::= =
<binary-comp>         ::= >=
<binary-comp>         ::= <=
<number>              ::= Any numeric literal
                         (integers and floats of form n.n).
<effect>        ::= (and <c-effect>*)
<effect>        ::= <c-effect>
<c-effect>      ::=:conditional-effects
                      (forall (<typed list (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
```

Durative Actions

The syntax for durative actions was introduced for PDDL2.1. It remains essentially unchanged, although conditions may now contain preferences. Preferences are always expressed outside the time specifier for a condition.

```
<durative-action-def> ::= (:durative-action <da-symbol>
                              :parameters (<typed list (variable)>)
                              <da-def body>)
<da-symbol>           ::= <name>
```

```
<da-def body>         ::= :duration <duration-constraint>
                          :condition <emptyOr (da-GD)>
                          :effect <emptyOr (da-effect)>
<da-GD>                ::= <pref-timed-GD>
<da-GD>                ::= (and <da-GD>*)
<da-GD>                ::=:universal−preconditions
                             (forall (<typed-list (variable)>) <da-GD>)
<pref-timed-GD>        ::= <timed-GD>
<pref-timed-GD>        ::=:preferences
                             (preference [<pref-name>] <timed-GD>)
<timed-GD>             ::= (at <time-specifier> <GD>)
<timed-GD>             ::= (over <interval> <GD>)
<time-specifier>       ::= start
<time-specifier>       ::= end
<interval>             ::= all
```

Duration constraints can take a wide range of forms, but have only be used in previous IPC domains in the simplest forms, where `:duration-inequalities` are not required. No preferences are supported in duration constraints. This is a simplification, but since duration inequalities have not been explored to any serious extent, it seems premature to add still more complication to this part of the language.

```
<duration-constraint>        ::= :duration−inequalities
                                 (and <simple-duration-constraint>+)
<duration-constraint>        ::= ()
<duration-constraint>        ::= <simple-duration-constraint>
<simple-duration-constraint> ::= (<d-op> ?duration <d-value>)
<simple-duration-constraint> ::= (at <time-specifier>
                                     <simple-duration-constraint>)
<d-op>                       ::=:duration−inequalities <=
<d-op>                       ::=:duration−inequalities >=
<d-op>                       ::= =
<d-value>                    ::= <number>
<d-value>                    ::=:fluents <f-exp>
```

```
<da-effect>       ::= (and <da-effect>*)
<da-effect>       ::= <timed-effect>
<da-effect>       ::=:conditional−effects
                        (forall (<typed list (variable)>) <da-effect>)
<da-effect>       ::=:conditional−effects (when <da-GD> <timed-effect>)
<da-effect>       ::=:fluents (<assign-op> <f-head> <f-exp-da>)
<timed-effect>    ::= (at <time-specifier> <a-effect>)
<timed-effect>    ::= (at <time-specifier> <f-assign-da>)
<timed-effect>    ::=:continuous−effects (<assign-op-t> <f-head> <f-exp-t>)
<f-assign-da>     ::= (<assign-op> <f-head> <f-exp-da>)
<f-exp-da>        ::= (<binary-op> <f-exp-da> <f-exp-da>)
<f-exp-da>        ::= (- <f-exp-da>)
<f-exp-da>        ::=:duration−inequalities ?duration
<f-exp-da>        ::= <f-exp>
```

Problems

Problems now have an optional additional field for constraints. These are added to those (if any) in the domain file and together they represent a collection of additional goals that must be satisfied by any valid plan. The difference between constraints and regular goals is that they may include conditions on trajectories rather than simply on the final state to which a plan leads.

```
<problem>              ::= (define (problem <name>)
                                   (:domain <name>)
                                   [<require-def>]
                                   [<object declaration> ]
                                   <init>
                                   <goal>
                                   [<constraints>]
                                   [<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 <pre-GD>)
<constraints>          ::=:constraints (:constraints <pref-con-GD>)
<pref-con-GD>          ::= (and <pref-con-GD>*)
<pref-con-GD>          ::=:universal-preconditions
                              (forall (<typed list (variable)>) <pref-con-GD>)
<pref-con-GD>          ::=:preferences (preference [<pref-name>] <con-GD>)
<pref-con-GD>          ::  <con-GD>
<con-GD>               ::= (and <con-GD>*)
<con-GD>               ::= (forall (<typed list (variable)>) <con-GD>)
<con-GD>               ::= (at end <GD>)
<con-GD>               ::= (always <GD>)
<con-GD>               ::= (sometime <GD>)
<con-GD>               ::= (within <number> <GD>)
<con-GD>               ::= (at-most-once <GD>)
<con-GD>               ::= (sometime-after <GD> <GD>)
<con-GD>               ::= (sometime-before <GD> <GD>)
<con-GD>               ::= (always-within <number> <GD> <GD>)
<con-GD>               ::= (hold-during <number> <number> <GD>)
<con-GD>               ::= (hold-after <number> <GD>)
<metric-spec>          ::= (:metric <optimization> <metric-f-exp>)
<optimization>         ::= minimize
<optimization>         ::= maximize
<metric-f-exp>         ::= (<binary-op> <metric-f-exp> <metric-f-exp>)
<metric-f-exp>         ::= (<multi-op> <metric-f-exp> <metric-f-exp>+)
<metric-f-exp>         ::= (- <metric-f-exp>)
<metric-f-exp>         ::= <number>
<metric-f-exp>         ::= (<function-symbol> <name>*)
<metric-f-exp>         ::= <function-symbol>
<metric-f-exp>         ::= total-time
<metric-f-exp>         ::= (is-violated <pref-name>)
```

The BNF is more permissive than is considered a well-formed problem description in PDDL3.0. 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 directly modify any of these: that is, like action effects, they are only allowed to affect the truth values of the basic (non-derived) predicates.

References

[1] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[2] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - the planning domain definition language. Technical Report CVC TR98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[3] J. Hoffmann and S. Edelkamp. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.