Complete BNF definition of MA-PDDL with privacy

Hereby a complete BNF syntax definition of MA-PDDL (Multi-Agent PDDL) extended with privacy is presented based on the complete and corrected BNF syntax definition of PDDL3.1 [1], the original definition of MA-PDDL [2] and privacy as defined in MA-STRIPS [3]. The minimalistic additions to PDDL3.1 syntax are highlighted separately each time. The additions to PDDL3.1 enabling description of **multiple agents** are highlighted with **yellow**, while the additions enabling the description of **privacy** are highlighted with **light blue**.

1. Domain description

```
<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>                     ::= See Section 7.2.3
<types-def>                       ::= (:types <typed list (name)>)
<constants-def>                   ::= (:constants <typed list (name)>)
<constants-def>                   ::=:factored-privacy                  (:constants <typed list (name)> [<private-objects>])
<constants-def>                   ::=:multi-agent + :unfactored-privacy (:constants <typed list (name)> <private-objects>*)
<private-objects>                 ::=:factored-privacy                  (:private <typed list (name)>)
<private-objects>                 ::=:multi-agent + :unfactored-privacy (:private <agent-def> <typed list (name)>)
<predicates-def>                  ::= (:predicates <atomic formula skeleton>+)
<predicates-def>                  ::=:factored-privacy                  (:predicates <atomic formula skeleton>+ [<private-predicates>])
<predicates-def>                  ::=:multi-agent + :unfactored-privacy (:predicates <atomic formula skeleton>+ <private-predicates>*)
<private-predicates>              ::=:factored-privacy                  (:private <atomic formula skeleton>+)
<private-predicates>              ::=:multi-agent + :unfactored-privacy (:private <agent-def> <atomic formula skeleton>+)
<atomic formula skeleton>         ::= (<predicate> <typed list (variable)>)
<atomic formula skeleton>         ::=:factored-privacy (<predicate> <typed list (term)>)
<predicate>                       ::= <name>
<variable>                        ::= ?<name>
<atomic function skeleton>        ::= (<function-symbol> <typed list (variable)>)
<atomic function skeleton>        ::=:factored-privacy (<function-symbol> <typed list (term)>)
<function-symbol>                 ::= <name>
<functions-def>                   ::=:fluents (:functions <function typed list (atomic function skeleton)>)
<functions-def>                   ::=:fluents + :factored-privacy                  (:functions <function typed list (atomic function skeleton)> [<private-functions>])
<functions-def>                   ::=:fluents + :multi-agent + :unfactored-privacy (:functions <function typed list (atomic function skeleton)> <private-functions>*)
<private-functions>               ::=:fluents + :factored-privacy                  (:private <function typed list (atomic function skeleton)>)
<private-functions>               ::=:fluents + :multi-agent + :unfactored-privacy (:private <agent-def> <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+
                                     This is deprecated since PDDL3.1, where the default fluent type is number.
<function type>                   ::=:numeric-fluents number
<function type>                   ::=:typing + :object-fluents <type>
<constraints>                     ::=:constraints (:constraints <con-GD>)
<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>
<primitive-type>                  ::= object
<type>                            ::= (either <primitive-type>+)
<type>                            ::= <primitive-type>
<emptyOr (x)>                     ::= ()
<emptyOr (x)>                     ::= x
<action-def>                      ::= (:action <action-symbol>
                                        [:agent <agent-def>]:multi-agent
                                        [:parameters (<typed list (variable)>)]
                                        <action-def body>)
<action-symbol>                   ::= <name>
<agent-def>                       ::= <name>
<agent-def>                       ::= <variable>
<agent-def>                       ::=:typing <type>
<agent-def>                       ::=:typing <variable> - <type>
<action-def body>                 ::= [:precondition <emptyOr (pre-GD)>]
                                      [:effect <emptyOr (effect)>]
<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>                              ::=:multi-agent <action formula(term)>
```

```
<GD>                                  ::=:multi-agent + :negative-preconditions (not <action formula(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>                                  ::=:numeric-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*)
<atomic formula(t)>                   ::=:equality (= t t)
<action formula(t)>                   ::= (<action-symbol> t t*)
<action formula(t)>                   ::=:durative-actions (<da-symbol> t t*)
<term>                                ::= <name>
<term>                                ::= <variable>
<term>                                ::=:object-fluents <function-term>
<function-term>                       ::=:object-fluents (<function-symbol> <term>*)
<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 + :multi-agent (num (<typed list (variable)>) <emptyOr (GD)> <action formula(term)>)
<f-exp>                               ::=:numeric-fluents (- <f-exp>)
<f-exp>                               ::=:numeric-fluents <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>                         ::= <=
<name>                                ::= <letter> <any char>*
<letter>                              ::= a..z | A..Z
<any char>                            ::= <letter> | <digit> | - | _
<number>                              ::= <digit>+ [<decimal>]
<digit>                               ::= 0..9
<decimal>                             ::= .<digit>+
<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>                            ::= (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)
<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-action-def>                 ::= (:durative-action <da-symbol>
                                                [:agent <agent-def>]:multi-agent
                                                [: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-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>                             ::=:numeric-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>)
<timed-effect>                 ::= (at <time-specifier> <cond-effect>)
<timed-effect>                 ::=:numeric-fluents (at <time-specifier> <f-assign-da>)
<timed-effect>                 ::=:continuous-effects + :numeric-fluents (<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>                     ::= (<multi-op> <f-exp-da> <f-exp-da>+)
<f-exp-da>                     ::= (- <f-exp-da>)
<f-exp-da>                     ::=:duration-inequalities ?duration
<f-exp-da>                     ::= <f-exp>
<assign-op-t>                  ::= increase
<assign-op-t>                  ::= decrease
<f-exp-t>                      ::= (* <f-exp> #t)
<f-exp-t>                      ::= (* #t <f-exp>)
<f-exp-t>                      ::= #t
<derived-def>                  ::= (:derived <atomic formula skeleton> <GD>)
```

2. Problem description

```
<problem>                      ::= (define (problem <name>)
                                       (:domain <name>)
                                       [<require-def>]
                                       [<object declaration>]
                                       <init>
                                       <goal>+
                                       [<constraints>]:constraints
                                       <metric-spec>*:numeric-fluents
                                       [<length-spec>])
<object declaration>           ::= (:objects <typed list (name)>)
<object declaration>           ::=:factored-privacy                    (:objects <typed list (name)> [<private-objects>])
<object declaration>           ::=:multi-agent + :unfactored-privacy   (:objects <typed list (name)> <private-objects>*)
<init>                         ::= (:init <init-el>*)
<init-el>                      ::= <literal(name)>
<init-el>                      ::=:timed-initial-literals (at <number> <literal(name)>)
<init-el>                      ::=:numeric-fluents (= <basic-function-term> <number>)
<init-el>                      ::=:object-fluents (= <basic-function-term> <name>)
<basic-function-term>          ::= <function-symbol>
<basic-function-term>          ::= (<function-symbol> <name>*)
<goal>                         ::= (:goal <pre-GD>)
<goal>                         ::=:multi-agent (:goal
                                                   [:agent <agent-def>]
                                                   :condition <emptyOr (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>                  ::=:numeric-fluents (:metric <optimization> <metric-f-exp>)
<metric-spec>                  ::=:multi-agent + :numeric-fluents (:metric
                                                                      [:agent <agent-def>]
                                                                      :utility <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>                 ::=:multi-agent (<function-symbol> <term>*)
<metric-f-exp>                 ::= <function-symbol>
<metric-f-exp>                 ::= total-time
<metric-f-exp>                 ::=:preferences (is-violated <pref-name>)
<length-spec>                  ::= (:length [(:serial <integer>)] [(:parallel <integer>)])
                                   The length-spec is deprecated since PDDL 2.1.
```

2.1 Lifting restrictions (from constraint declaration)

If we wish to embed modal operators into each other, then we should use these rules instead of those in Section 2 respectively.

```
<con-GD>                    ::= (always <con2-GD>)
<con-GD>                    ::= (sometime <con2-GD>)
<con-GD>                    ::= (within <number> <con2-GD>)
<con-GD>                    ::= (at-most-once <con2-GD>)
<con-GD>                    ::= (sometime-after <con2-GD> <con2-GD>)
<con-GD>                    ::= (sometime-before <con2-GD> <con2-GD>)
<con-GD>                    ::= (always-within <number> <con2-GD> <con2-GD>)
<con-GD>                    ::= (hold-during <number> <number> <con2-GD>)
<con-GD>                    ::= (hold-after <number> <con2-GD>)
<con2-GD>                   ::= <con-GD>
<con2-GD>                   ::= <GD>
```

2.2 Requirements

Here is a table of all requirements in MA-PDDL extended with privacy. Some requirements imply others; some are abbreviations for common sets of requirements. If a domain stipulates no requirements, it is assumed to declare `:strips`.

`:strips`	Basic STRIPS-style adds and deletes
`:typing`	Allows type names in declarations of variables
`:negative-preconditions`	Allows `not` in goal descriptions
`:disjunctive-preconditions`	Allows `or` in goal descriptions
`:equality`	Support `=` as built-in predicate
`:existential-preconditions`	Allows `exists` in goal descriptions
`:universal-preconditions`	Allows `forall` in goal descriptions
`:quantified-preconditions`	= `:existential-preconditions` + `:universal-preconditions`
`:conditional-effects`	Allows `when` in action effects
`:fluents`	= `:numeric-fluents` + `:object-fluents`
`:numeric-fluents`	Allows numeric function definitions and use of effects using assignment operators and arithmetic preconditions.
`:object-fluents`	Allows object function definitions similarly to `:numeric-fluents`.
`:adl`	= `:strips` + `:typing` + `:negative-preconditions` + `:disjunctive-preconditions` + `:equality` + `:quantified-preconditions` + `:conditional-effects`
`:durative-actions`	Allows durative actions. Note that this does not imply `:numeric-fluents`.
`:duration-inequalities`	Allows duration constraints in durative actions using inequalities.
`:continuous-effects`	Allows durative actions to affect fluents continuously over the duration of the actions.
`: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`
`:preferences`	Allows use of preferences in action preconditions and goals.
`:constraints`	Allows use of constraints fields in domain and problem files. These may contain modal operators supporting trajectory constraints.
`:action-costs`	If this requirement is included in a PDDL specification, the use of numeric fluents is enabled (similar to the `:numeric-fluents` requirement). However, numeric fluents may only be used in certain very limited ways: 1. Numeric fluents may not be used in any conditions (preconditions, goal conditions, conditions of conditional effects, etc.). 2. A numeric fluent may only be used as the target of an effect if it is 0-ary and called `total-cost`. If such an effect is used, then the `total-cost` fluent must be explicitly initialized to 0 in the initial state. 3. The only allowable use of numeric fluents in effects is in effects of the form `(increase (total-cost) <numeric-term>)`, where the `<numeric-term>` is either a non-negative numeric constant or of the form `(<function-symbol> <term>*)`. (The `<term>` here is interpreted as shown in the PDDL grammar, i.e. it is a variable symbol or an object constant. Note that this `<term>` cannot be a `<function-term>`, even if the object fluents requirement is used.) 4. No numeric fluent may be initialized to a negative value. 5. If the problem contains a `:metric` specification, the objective must be `(minimize (total-cost))`, or - only if the `:durative-actions` requirement is also set - to minimize a linear combination of `total-cost` and `total-time`, with non-negative coefficients. Note that an action can have multiple effects that increase `(total-cost)`, which is particularly useful in the context of conditional effects. Also note that these restrictions imply that `(total-cost)` never decreases throughout plan execution, i.e., action costs are never negative.
`:multi-agent`	Allows declaration of multi-agent planning domains and problems with different agents having possibly different actions, goals and metrics, where concurrent/joint actions may interact.
`:unfactored-privacy`	Allows the declaration of private (or internal) propositions (or atoms, grounded predicates) along the lines of MA-STRIPS [3]. The idea is to allow the declaration of private predicates/constants/objects on a per agent basis. So the `:multi-agent` requirement should also be declared for this requirement to have an effect. A non-private constant/object is public. A proposition (or atom) is public if and only if it refers to a non-private predicate and each argument of the proposition is public. Otherwise the proposition (or atom) is private. The arguments of a private proposition (or atom) of an agent can include only either public constants/objects or private constants/objects of the same agent. `:unfactored-privacy` is mutually exclusive with `:factored-privacy`.
`:factored-privacy`	Allows the declaration of private (or internal) propositions (or atoms, grounded predicates) along the lines of MA-STRIPS [3] for MA-PDDL descriptions factored from unfactored MA-PDDL descriptions, on a per agent basis. Thus the difference compared to `:unfactored-privacy` is that in this case the MA-PDDL description does not contain the private predicates/constants/objects of other agents (it is assumed that they are not observable). A factored description is single-agent. Only the private predicates/constants/objects of the agent for which the MA-PDDL description was factored are indicated. A non-private constant/object is public. A proposition (or atom) is public if and only if it refers to a non-private predicate and each argument of the proposition is public, otherwise it is private. This requirement is exclusive with `:unfactored-privacy`.

3. References

[1] D. L. Kovacs (2011). BNF definition of PDDL3.1, unpublished manuscript from the IPC-2011 and IPC-2014 website, http://helios.hud.ac.uk/scommv/IPC-14/repository/kovacs-pddl-3.1-2011.pdf

[2] D. L. Kovacs (2012). A Multi-Agent Extension of PDDL3.1. In Proc. of the 3rd Workshop on the International Planning Competition (IPC), ICAPS-2012, Atibaia, Sao Paulo, Brazil, pp. 19-27, http://home.mit.bme.hu/~dkovacs/pubs/d.l.kovacs_2012_ICAPS-WIPC.pdf

[3] R. I. Brafman and C. Domshlak (2008). From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In Proc. of ICAPS-08, AAAI Press, pp. 28-35. http://www.aaai.org/Papers/ICAPS/2008/ICAPS08-004.pdf