Converting multi-agent PDDL to extensive form games

D. L. Kovacs [1]) and T. P. Dobrowiecki [1])

[1]) Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, Department of Measurement and Information Systems, Hungary
dkovacs@mit.bme.hu, tade@mit.bme.hu

Abstract:

This paper presents conversion methods of fully- and partially-observable multi-agent planning problems described in the Multi-Agent Planning Domain Definition Language (MA-PDDL) to extensive-form games. MA-PDDL and extensive-form games model essentially the same multi-agent situations, but the former is much more detailed. The proposed conversion is fruitful in both directions: 1) extensive games can be solved via available game theoretic solution principles providing solutions to corresponding MA-PDDL models, and 2) MA-PDDL can be solved via multi-agent planning methods providing solutions to the corresponding game.
Keywords: multi-agent pddl planning game-theory partial-observability

1. Introduction

This paper is about the conversion of formal models of multi-agent planning problems [1] described in MA-PDDL (Multi-Agent Planning Domain Definition Language) [2] to extensive-form games [3]. MA-PDDL is a multi-agent extension of single-agent PDDL [4]. It provides a detailed description of multi-agent environments. Extensive-form games on the other hand are abstract game theoretic constructs modeling essentially the same multi-agent situations. To our knowledge there are currently no other methods for automatically converting multi-agent problem descriptions to game theoretic games in the literature.

The proposed connection of MA-PDDL and extensive form games is fruitful in both directions: **(1)** an extensive-form game can be solved via available game theoretic solution principles (e.g. by finding its Nash-equilibria [5]) providing solutions to the corresponding multi-agent planning problem described in MA-PDDL or **(2)** MA-PDDL can provide a much richer description of the same game theoretic situation, and solved with available multi-agent planning methods.

The paper is structured as follows: Section 2 introduces the preliminaries of PDDL, MA-PDDL and extensive-form games. Section 3 proposes a partial-observability extension to MA-PDDL, and conversion methods of fully- and partially-observable MA-PDDL descriptions to extensive-form games. Finally, Section 4 concludes the work and outlines future research directions.

2. Preliminaries

In the following fundamentals of PDDL, MA-PDDL and game theory are provided.

2.1. PDDL

The latest version of PDDL is 3.1 [6,7]. Each new version of the language added new, modular features to previous versions. PDDL3.1 has the following features: the description of a single-agent planning problem is divided in two parts: a domain- and a problem-description. The former holds those model-elements which are present in every planning problem of the domain, while the latter specifies the concrete planning problem at hand within the domain. Thus the input of a domain-independent PDDL-based planner is the domain- and problem-description, and its output is a plan that solves the specified planning problem.

The domain description consists of a name, a list of requirements (list of used features), a type-hierarchy (classifying objects), constants (objects present in every problem of the domain), a list of predicates and actions. Actions have input parameters, preconditions (that need to be satisfied in a given state of the environment for the action to be executable) and effects (describing the change to the state where the action is executed). Effects of an action can be conditional or continuous. Moreover, actions may have arbitrary, non-unit duration. A domain description may also include a list of functions, derived predicates or hard constraints. The domain of a function is a Cartesian product of object-types, while its range may be either the set of real numbers or any object-type. A derived predicate is true, if its preconditions are true. Constraints are statements in modal logic about state-trajectories that must be true for valid solution plans.

The problem description consists of a name, a reference to its domain, a list of all objects, an initial state and goal states of the environment. It can include a metric (a real-valued function for measuring the quality of solutions), timed initial literals (facts becoming true at a given time) and constraints similarly to the domain description, but they can refer to preferences (soft constraints, which should not necessarily be satisfied, but they can be incorporated in the metric). Preferences can also be defined in the goal description or in preconditions of actions.

2.2. MA-PDDL

MA-PDDL [2] is a minimalistic extension to PDDL3.1 allowing planning *by* and *for* multiple agents. It can distinguish between different agents' possibly different actions. Similarly different agents can have different goals or metrics. Actions' preconditions can directly refer to concurrent actions and thus actions with interacting effects can be handled in general (e.g. when at least 2 agents need to execute the `lift` action to lift a heavy table, or it will remain on the ground). However, since PDDL3.1 assumes that the environment is fully-observable (i.e. every agent can access the value of every state-fluent at every instant and observe every action), thus by default this is assumed in MA-PDDL too. The PDDL-requirement for the multi-agent extension of PDDL is `:multi-agent`.

2.3. Game theoretic fundamentals

An incomplete information game [8] (the most general non-cooperative game) is described by a 5-tuple $\Gamma = (N, \{S_i\}_{i\in N}, \{u_i\}_{i\in N}, \{T_i\}_{i\in N}, p)$. This is the normal form of game Γ, where $N = \{1,2,\dots,n\}$ denotes the set of agents; S_i is the set of pure strategies of $i \in N$ and T_i is the set of its types; $u_i: S \times T_i \to \mathbb{R}$ is the real-valued utility function of agent i, where $S = \prod_{i=1}^{n} S_i$ is the set of all strategy-combinations.

The goal of an agent is to choose its strategy so as to maximize its own expected utility. The difficulty is that agents choose their strategies simultaneously and independently. Moreover each agent i plays with an active type, $t_i \in T_i$, which is revealed only to i, and chosen randomly by Nature (or Chance) at the beginning of each play. p is the a priori probability distribution above all type-combinations $t \in T = \prod_{i=1}^{n} T_i$ according to which Nature chooses active types for agents. A type-combination $t \in T$ is thus realized with probability $p(t)$. If there is only 1 type-combination, i.e. when $|T| = 1$, then Γ is of complete information. Otherwise, when $|T| > 1$, Γ is of incomplete information. In any case Γ is common knowledge among the agents (every agent knows, that every agent knows, that ... knows Γ).

The extensive form of Γ adds the notion of choice nodes $\omega \in \Omega$, where Ω is the finite set of all choice nodes with a distinguished initial choice node, $\omega_0 \in \Omega$, from where each play of Γ begins. A function $g: \Omega \to N \cup \{0\}$ can indicate which agent $i = g(\omega)$ chooses an elementary move (or action) in $\omega \in \Omega$ from the finite, non-empty set of its moves, A_i (one and only one agent is associated to each $\omega \in \Omega$). Similarly function $h: \Omega \to 2^{\cup_{i=0}^{n} A_i} \setminus \{\emptyset\}$ may indicate the set of those moves, $h(\omega) \subseteq A_{g(\omega)}$, which agent $g(\omega)$ can choose in ω (one and only one move can be chosen in each ω). Thus in an incomplete information game $g(\omega_0) = 0$ and $h(\omega_0) = T$ holds. Agent 0 represents Nature (or Chance).

In any given $\omega \in \Omega$ node, where $g(\omega) = 0$ holds, agent 0 chooses its respective moves randomly according to a probability distribution $s_0(\omega)$, where $s_0: \Omega \to \Delta(A_0)$ denotes the stochastic strategy of agent 0, and $\Delta(A_0)$ is the set of all probability distributions above A_0. It follows that $s_0(\omega_0) = p$ holds for ω_0. Eventually each choice node corresponds to a unique sequence of moves of length between 0 and $\kappa \in \mathbb{Z}^+$ (a given maximum), with ω_0 corresponding to the empty sequence $\mathcal{E}$ of length 0. So a play begins initially in ω_0. Then, after agent $g(\omega_0)$ choses a move $a_{g(\omega_0)} \in A_{g(\omega_0)}$, the play continues in ω_1 corresponding to the sequence $\langle a_{g(\omega_0)} \rangle$. This continues until the play reaches a sequence $\langle a_{g(\omega_0)}, a_{g(\omega_1)}, \dots, a_{g(\omega_{\kappa-1})} \rangle$. Thus choice nodes can be connected in a tree-graph G of maximal depth κ with ω_0 being the root-node.

Agents can't necessarily observe all the previous moves of other agents during a play. For this reason information functions $P_i: \Omega \to 2^{\Omega} \setminus \{\emptyset\}$ are introduced for each $i = 1,2,\dots,n$. The information function of agent i associates a non-empty information set, $P_i(\omega) \subseteq \Omega$, to each choice node ω, where $i = g(\omega)$. An information set $P_i(\omega)$ denotes the set of those choice nodes that agent i thinks possible in ω. It is assumed that $\omega \in P_i(\omega)$ holds for every $\omega \in \Omega$ and $i \in N$ and

that $\forall \omega', \omega'' \in P_i(\omega): h(\omega') = h(\omega'')$. Thus the choice nodes inside an information set are indistinguishable for the respective agent. Information sets of agent i are disjoint, forming an information partition $\mathbf{P}_i = \bigcup_{\omega \in \Omega, g(\omega)=i} P_i(\omega)$. Now the set of pure strategies S_i of agent $i = 1,2,\dots,n$ in Γ is the set of all $s_i: \mathbf{P}_i \to A_i$ functions, where for $\forall P_i(\omega) \in \mathbf{P}_i \quad s_i\big(P_i(\omega)\big) \in h(\omega)$ holds. Eventually an incomplete information extensive-form game is thus a 15-tuple: $\Gamma = \big(N, \{S_i\}_{i \epsilon N}, \{u_i\}_{i \epsilon N}, \{T_i\}_{i \epsilon N}, p, \Omega, \omega_0, s_0, \{\mathbf{P}_i\}_{i \epsilon N}, \{P_i\}_{i \epsilon N}, \{A_i\}_{i \epsilon N \cup \{0\}}, g, h, \kappa, G\big)$.

3. Conversion of MA-PDDL to extensive form games

Now the main results of this paper are presented: algorithms for converting fully- and partially-observable MA-PDDL descriptions to extensive-form games.

3.1. Case of full-observability

The idea of the conversion method is to generate successor states from the initial state of an MA-PDDL problem, $PROB$, in every possible way (i.e. via every applicable action-combination of agents, including `no-op` actions, with every agent executing one action at a time), and then recursively apply the same process to the resulting states altogether k-times, and convert this graph into an extensive-form game. **Algorithm 1** forms the backbone of this method.

Algorithm 1: Convert a fully-observable MA-PDDL description to an extensive form game

CONVERT$(PROB, k)$
  $l \leftarrow 0$
  $N \leftarrow$ **AGENT_OBJECTS**$(PROB)$
  $n = |N|, \quad \kappa = 1 + k \cdot n$
  foreach $i \in N$
  | $T_i \leftarrow \{t_i \leftarrow$ **NEW_TYPE**$()\}, \quad \mathbf{P}_i \leftarrow \emptyset$
  | $A_i \leftarrow$ **ALL_GROUNDED_ACTIONS**$(PROB, i) \cup \{$`no-op`$\}$
  end-foreach
  $\omega_0 \leftarrow$ **NEW_CHOICE_NODE**$(), \quad g(\omega_0) \leftarrow 0, \quad A_0 \leftarrow \{t = (t_1, t_2, \dots, t_n)\}, \quad h(\omega_0) \leftarrow A_0$
$p\big(t = (t_1, t_2, \dots, t_n)\big) \leftarrow 1, \quad s_0(\omega_0) \leftarrow p$
$\omega_1 \leftarrow$ **CHOICE_NODE_FROM_INITIAL_STATE**$(PROB)$
$g(\omega_1) \leftarrow 1, \quad h(\omega_1) \leftarrow A_1, \quad P_1(\omega_1) \leftarrow \{\omega_1\}, \quad \mathbf{P}_1 \leftarrow \mathbf{P}_1 \cup \{P_1(\omega_1)\}$
$\Omega \leftarrow \{\omega_0, \omega_1\}, \quad G \leftarrow (\Omega, \{(\omega_0, t, \omega_1)\}), \quad state\_level_0 \leftarrow \{\omega_1\}$
while $(l < k)$
| $l \leftarrow l + 1$
| $\langle G, \Omega, \{\mathbf{P}_i\}_{i \epsilon N}, \{P_i\}_{i \epsilon N}, state\_level_l \rangle \leftarrow$
| **ADD_NEXT_LEVEL**$(PROB, G, \omega_0, \Omega, \{\mathbf{P}_i\}_{i \epsilon N}, \{P_i\}_{i \epsilon N}, state\_level_{l-1}, \{A_i\}_{i \epsilon N}, n)$
end-while
$\{S_i\}_{i \epsilon N} \leftarrow$ **ENUMERATE_STRATEGIES**$(N, \{\mathbf{P}_i\}_{i \epsilon N}, h, \{A_i\}_{i \epsilon N})$
$\{u_i\}_{i \epsilon N} \leftarrow$ **GET_METRIC_VALUES**$(PROB, N, \{S_i\}_{i \epsilon N}, \{T_i\}_{i \epsilon N}, p, \omega_0, s_0, G, state\_level_k)$
return $\Gamma = \big(N, \{S_i\}_{i \epsilon N}, \{u_i\}_{i \epsilon N}, \{T_i\}_{i \epsilon N}, p, \Omega, \omega_0, s_0, \{\mathbf{P}_i\}_{i \epsilon N}, \{P_i\}_{i \epsilon N}, \{A_i\}_{i \epsilon N \cup \{0\}}, g, h, \kappa, G\big)$

The **CONVERT** method has 2 inputs **(#1)**: $PROB$ is a fully-observable, discrete, deterministic MA-PDDL domain- and problem-description, and $k \geq 0$ is a positive integer specifies the number of levels of successor states generated. In case of $n = |N|$ agents the resulting extensive-form game Γ **(#21)** has a tree-graph G of depth $\kappa = 1 + k \cdot n$ **(#4)**, where N is the set of agent-objects in $PROB$ **(#3)**.

The algorithm first sets a level-counter l to zero **(#2)**, then for every agent it initializes the set of types to a one-element set (since a deterministic MA-PDDL description is converted to a complete information game). Information partition $\mathbf{P}_i$ is set to the empty-set for every $i \in N$, and all grounded actions of agent i are extracted from $PROB$ into respective sets of moves, A_i, including the always executable `no-op` action, standing for no-operation and having no preconditions or effects **(#5-8)**. Next **(#9)** the root node of the game-tree, ω_0, is created, and its actor is set to agent 0, the actions of agent 0 are set to A_0 (having only one element, the only type-combination, t), and A_0 is allowed in ω_0. Then **(#10)** the probability of "action" t, $p(t)$, is set to 1, so this degenerate probability distribution will govern the stochastic strategy of agent 0 in ω_0, i.e. $s_0(\omega_0)$ is set to p.

Next **(#11)** the **`CHOICE_NODE_FROM_INITIAL_STATE`** method creates a new choice node, ω_1, which corresponds to the initial state of $PROB$. Agent 1 is set to act in ω_1 **(#12)**, allowing any move from A_1. Line **(#12)** initializes also the information set $P_1(\omega_1)$ and information partition $\mathbf{P}_1$ of agent 1 to $\{\omega_1\}$, a one-element set (because of full-observability). Line **(#13)** initializes the set of choice nodes, Ω, to include only ω_0 and ω_1; and the game-graph G to have these nodes as vertices with only one edge – labeled with move t –, (ω_0, t, ω_1), and then also the 0th state-level is initialized to a set having only a single element, ω_1.

State-levels are a central importance. They consist of those choice nodes in G, which correspond directly to states of the multi-agent environment. The following 5 lines **(#14-18)** create new state-levels via intermediate action-levels by calling the **`ADD_NEXT_LEVEL`** method iteratively in a `while`-loop. This method extends the game-graph, the set of choice-nodes, the information functions and partitions of agents, and also creates the next state-level. The detailed pseudo-code of the method is shown in Algorithm 2. After k iterations the `while`-loop exits, and the finalized information partitions of agents, $\{\mathbf{P}_i\}_{i \in N}$, are used to enumerate **(#19)** all the possible $s_i: \mathbf{P}_i \to A_i$ functions (for every $i \in N$) to form the sets of pure strategies, $\{S_i\}_{i \in N}$. This is done by the **`ENUMERATE_STRATEGIES`** method. However $\{S_i\}_{i \in N}$ is implied implicitly by the game-graph and information partitions.

The utility of agents is defined explicitly for every possible outcome (i.e. for every respective strategy-combination). These outcomes are represented in the game-tree with choice-nodes of the last state-level. Each of them corresponds to exactly one k-step state/action-trajectory, thus the idea is to simply get the MA-PDDL metric-value of these state/action-trajectories from $PROB$ for every agent-object, and associate them to the respective choice-nodes. This way each choice-node in the last state-level will have an n-long utility-vector. This is what the **`GET_METRIC_VALUES`** method does **(#20)**. Finally we can summarize Γ, having a graph G of depth κ, abstractly encoding the MA-PDDL problem, $PROB$, up to a finite horizon k. The algorithm returns Γ **(#21)**.
The heart of the above presented **`CONVERT`** method is the iterative call of the **`ADD_NEXT_LEVEL`** method, which effectively builds the game-tree, level-by-level **(#16-17)**. This method is presented in **Algorithm 2**.

Algorithm 2: Add a level to the extensive game-tree of a fully-observable MA-PDDL desc.

```
1: ADD_NEXT_LEVEL(PROB, G = (V, E), ω_0, Ω, {P_i}_{i∈N}, {P_i}_{i∈N}, last_state_level, {A_i}_{i∈N}, n)
2:   next_state_level ← ∅
3:   foreach ω ∈ last_state_level
4:   | action_level_1 ← {ω},  action_level_{n+1} ← ∅
5:   | TRACE(ω) ← ε,  ω' ← CLONE(ω),  Ω ← Ω ∪ {ω'}
6:   | g(ω') ← 1,  h(ω') ← A_1,  P_1(ω') ← {ω'},  P_1 ← P_1 ∪ {P_1(ω')}
7:   | for i = 1,  i ≤ n,  i++
8:   | | if i > 1 then P_i ← P_i ∪ {action_level_i} end-if
9:   | | if i < n then action_level_{i+1} ← ∅ end-if
10:  | | foreach x ∈ action_level_i
11:  | | | if i > 1 then P_i(x) ← action_level_i end-if
12:  | | | foreach a_i ∈ h(x)
13:  | | | | if i < n then
14:  | | | | | y ← NEW_CHOICE_NODE()
15:  | | | | | TRACE(y) ← (TRACE(x), a_i)
16:  | | | | | g(y) ← i + 1,  h(y) ← A_{i+1}
17:  | | | | else
18:  | | | | | if HAS_CONSISTENT_EXECUTABLE_SUBSET((TRACE(x), a_i), PROB, ω_0, ω, G) then
19:  | | | | | | y ← CHOICE_NODE_FROM_SUCCESSOR_STATE(ω, (TRACE(x), a_i), PROB, ω_0, G)
20:  | | | | | | g(y) ← 1,  h(y) ← A_1,  P_1(y) ← {y},  P_1 ← P_1 ∪ {P_1(y)}
21:  | | | | | else
22:  | | | | | | y ← ω'
23:  | | | | | end-if
24:  | | | | end-if
25:  | | | | action_level_{i+1} ← action_level_{i+1} ∪ {y}
26:  | | | | V ← V ∪ {y},  E ← E ∪ {(x, a_i, y)},  Ω ← Ω ∪ {y}
27:  | | | end-foreach
28:  | | end-foreach
29:  | end-for
30:  | next_state_level ← next_state_level ∪ action_level_{n+1}
31: end-foreach
32: return ⟨G, Ω, {P_i}_{i∈N}, {P_i}_{i∈N}, next_state_level⟩
```

The **ADD_NEXT_LEVEL** method has 9 inputs **(#1)**: $PROB$ is the fully-observable, discrete, deterministic MA-PDDL description; G is the actual game-graph (a set of vertices, V, and a set of labeled edges, E); ω_0 is the root node of G; Ω is the actual set of choice-nodes; $\{\mathbf{P}_i\}_{i\in N}$ and $\{P_i\}_{i\in N}$ are the actual information partitions and functions of agents; $last\_state\_level$ is the latest state-level; $\{A_i\}_{i\in N}$ is the set of sets of all the possible moves of agents; and n is the number of agents. First **(#2)** the next state-level is initialized to an empty-set, and then it is gradually built in a `foreach`-loop **(#3-31)**, which goes through every $\omega \in last\_state\_level$ node, and grows a sub-tree of moves from it, one move for each agent, starting from agent 1 until agent n **(#7-29)**, in every possible way.

The levels of sub-trees are called action-levels, and the choice nodes of an action-level belong to the same information set (of the respective actor agent), since agents act simultaneously in every instant and thus they cannot observe each other's moves. However each information set at state-levels (where agent 1 acts) consists of one choice node because of full-observability. The sub-tree built from ω in the `for`-loop **(#7-29)** has $n+1$ levels, level 1 being part of $last\_state\_level$ and level $n+1$ being part of the $next\_state\_level$. The latter consists of choice nodes that correspond to successor states of the environment,

produced in every possible way from the state corresponding to ω. So the `for`-loop **(#7-29)** creates action-levels $i = 2, 3, \dots, n+1$ of the sub-tree. Inside it a `foreach`-loop **(#10-28)** goes through every choice node x of action-level i, and a further `foreach`-loop inside it **(#12-27)** goes through every possible move $a_i \in h(x)$ of agent i supposing that the previous $i-1$ agents chose action-combination **TRACE**(x). **TRACE**(ω) is initially empty. Further choice nodes of the game-tree are updated with the move-path (trace) which leads to them from ω.

If $i < n$ **(#13-16)**, then the possible action-combination is not yet ready, so a new choice node y is created in action-level $i+1$, and its trace, actor and move-set is set. Otherwise, if $i = n$ **(#18-24)**, then $(\textbf{TRACE}(x), a_i)$ is an n-element action-combination, which may have an executable subset in the state corresponding to ω in light of the generated state-trajectory. This is checked by the **HAS_CONSISTENT_EXECUTABLE_SUBSET** method **(#18)** in 5 steps: **(a)** first it collects those actions from $(\textbf{TRACE}(x), a_i)$ which are potentially executable in the state corresponding to ω. A single-action is considered potentially executable in a state if its pre-conditions are satisfied taking **Assumption 1** into account.

Assumption 1 (undefined effects). If the executability of a grounded action a in a state requires the concurrent execution (or no execution) of some actions, then if any of those actions are not executed (or executed) concurrently with a, then we assume that a still remains executable, but it has no effects (empty effects).

Assumption 1 covers the case when an MA-PDDL action is defined to refer to an action in its pre-conditions, but no effects are specified for the case, when that reference is negated. For example if a `pick-up` action of agent i requires in its pre-conditions, that no other agent executes `pick-up` for the same object at the same time, but the Designer did not specify what happens if still at least one other agent does so, then if this actually happens, then the `pick-up` action of agent i still remains executable, but has no effects. I.e. in light of Assumption 1 the executability of actions is independent from concurrently executed actions.

After collecting potentially executable single-actions from $(\textbf{TRACE}(x), a_i)$**, (b)** all single- and joint-actions are identified within the collection. A joint-action within an action-collection C in ω is a subset of C, where all members either refer to at least one other member in their pre-conditions or the conditions of their active conditional effects, or they are referred to by at least one of the other members. Conditional effects are active in ω if their conditions are satisfied in the state corresponding to ω with actions C being executed. This approach produces an unambiguous partition of C. Next **(c)** individually inconsistent or not executable elements are removed from the produced partition. A single-action is individually consistent in ω if its active conditions and effects are both consistent in (the state corresponding to) ω. A joint-action is individually consistent in ω if its joint active conditions and joint active effects are both consistent in (the state corresponding to) ω in case the actions within the joint-action are executed simultaneously. Interference of conditions and effects of concurrent discrete actions is not considered. Individual executability requires satisfaction of (joint) pre-conditions.

After removing individually inconsistent or not executable elements from the partition, **(d)** elements with pairwise inconsistent joint-effects are removed. Finally **(e)** the remaining elements are checked, whether their joint execution is allowed by the hard state-trajectory constraints in $PROB$. **If yes**, then these actions are the consistent and executable subset of $(\mathbf{TRACE}(x), a_i)$, i.e. they can be executed, and so the **HAS_CONSISTENT_EXECUTABLE_SUBSET** method returns `true`. Otherwise, **if not**, or if the subset is empty, then the return-value is `false`. In case the return-value is `true`, a new choice node y corresponding to the state produced by the consistent and executable subset is created and put into action-level $n+1$ **(#19,20,25)**. Otherwise nothing happens, i.e. a choice node $y = \omega' = \omega$ is put into action-level $n+1$ **(#5,22,25)**. In both cases G and Ω is updated appropriately **(#26)** and action-level $n+1$ is added to the $next\_state\_level$ **(#30)**. This is repeated for every ω in $last\_state\_level$ **(#3-31)** before Alg.2 finishes.

3.2. Case of partial-observability

To capture partial-observability the following 6 rules should be **added** to the BNF (Backus-Naur Form) grammar of MA-PDDL (they should be read in conjunction).

```
<structure-def>    ::=:partial-observability <observation-def>
<observation-def>  ::=
                (:observation <observation-symbol>
                     [:agent <agent-def>]:multi-agent
                     [:parameters (<typed list (variable)>)]
                     [:condition <emptyOr (pre-GD)>]
                     [:value <emptyOr (observation-value)>])

<observation-symbol> ::= <name>
<observation-value>  ::=:numeric-fluents <f-exp>
<observation-value>  ::=:numeric-fluents + :continuous-effects <f-exp-t>
<observation-value>  ::=:object-fluents <function-term>
```

The above extension has essentially the same semantics as events proposed in [9] except that observations can be defined for multiple agents (their inheritance and polymorphism is the same as of actions in [2]), and they can have a `:value` field instead of `:effects`. A grounded observation holds in states where its conditions are satisfied, but it can't be referred to in conditions or effects of actions or anywhere else. Observations can be used solely by the planners. In case the `:partial-observability` requirement is present, it is assumed, that agents modeled with agent-objects can access only their observations (and their value), but they can't access state-fluents (facts, functions' value, and actions) directly. Planners planning for agent-objects should take this into account. Moreover, in this case information sets at state-levels of the converted game may not be singular, since there may be state/action-trajectories, where the observation-history (including observation of actions) is the same for an agent, and thus the choice-nodes corresponding to these trajectories should be members of the same information set. Based on this remark Alg.1 should be extended. First the below 3 lines should be inserted between line **#12** and **#13**.

```
13: foreach i ∈ N,
14: | OBS_HIST(i, ω1) ← ⟨OBS(i, ω1)⟩, Qi(ω1) ← {ω1}, Qi,0 ← {Qi(ω1)}
15: end-foreach
```

This `foreach`-loop initializes the observation-history of agent i (for $\forall i \in N$) to a list including only the observation in the state corresponding to ω_1, $\mathbf{OBS}(i, \omega_1)$, which is a set of grounded observations and their value holding in the state. $Q_i(\omega_1)$ is the information-set of agent i in ω_1 (even though agent 1 acts in ω_1), and $\mathbf{Q}_{i,0}$ is the information-partition at state-level 0 of agent i. The **ADD_NEXT_LEVEL** method should now have $\{\mathbf{Q}_{i,l-1}\}_{i\in N}$ and $\{Q_i\}_{i\in N}$ among its inputs, and $\{\mathbf{Q}_{i,l}\}_{i\in N}$ and $\{Q_i\}_{i\in N}$ among its outputs. Alg.2 is changed accordingly: first, manipulation of agent 1's information-partition, $\mathbf{P}_1$, in line **#6** and **#20** is removed together with line **#8**. Second, each reference to any $action\_level_m$ is replaced with $action\_level_m(\omega)$ to keep track of the sub-tree of each $\omega \in last\_state\_level$. Line **#11** is deleted, but the following line is added after line **#6** to initialize the observation-history and information-sets $Q_i(\omega')$ of choice node ω' (clone of ω).

```
7: | foreach i ∈ N, OBS_HIST(i, ω') ← OBS_HIST(i, ω), Qi(ω') ← {ω'} end-foreach
```

To update the observation-histories of agents and to initialize their information sets for a new choice node y corresponding to a successor state, the next 5 lines should be added after line **#16** in the original code of Alg.2.

```
20: | | | | | | foreach j ∈ N,
21: | | | | | | | OBS_HIST(j, y) ← ⟨OBS_HIST(j, ω), …
22: | | | | | | |          OBS(j, INCL_PROGR_FACTS(ω, (TRACE(x), ai), PROB, ω0, G)), …
23: | | | | | | |          OBS(j, y)⟩, Qj(y) ← {y}
24: | | | | | | end-foreach
```

In line **#22** the **INCL_PROGR_FACTS** call produces a choice node that corresponds to a state, where the progressive facts about consistent and executable actions of $(\mathbf{TRACE}(x), a_i)$ are added to the state corresponding to ω. These actions are identified in the same way as in the **HAS_CONSISTENT_EXECUTABLE_SUBSET** method. Finally, line **#32** in the original Alg.2 should be replaced with the following lines.

```
36: foreach i ∈ N
37: | if i > 1
38: | | foreach qi ∈ Qi
39: | | | action_levels_qi ← ∪ω∈qi action_level_i(ω)
40: | | | foreach x ∈ action_levels_qi , Pi(x) ← action_levels_qi end-foreach
41: | | | Pi ← Pi ∪ {action_levels_qi}
42: | | end-foreach
43: | end-if
44: | foreach ω, ω' ∈ next_state_level where ω ≠ ω'
45: | | if OBS_HIST(i, ω) == OBS_HIST(i, ω') then
46: | | | if i == 1 then P1(ω) ← P1(ω) ∪ {ω'}, P1(ω') ← P1(ω') ∪ {ω} end-if
47: | | | Qi(ω) ← Qi(ω) ∪ {ω'}, Qi(ω') ← Qi(ω') ∪ {ω}
48: | | end-if
49: | end-foreach
50: | Ri ← ∅
51: | foreach ω ∈ next_state_level
52: | | if i == 1 then P1 ← P1 ∪ {P1(ω)} end-if
53: | | Ri ← Ri ∪ {Qi(ω)}
```

```
54: | end-foreach
55: end-foreach
56: return ⟨G, Ω, {P_i}_{i∈N}, {P_i}_{i∈N}, {R_i}_{i∈N}, {Q_i}_{i∈N}, next_state_level⟩
```

In the `foreach`-loop **(#36-55)**, if $i > 1$ **(#37-43)**, then those i^{th} action-levels are unified into an information-set P_i of agent i, where the root nodes belong to the same information-set. $\mathbf{P}_i$ is updated accordingly. $\mathbf{Q}_i$ in line **#38** is $\mathbf{Q}_{i,*}$ received as an input of the **ADD_NEXT_LEVEL** method. In lines **(#44-49)** choice nodes in the next state-level corresponding to states with same observation-history are put in the same information-set. Finally the information-partitions of all agents are finalized in the next state-level **(#50-54)**, and it is returned by the method **(#55)**.

4. Conclusions

This paper provides algorithms for converting fully- and partially-observable MA-PDDL descriptions to extensive-form games. Partial-observability is introduced as a separate extension to MA-PDDL. Limitations include the discrete (or fixed-length durative), deterministic nature of converted descriptions, and that each agent needs to execute exactly one action at a time (even `no-op`). In the future this should be extended to allow probabilistic and non-deterministic descriptions with durative actions and without the limit on the number of concurrent actions.

Acknowledgments

This work was partially supported by the ARTEMIS JU and the Hungarian National Development Agency (NFÜ) in frame of the R3-COP (Robust & Safe Mobile Co-operative Systems) project.

References

[1] M. de Weerdt, B. Clement, *Introduction to planning in multiagent systems*. *Multiagent Grid Systems.* **5(4)** (2009), 345-355.
[2] D. L. Kovacs, *A Multi-Agent Extension of PDDL3.1*, Proc. of: 3rd Workshop on the Int. Planning Comp. (IPC), ICAPS-2012, (25–29 June 2012, Atibaia, Brazil).
[3] J. von Neumann, O. Morgenstern, “Theory of games and economic behavior”, Princeton, 1944.
[4] D. McDermott et al., *PDDL---The Planning Domain Definition Language*, Tech. Rep., TR-98-003/DCS TR-1165, Yale Center for CVC, NH, CT, (1998).
[5] J. F. Nash, *Non-cooperative Games*. *Annals of Maths*. **54** (1951), 286-295.
[6] M. Helmert, *Changes in PDDL 3.1*, Unpublished summary from the IPC-2008 website, (2008).
[7] D. L. Kovacs, *BNF Definition of PDDL3.1*, Unpublished manuscript from the IPC-2011 website, (2011).
[8] J. C. Harsanyi, *Games with incomplete information played by Bayesian players, Part I-III*. *Manag. Sc*. **14(3,5,7)** (1967-1968), 159-182, 320-334, 486-502.
[9] M. Fox, D. Long, *Modelling Mixed Discrete-Continuous Domains for Planning*. *Journal of Artificial Intelligence Research*. **27** (2006), 235-297.