Converting MA-PDDL to Extensive-Form Games

Dániel L. Kovács, Tadeusz P. Dobrowiecki

Department of Measurement and Information Systems, Budapest University of Technology and Economics
Magyar tudósok krt. 2, H-1117, Budapest, Hungary
E-mail: dkovacs@mit.bme.hu, tade@mit.bme.hu

Abstract: This paper presents algorithms for converting multi-agent planning (MAP) problems described in Multi-Agent Planning Domain Definition Language (MA-PDDL) to extensive-form games in order to analyse and solve them with game-theoretic tools in general. MA-PDDL is an attempt to standardize the description of MAP problems similarly to PDDL in the single-agent setting. In this paper MA-PDDL is extended with partial-observability and probabilistic-effects to model more realistic domains. The conversion is fruitful in both ways: 1) extensive-form games can be solved via game-theoretic solution concepts (e.g. Nash-equilibrium) providing solutions to corresponding MAP problems in general, and 2) MA-PDDL problems can be solved via MAP methods providing solutions to corresponding games. Both cooperative and non-cooperative solutions can be achieved.

Keywords: multi-agent; planning; pddl; game theory; partial observability; probabilistic

1 Introduction

This paper presents methods for converting multi-agent planning (MAP) problems [1] described in MA-PDDL (Multi-Agent Planning Domain Definition Language) [2] to extensive-form games [3][4], in order to enable the application of game-theoretic principles (e.g. solution concepts) to MAP problems in general.

PDDL [5] is quasi the standard description language for modeling deterministic, single-agent planning problems. Such problems form the basis of automated planning [6], which is of central importance in Artificial Intelligence (AI) [7] due to it provides practical methods for designing goal- and utility-based intelligent agents, with real-world applications ranging from game playing to control of space vehicles. However PDDL is limited to only one planner, whereas real-world planning problems may involve multiple cooperative or adversary, controllable or non-controllable planner agents with different goals, different capabilities and interacting actions (competing corporations, multiplayer games, electronic auctions, assisted living, computer networks, robotic soccer, etc.). To model these aspects PDDL was recently extended to the multi-agent setting in [2].

However solving a MAP problem can prove to be quite difficult due to its inherent complexity. It is well known, that single-agent planning in discrete-time is PSPACE-complete even in the propositional case [8] (where conditions within actions are just literals without variables), i.e. propositional single-agent planning is among the hardest problems in PSPACE, and PSPACE contains NP. Now in case of multiple agents the number of actions – and thus complexity – increases exponentially (since all action-combinations of the multiple agents need to be considered in general), not speaking of richer MAP models (including predicates, numeric variables, plan metrics, continuous-time, uncertainty, partial-observability, etc.). This makes MAP intractable for realistic domains in general, and only approximations of the global optima are possible in practice. Thus it comes to no surprise, that currently – in lack of a (quasi)standard MAP problem modeling language – there are no general means for solving MAP problems.

This paper tries to overcome the above issue by proposing a translation of MAP problems to extensive-form games in order to analyze and solve them via game-theoretic methods. MAP problems are described in MA-PDDL, which is an attempt to standardize the description of MAP problems (similarly to PDDL in single-agent planning). Naturally the translation cannot reduce the complexity of converted MAP problems (i.e. solution approximation or shrinking of games [9] may be required for tractability), but at least it opens a way to strategically analyze and solve MAP problems in general. To our knowledge this is the first result in automatically converting MAP problems to game-theoretic models.

Game theory [3][10] describes essentially the same multi-agent situations as MAP (i.e. strategic interaction of agents), thus the conversion of MAP problems to games is relatively straightforward, but game theory also provides a rich repertoire of useful solution concepts that can be applied to MAP problems after the conversion. The solution of a game is usually a set of strategy-combinations, which corresponds to a set of joint-plans in the MAP problem. These solutions may be cooperative or non-cooperative depending on the solution concept used. Extensive-form games are appropriate for both cases even though they are part of non-cooperative game theory. In the non-cooperative case e.g. Nash-equilibrium (NE) [11] or Subgame Perfect NE [12] or Perfect Bayesian Equilibrium [13], while in the cooperative case e.g. Pareto-optimum [10] can be used to find suitable solutions. Cooperation can also be achieved by maximizing social-welfare (i.e. the sum of utilities of agents) or individual utility of agents may reflect their collective preferences, so even non-cooperative solution concepts can lead to cooperative solutions. I.e. the proposed conversion does not limit the cooperation of agents.

Eventually 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 concepts providing solutions to the corresponding MAP problem and **(2)** MA-PDDL can provide a much richer model of the same game-theoretic situation, and can be solved via available state-of-the-art MAP methods [1] providing solutions to the corresponding game (converted from MA-PDDL).

The paper is structured as follows: Section 2 introduces the preliminaries of automated planning, PDDL, MA-PDDL and extensive-form games. Section 3 proposes conversion algorithms from MA-PDDL to extensive-form games. First the fully-observable, deterministic case is discussed, then partial-observability and probabilistic-effects are added gradually. At the end of Section 3 a short example illustrates the concept. Finally Section 4 concludes the work and outlines future research directions. Appendix 1-3 provides the additional BNF (Backus-Naur Form) grammar for partial-observability and probabilistic-effects in MA-PDDL.

2 Preliminaries

2.1 Automated Planning

Automated planning [6] is a process of finding a plan of action (e.g. either a totally ordered sequence of actions, or some conditional plan of action) that upon execution is expected to solve a planning problem. A *planning problem* typically defines an *initial state* and desired *goal states* of an environment, i.e. the solution of a planning problem, a *solution plan* should drive the environment from the initial state to a goal state upon execution (hopefully in a minimal number of steps, minimizing the risk and the cost of execution). In case of a deterministic environment with only a single agent the execution of a solution plan should lead to a goal state, however in case the environment is only partially observable to the agent, or it is probabilistic/non-deterministic, or there are multiple autonomous agents in it, then the execution may fail (e.g. an other agent may interfere during execution). So in this case a solution plan should be either prepared for all contingencies or its execution should be monitored and the plan should be repaired on-the-fly. A planning problem may have many or no solutions at all.

2.2 PDDL (Planning Domain Definition Language)

PDDL [5] is the quasi-standard, predicate logic based declarative description language for deterministic, single-agent planning problems. The latest official version of PDDL is 3.1 [14,15]. Each new version of the language adds new, modular features to previous versions. PDDL divides the description of the planning problem in two parts: a *domain- and a problem-description*. The domain-description contains those model-elements which are present in every particular problem of the domain, while the problem-description 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, while its *output* is a plan that solves the specified planning problem (if it is solvable).

More precisely, the domain-description contains the following: a unique *name*; a list of *requirements* (a list of PDDL-features used); a *type-hierarchy* (classifying *objects*); *constants* (objects present in every problem of the domain); and 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 if 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. Actions may refer to derived predicates in their preconditions. Constraints are statements in modal logic about state-trajectories that must be true for valid *solution plans*.

The problem-description also has a unique *name*; a *reference* to the respective domain-description; a list of all *objects* in the logical universe; an *initial state*; and a specification of *goal states* of the environment. Problem-descriptions can also include a *metric* (a real-valued function for measuring the quality of solution plans); *timed initial literals* (facts becoming true at a given time); and *constraints* similarly to the domain-description, but here 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 goal, or in action preconditions.

2.3 MA-PDDL (Multi-Agent PDDL)

MA-PDDL [2] is a minimalistic, modular extension of PDDL3.1, indicated by a new additional PDDL-requirement, `:multi-agent`. It extends PDDL3.1 to allow *planning by and for multiple agents*. *Different agents* may have *different actions*, *different goals* and *different metrics*, unlike in original PDDL. This allows modeling of not just homogeneous, but also heterogeneous agents in either cooperative or competitive scenarios. Moreover, in MA-PDDL the preconditions of actions can directly refer to concurrent actions and thus *actions with interacting effects* can be modeled in general (e.g. when at least 2 agents are needed to execute the lift action to lift a heavy table, or it will remain on the ground, or a third agent may interfere by pushing the table down to the ground), which allows for a more refined model of cooperation and inter-dependence of agents. However, since PDDL3.1 assumes that the environment is deterministic and fully-observable (i.e. every agent can access the value of every state-fluent at every instant and observe every previously executed action), thus by default the same holds in MA-PDDL too. Nonetheless in Section 3.2 and 3.3 these constraints are lifted by extending MA-PDDL with partial-observability and probabilistic effects.

2.4 Game Theoretic Fundamentals

The *normal form* of an *incomplete information game* Γ [16] (the most general non-cooperative game) is a 5-tuple $\Gamma = (N, \{S_i\}_{i\in N}, \{u_i\}_{i\in N}, \{T_i\}_{i\in N}, p)$, where $N = \{1,2,\dots,n\}$ denotes the set of *agents*; S_i is the finite set of *pure strategies* of $i \in N$ and T_i is the finite set of its *types* of $i \in N$; typically $u_i: S \times T_i \to \mathbb{R}$ is the real-valued *utility function* of agent i, where $S = \times_{i=1}^{n} S_i$ is the set of all *strategy-combinations*. Depending on the interpretation and dependence of types, sometimes the utility of an agent may also depend on the type of other agents too, i.e. $u_i: S \times T \to \mathbb{R}$, where $T = \times_{i=1}^{n} T_i$ denotes the set of all *type-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 $t \in T$ 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... 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, where *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 the 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 believes to be possible in ω. It is assumed that $\omega \in P_i(\omega)$ holds for every $\omega \in \Omega$ and $i \in N$, and also 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 G is the set of all $s_i: \mathbf{P}_i \to A_i$ functions, where for $\forall P_i(\omega) \in \mathbf{P}_i$ $s_i\big(P_i(\omega)\big) \in h(\omega)$ holds. This finishes the description of the extensive-form of an incomplete information game.

3 Conversion of MA-PDDL to Extensive Form Games

This section presents the main results of the paper: the conversion of fully- and partially-observable, probabilistic MA-PDDL models to extensive-form games.

3.1 Case of Full-Observability

The idea of the conversion 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` (no-operation) 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. Thus all joint-plans with agents acting effectively $\leq k$ times (maybe even heterogeneously) are found. **Alg. 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($PROB$)
  $n = |N|,\ \kappa = 1 + k \cdot n$
  foreach $i \in N$
  |  $T_i \leftarrow \{t_i \leftarrow$ NEW_TYPE()$\},\ \mathbf{P}_i \leftarrow \emptyset$
  |  $A_i \leftarrow$ ALL_GROUNDED_ACTIONS($PROB, i$) $\cup$ {no-op}
  end-foreach
$\omega_0 \leftarrow$ NEW_CHOICE_NODE(), $g(\omega_0) \leftarrow 0,\ A_0 \leftarrow \{t = (t_1, t_2, \dots, t_n)\},\ h(\omega_0) \leftarrow A_0$
$p\big(t = (t_1, t_2, \dots, t_n)\big) \leftarrow 1,\ s_0(\omega_0) \leftarrow p$
$\omega_1 \leftarrow$ CHOICE_NODE_FROM_INITIAL_STATE($PROB$)
$g(\omega_1) \leftarrow 1,\ h(\omega_1) \leftarrow A_1,\ P_1(\omega_1) \leftarrow \{\omega_1\},\ \mathbf{P}_1 \leftarrow \mathbf{P}_1 \cup \{P_1(\omega_1)\}$
$\Omega \leftarrow \{\omega_0, \omega_1\},\ G \leftarrow (\Omega, \{(\omega_0, t, \omega_1)\}),\ state\_level_0 \leftarrow \{\omega_1\}$
while ($l < k$)
|  $l \leftarrow l + 1$
|  $\langle G, \Omega, \{\mathbf{P}_i\}_{i \in N}, \{P_i\}_{i \in N}, state\_level_l \rangle \leftarrow$
|        ADD_NEXT_LEVEL($PROB, G, \omega_0, \Omega, \{\mathbf{P}_i\}_{i \in N}, \{P_i\}_{i \in N}, state\_level_{l-1}, \{A_i\}_{i \in N}, n$)
end-while
$\{S_i\}_{i \in N} \leftarrow$ ENUMERATE_STRATEGIES($N, \{\mathbf{P}_i\}_{i \in N}, h, \{A_i\}_{i \in N}$)
```

```
20: $\{u_i\}_{i\in N} \leftarrow$ GET_METRIC_VALUES$(PROB, N, \{S_i\}_{i\in N}, \{T_i\}_{i\in N}, p, \omega_0, s_0, G, state\_level_k)$
21: return $\Gamma = (N, \{S_i\}_{i\in N}, \{u_i\}_{i\in N}, \{T_i\}_{i\in N}, p, \Omega, \omega_0, s_0, \{\mathbf{P}_i\}_{i\in N}, \{P_i\}_{i\in N}, \{A_i\}_{i\in N\cup\{0\}}, g, h, \kappa, G)$
```

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 (deterministic MA-PDDL 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 with no 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, and A_0 is allowed in ω_0. Then **(#10)** the probability of "action" t, $p(t)$, is set to 1, so this distribution governs 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. 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 $\{\omega_1\}$.

State-levels are of 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. The detailed pseudo-code of the method is shown in Alg. 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 $s_i: \mathbf{P}_i \rightarrow A_i$ 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.

The utility of agents is defined explicitly for every possible outcome (i.e. for every strategy-combination). These outcomes are represented with choice-nodes of the last state-level in the game-tree. 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 trajectories from $PROB$ for every agent-object, and associate them to the respective choice-nodes. If an agent-object has no metric defined in $PROB$, then its utility is 1, if its goal was achieved during the given trajectory, and 0 otherwise. 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 the algorithm returns the converted game, 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 described in **Alg. 2** below.

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

```
1:  ADD_NEXT_LEVEL($PROB, G = (V, E), \omega_0, \Omega, \{\mathbf{P}_i\}_{i\in N}, \{P_i\}_{i\in N}, last\_state\_level, \{A_i\}_{i\in N}, n$)
2:    $next\_state\_level \leftarrow \emptyset$
3:    foreach $\omega \in last\_state\_level$
4:    | $action\_level_1 \leftarrow \{\omega\}$,  $action\_level_{n+1} \leftarrow \emptyset$
5:    | TRACE($\omega$) $\leftarrow \varepsilon$,  $\omega' \leftarrow$ CLONE($\omega$),  $\Omega \leftarrow \Omega \cup \{\omega'\}$
6:    | $g(\omega') \leftarrow 1$,  $h(\omega') \leftarrow A_1$,  $P_1(\omega') \leftarrow \{\omega'\}$,  $\mathbf{P}_1 \leftarrow \mathbf{P}_1 \cup \{P_1(\omega')\}$
7:    | for $i = 1$,  $i \le n$,  $i{+}{+}$
8:    | | if $i > 1$ then $\mathbf{P}_i \leftarrow \mathbf{P}_i \cup \{action\_level_i\}$ end-if
9:    | | if $i < n$ then $action\_level_{i+1} \leftarrow \emptyset$ end-if
10:   | | foreach $x \in action\_level_i$
11:   | | | if $i > 1$ then $P_i(x) \leftarrow action\_level_i$ end-if
12:   | | | foreach $a_i \in h(x)$
13:   | | | | if $i < n$ then
14:   | | | | | $y \leftarrow$ NEW_CHOICE_NODE()
15:   | | | | | TRACE($y$) $\leftarrow$ (TRACE($x$), $a_i$)
16:   | | | | | $g(y) \leftarrow i + 1$,  $h(y) \leftarrow A_{i+1}$
17:   | | | | else
18:   | | | | | if HAS_CONSISTENT_EXECUTABLE_SUBSET((TRACE($x$), $a_i$), $PROB, \omega_0, \omega, G$) then
19:   | | | | | | $y \leftarrow$ CHOICE_NODE_FROM_SUCCESSOR_STATE($\omega$, (TRACE($x$), $a_i$), $PROB, \omega_0, G$)
20:   | | | | | | $g(y) \leftarrow 1$,  $h(y) \leftarrow A_1$,  $P_1(y) \leftarrow \{y\}$,  $\mathbf{P}_1 \leftarrow \mathbf{P}_1 \cup \{P_1(y)\}$
21:   | | | | | else
22:   | | | | | | $y \leftarrow \omega'$
23:   | | | | | end-if
24:   | | | | end-if
25:   | | | | $action\_level_{i+1} \leftarrow action\_level_{i+1} \cup \{y\}$
26:   | | | | $V \leftarrow V \cup \{y\}$,  $E \leftarrow E \cup \{(x, a_i, y)\}$,  $\Omega \leftarrow \Omega \cup \{y\}$
27:   | | | end-foreach
28:   | | end-foreach
29:   | end-for
30:   | $next\_state\_level \leftarrow next\_state\_level \cup action\_level_{n+1}$
31:   end-foreach
32:   return $\langle G = (V, E), \Omega, \{\mathbf{P}_i\}_{i\in N}, \{P_i\}_{i\in N}, next\_state\_level \rangle$
```

The **ADD_NEXT_LEVEL** method has 9 inputs **(#1)**: $PROB$ is the 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 actual information partitions and functions of agents respectively; $last\_state\_level$ is the latest state-level; $\{A_i\}_{i\in N}$ is the set of sets of 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, every move of every agent, starting with agent 1 until agent n **(#7-29)**.

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. Inside the `for`-loop a `foreach`-loop **(#10-28)** goes through every choice node x of action-level i, and inside it a further `foreach`-loop **(#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 $\texttt{TRACE}(x)$. $\texttt{TRACE}(\omega)$ 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 $(\texttt{TRACE}(x), a_i)$ 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: first **(i)** it collects those actions from $(\texttt{TRACE}(x), a_i)$ into a set C, which are potentially executable in ω. A single-action is considered *potentially executable* in a state if its pre-conditions are satisfied taking also **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 will have 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. I.e. potential executability of actions is independent from concurrently executed actions in light of Assumption 1.

Next **(ii)** all single- and joint-actions are identified within C. A *joint-action* within C 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. Reference to actions may be positive or negative, and the conditional effects are *active* in ω if their conditions are satisfied in ω with actions in C being executed. This produces an unambiguous partition of C. Next **(iii)** individually inconsistent or not executable elements are removed from that partition. A *single-action is individually consistent* in ω if its active conditions and effects are both consistent on their own in ω. A *joint-action is individually consistent* in ω if its joint active conditions and joint active effects are both consistent on their own in ω in case the actions within the given 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 in ω.

Next **(iv)** elements with *pairwise inconsistent joint-effects* are removed from the partition. Finally **(v)** the remaining elements are checked, whether their *joint execution* is allowed by the hard state-trajectory constraints in $PROB$. **If yes**, then these actions form the *consistent and executable subset* of $(\texttt{TRACE}(x), a_i)$, i.e.

they can be executed, and so the **HAS_CONSISTENT_EXECUTABLE_SUBSET** method returns `true`. Otherwise, **if not**, or if the executable 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 added to action-level $n+1$ **(#19,20,25)**. Otherwise a choice node $y=\omega'=\omega$ is put into action-level $n+1$ **(#5,22,25)**. In both cases G and Ω are 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 finishing.

3.2 Case of Partial-Observability

Now MA-PDDL is extended with partial-observability (cf. Appendix 1 and 3). In case of partial-observability information sets at state-levels of the converted game may not be singleton, since there may be state/action-trajectories, where the observation-history (including the observation of actions) is the same for an agent, and thus the choice-nodes corresponding to those trajectories should be members of the same information set. Based on this *Alg.1 needs to be extended* as follows. First the following 3 lines should be inserted between line **#12** and **#13** in Alg. 1.

```
13: foreach $i \in N$,
14: |  OBS_HIST$(i,\omega_1) \leftarrow \langle$OBS$(i,\omega_1)\rangle$,  $Q_i(\omega_1) \leftarrow \{\omega_1\}$,  $\mathbf{Q}_{i,0} \leftarrow \{Q_i(\omega_1)\}$
15: end-foreach
```

This `foreach`-loop initializes the observation-history of agent i (for $\forall i \in N$) to a list including only its observations in ω_1, OBS(i,ω_1), which is a set of grounded observations and their value holding in ω_1. $Q_i(\omega_1)$ 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 now has also $\{\mathbf{Q}_{i,l-1}\}_{i\in N}$ 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** are removed together with the complete line **#8**. Then each reference to any $action\_level_m$ is replaced with $action\_level_m(\omega)$ to keep track of the possibly distinct sub-trees 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 $i \in N$, OBS_HIST$(i,\omega') \leftarrow$ OBS_HIST$(i,\omega)$, $Q_i(\omega') \leftarrow \{\omega'\}$ 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 **#20** in the original pseudo-code of Alg.2.

```
20: | | | | | | foreach $j \in N$,
21: | | | | | | |  OBS_HIST$(j,y) \leftarrow \langle$OBS_HIST$(j,\omega)$, …
22: | | | | | | |            OBS$\big(j,$INCL_PROGR_FACTS$(\omega,($TRACE$(x),a_i),PROB,\omega_0,G)\big)$,  …
23: | | | | | | |            OBS$(j,y)\rangle$,  $Q_j(y) \leftarrow \{y\}$
24: | | | | | | end-foreach
```

In line **#22** `INCL_PROGR_FACTS` produces a choice node corresponding to a state, where *progressive facts* about consistent and executable actions of $(\texttt{TRACE}(x), a_i)$ are added to ω. They are identified as in **`HAS_CONSISTENT_EXECUTABLE_SUBSET`**. Finally line **#32** in Alg.2 is replaced with the following code-segment.

```
36: foreach $i \in N$
37: | if $i > 1$
38: | | foreach $\mathbf{q}_i \in \mathbf{Q}_i$
39: | | | $action\_levels_{\mathbf{q}_i} \leftarrow \bigcup_{\omega \in \mathbf{q}_i} action\_level_i(\omega)$
40: | | | foreach $x \in action\_levels_{\mathbf{q}_i}$ , $P_i(x) \leftarrow action\_levels_{\mathbf{q}_i}$ end-foreach
41: | | | $\mathbf{P}_i \leftarrow \mathbf{P}_i \cup \{action\_levels_{\mathbf{q}_i}\}$
42: | | end-foreach
43: | end-if
44: | foreach $\omega, \omega' \in next\_state\_level$ where $\omega \neq \omega'$
45: | | if OBS_HIST$(i,\omega)$ == OBS_HIST$(i,\omega')$ then
46: | | | if $i == 1$ then $P_1(\omega) \leftarrow P_1(\omega) \cup \{\omega'\}$, $P_1(\omega') \leftarrow P_1(\omega') \cup \{\omega\}$ end-if
47: | | | $Q_i(\omega) \leftarrow Q_i(\omega) \cup \{\omega'\}$, $Q_i(\omega') \leftarrow Q_i(\omega') \cup \{\omega\}$
48: | | end-if
49: | end-foreach
50: | $\mathbf{R}_i \leftarrow \emptyset$
51: | foreach $\omega \in next\_state\_level$
52: | | if $i == 1$ then $\mathbf{P}_1 \leftarrow \mathbf{P}_1 \cup \{P_1(\omega)\}$ end-if
53: | | $\mathbf{R}_i \leftarrow \mathbf{R}_i \cup \{Q_i(\omega)\}$
54: | end-foreach
55: end-foreach
56: return $\langle G = (V,E), \Omega, \{\mathbf{P}_i\}_{i \in N}, \{P_i\}_{i \in N}, \{\mathbf{R}_i\}_{i \in N}, \{Q_i\}_{i \in N}, next\_state\_level \rangle$
```

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 **`ADD_NEXT_LEVEL`**. 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 **(#56)**.

3.3 Case of Probabilistic Effects

In this section probabilistic elements are assumed to be added to MA-PDDL on top of partial-observability (cf. Appendix 2-3), so the **`CONVERT`** method in Section 3.2 needs to be modified accordingly. In row **(#4)** the calculation of κ should be: $\kappa = 1 + k \cdot (n+1)$, since now a dedicated chance-node level is added after each action-tree to represent all the possible probabilistic outcomes. Next, the initialization of types-sets should be replaced with initialization of information-partitions $\mathbf{Q}_{i,0} \leftarrow \emptyset$ in row **(#6)**. Then in row **(#9)** the initialization of the action-set of agent 0 should be replaced with: $(T,p) \leftarrow \textbf{P\_INITSTATES}(PROB)$, $A_0 \leftarrow T$. Here **`P_INITSTATES`** generates all the possible grounded initial states of the MA-PDDL problem $PROB$, corresponding to the finite set of type-combinations and their respective probabilities forming the a priori probability distribution p over T. Thus in row **(#10)** the initialization of p is omitted, and row **(#11)** is replaced with:

```
11: $(\Omega_1, inv) \leftarrow$ CHOICE_NODES_FROM_P_INITSTATES$(T)$, $\Omega \leftarrow \{\omega_0\} \cup \Omega_1$, $state\_level_0 \leftarrow \Omega_1$, $E \leftarrow \emptyset$
```

Here the set of choice nodes, Ω_1, is formed from T. An inverse function, $inv: \Omega_1 \rightarrow T$ is also returned for later use. The set of all nodes, edges and the initial state-level is initialized. Next, row **(#12)** is deleted, and the next 4 rows before the `while`-cycle are replaced with the following code-segment.

```
12: foreach $\omega_1 \in state\_level_0$
13:   $g(\omega_1) \leftarrow 1$, $h(\omega_1) \leftarrow A_1$, $P_1(\omega_1) \leftarrow \{\omega_1\}$, $E \leftarrow E \cup \{(\omega_0, t, \omega_1)\}|_{inv(\omega_1)=t\in T}$
14:   foreach $i \in N$,
15:     OBS_HIST$(i, \omega_1) \leftarrow \langle$OBS$(i, \omega_1)\rangle$, $Q_i(\omega_1) \leftarrow \{\omega_1\}$
16:   end-foreach
17: end-foreach
18: $G \leftarrow (\Omega, E)$
19: foreach $i \in N$,
20:   foreach $\omega, \omega' \in state\_level_0$ where $\omega \neq \omega'$
21:     if OBS_HIST$(i, \omega)$ == OBS_HIST$(i, \omega')$ then
22:       if $i == 1$ then $P_1(\omega) \leftarrow P_1(\omega) \cup \{\omega'\}$, $P_1(\omega') \leftarrow P_1(\omega') \cup \{\omega\}$ end-if
23:       $Q_i(\omega) \leftarrow Q_i(\omega) \cup \{\omega'\}$, $Q_i(\omega') \leftarrow Q_i(\omega') \cup \{\omega\}$
24:     end-if
25:   end-foreach
26:   foreach $\omega \in state\_level_0$
27:     if $i == 1$ then $\mathbf{P}_1 \leftarrow \mathbf{P}_1 \cup \{P_1(\omega)\}$ end-if
28:     $\mathbf{Q}_{i,0} \leftarrow \mathbf{Q}_{i,0} \cup \{Q_i(\omega)\}$
29:   end-foreach
30: end-foreach
```

This initializes information-sets, partitions and observation-histories of all agents at the initial state-level. Next, **`ADD_NEXT_LEVEL`** should include s_0 in its inputs and outputs; the set of type-sets, $\{T_i\}_{i\in N}$, should be replaced with T both in Γ returned by **`CONVERT`** and in the inputs of **`GET_METRIC_VALUES`**. Utilities returned by this method should be of form $u_i: S \times T \rightarrow \mathbb{R}$, and the contents of **`ADD_NEXT_LEVEL`** in Section 3.2 between rows **(#12-30)** should be replaced with:

```
12:         if $i < n$ then
13:           $y \leftarrow$ NEW_CHOICE_NODE(), TRACE$(y) \leftarrow ($TRACE$(x), a_i)$
14:           $g(y) \leftarrow i + 1$, $h(y) \leftarrow A_{i+1}$, $E \leftarrow E \cup \{(x, a_i, y)\}$, $Y \leftarrow \{y\}$
15:         else
16:           $c \leftarrow$ NEW_CHOICE_NODE(), $g(c) \leftarrow 0$, $h(c) \leftarrow A_0$
17:           $\Omega \leftarrow \Omega \cup \{c\}$, $V \leftarrow V \cup \{c\}$, $E \leftarrow E \cup \{(x, a_i, c)\}$
18:           if HAS_CONSISTENT_EXECUTABLE_SUBSET$(($TRACE$(x), a_i), PROB, \omega_0, \omega, G)$ then
19:             $(Y, p_Y) \leftarrow$ CHOICE_NODES_FROM_P_SUCC_STATES$(\omega, ($TRACE$(x), a_i), PROB, \omega_0, G)$
20:             foreach $y \in Y$,
21:               $g(y) \leftarrow 1$, $h(y) \leftarrow A_1$, $P_1(y) \leftarrow \{y\}$, $E \leftarrow E \cup \{(c, y, y)\}$
22:               foreach $j \in N$,
23:                 OBS_HIST$(j, y) \leftarrow \langle$OBS_HIST$(j, \omega)$, …
24:                              OBS$(j,$INCL_PROGR_FACTS$(\omega, ($TRACE$(x), a_i), PROB, \omega_0, G))$, …
25:                              OBS$(j, y)\rangle$, $Q_j(y) \leftarrow \{y\}$
26:               end-foreach
27:             end-foreach
28:           else
29:             $Y \leftarrow \{\omega'\}$, $p_Y(\omega') \leftarrow 1$, $E \leftarrow E \cup \{(c, \omega', \omega')\}$
30:           end-if
31:           $s_0(c) \leftarrow p_Y$
32:         end-if
33:         $action\_level_{i+1}(\omega) \leftarrow action\_level_{i+1}(\omega) \cup Y$, $V \leftarrow V \cup Y$, $\Omega \leftarrow \Omega \cup Y$
```

This includes management of chance nodes c, added after each n-long action-combination $($**`TRACE`**$(x), a_i)$ to represent possible probabilistic outcomes. The set of choice nodes in the next state level corresponding to these outcomes, Y, connected to c, is produced by **CHOICE_NODES_FROM_P_SUCC_STATES**. It generates all possible successor states via all possible consistent and executable action-subsets of $(\textbf{TRACE}(x), a_i)$ as described in the end of Section 3.1, except that the consistency of (joint)actions depends on every variation of their effects.

The **CONVERT** method in Sections 3.1-3.3 is of *constant-time complexity* in the number of possible state/action-trajectories within a finite horizon k, however the number of these trajectories is *super-exponential* in the number of agents., i.e. optimization is needed. The following 4 steps reduce the redundancy of G.

1. From top to bottom each edge of potentially not executable and `no-op` actions should be deleted from G together with their complete sub-graph.
2. If a chance node c has only 1 outgoing edge, then it should be deleted, and its parent action node a should be connected directly to its child state node ω.
3. Starting from the last state-level the cloned ω' nodes should be deleted together with the edges from their parents.
4. If the last remaining outgoing edge of a node is being deleted, then the node should be replaced with the end-vertex of that edge, if it exists. Otherwise, if it is an action-node, it should be deleted with all edges from its parents.

3.4 Example

A *one-card poker problem* is formulated in MA-PDDL based on pp. 37-40 in [10]. It is then converted to an extensive-form game, which is solved, and the solution is projected back to the MA-PDDL level. The example is best read and understood in conjunction with the original definition of MA-PDDL in [2] and Appendix 1-3.

```
(define  (domain one-card-poker)
(:requirements :strips :equality :negative-preconditions :typing :numeric-fluents
       :conditional-effects :partial-observability :probabilistic-effects :multi-agent)
(:types player) (:constants player1 player2 - player)
(:predicates (next-role fo ?p - player) (winning-hand-player1 (fo player1)))
(:functions (income-of fo ?p - player) (investment-of fo ?p - player) (money-in-pot fo))

(:action raise :agent player1 :parameters ()
:precondition (next-role player1)
:effect       (and (not (next-role player1))
                   (next-role player2)
                   (increase (investment-of player1) 1)
                   (increase (money-in-pot) 1)))

(:action fold :agent player1 :parameters ()
:precondition (next-role player1)
:effect       (and (not (next-role player1))
                   (when (not (winning-hand-player1))
                         (increase (income-of player2) (money-in-pot)))
                   (when (winning-hand-player1)
                         (increase (income-of player1) (money-in-pot)))))

(:action meet :agent player2 :parameters ()
:precondition (next-role player2)
:effect       (and (not (next-role player2))
                   (increase (investment-of player2) 1)
                   (when (not (winning-hand-player1))
                         (increase (income-of player2) (+ (money-in-pot) 1)))
                   (when (winning-hand-player1)
                         (increase (income-of player1) (+ (money-in-pot) 1)))))

(:action pass :agent player2 :parameters ()
:precondition (next-role player2)
:effect       (and (not (next-role player2))
                   (increase (income-of player1) (money-in-pot)))))

(define  (problem one-card-poker-prob1)
(:domain one-card-poker)
(:init   (next-role player1)
         (= (money-in-pot) 2)
         (= (income-of player1) 0)
         (= (income-of player2) 0)
         (= (investment-of player1) 1)
         (= (investment-of player2) 1)
         (probabilistic 0.5 (winning-hand-player1)))
(:goal   :condition ())
(:metric :agent  ?p - player
         :utility maximize (- (income-of ?p) (investment-of ?p))))
```

There are 2 agents, `player1` and `player2`, playing a simple poker game. Before the play, both put 1 coin in the pot. Then they both receive their cards. `player1` receives a winning hand with a probability of ½, and then it either *raise* the bid (to 2 coins) or *fold*. If it folds (shows its cards), then if it has a winning hand, then it wins the pot. Otherwise `player2` wins. But if `player1` raises, then `player2` can either *meet* this bid (also put 1 more coin in the pot) or *pass*. If it decides to pass, then `player1` wins the pot. But if `player2` meets, then if `player1` has a winning hand, then `player1` wins the pot. Otherwise `player2` wins the pot.

Both agents try to maximize their profit (i.e. the difference of their income and investment), but `player2` is in a worse position, since it is unable to observe the hand of `player1`. So `player2` decides between meeting or passing by chance.

To solve this problem it is first converted to an extensive-form game. Since Poker is non-cooperative game, a non-cooperative solution concept can be used, e.g.

Perfect Bayesian Equilibrium [13], since the game is of incomplete information. The horizon of the conversion is trivially $k = 2$. There are 2 possible initial states depending on whether `winning-hand-player1` is true with probability ½. A chance node, ω_0, is created with 2 edges to $state\_level_0$. Every action-combination is considered for $k = 2$ steps, and since both decentralized agents have 3 actions (including `no-op`), there are 9 combinations, so a resulting game-graph G with 423 nodes emerges, which can be reduced to 11 nodes by using the optimization steps at the end of Section 3.3 (see. Fig. 1). Finding the PBE of this game leads to a unique mixed NE, according to which `player1` should *raise* if it has a winning hand, otherwise it should *fold* with probability $^2/_3$ or *raise* (bluff) with probability $^1/_3$, while `player2` should *meet* with probability $^2/_3$ or *pass* with probability $^1/_3$. This is a rational joint-solution of the above non-cooperative MAP problem.

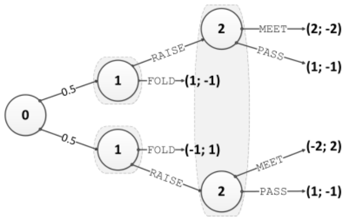

Figure 1
Extensive-form of the one-card poker problem (as shown also in Fig. 2.2 on p. 40 in [10])

The above (reduced) game may seem small, but this is only due to the simplicity of the example. More complex MAP problems can induce much larger games, which may need reduction or solution approximation [9]. Furthermore, the example given was non-cooperative, but MA-PDDL can also describe inherently cooperative situations, e.g. cf. the example in [2], where the only solution is for the self-interested agents to cooperate to achieve their common goal. Thus beyond using cooperative solution concepts, cooperation can be achieved even that way.

Conclusions

Algorithms for converting fully- and partially-observable probabilistic MA-PDDL descriptions to extensive-form games were proposed in order to solve multi-agent planning problems in general. Partial-observability and probabilistic effects were introduced as separate, additional extensions to MA-PDDL. Depending on the multi-agent planning problem at hand and the solution concepts used to solve the resulting game, both cooperative and non-cooperative behavior can be achieved. Limitations include the discrete nature of converted descriptions, and that each agent needs to execute exactly one action at a time. In the future this could be extended to durative cases without the limit on the number of concurrent actions.

Acknowledgement

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):345-355, 2009

[2] D. L. Kovacs: A Multi-Agent Extension of PDDL3.1, Proceedings of the 3rd Workshop on the International Planning Competition (IPC), ICAPS-2012, Atibaia, Brazil, 25-29 June 2012, pp. 19-27

[3] J. von Neumann, O. Morgenstern: Theory of Games and Economic Behavior, Princeton, 1944

[4] H. W. Kuhn: Extensive Games and the Problem of Information, in: H. W. Kuhn, A. W. Tucker, eds., Contributions to the Theory of Games, Vol. 2, Princeton University Press, Princeton, 1953, pp. 193-216

[5] 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

[6] M. Ghallab, D. S. Nau, and P. Traverso: Automated Planning: Theory and Practice, Morgan Kaufmann, 2004

[7] S. J. Russell, P. Norvig: Artificial Intelligence: A Modern Approach (3rd edition) Prentice Hall, 2010

[8] T. Bylander: Complexity Results for Planning, Proc. of 12th International Joint Conference on Artificial Intelligence (IJCAI 91), Sydney, New South Wales, Australia, 24-30 August 1991, pp. 274-279

[9] A. Gilpin: Algorithms for Abstracting and Solving Imperfect Information Games, Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, 2009

[10] R. B. Myerson: Game Theory: Analysis of Conflict, Harvard Univ., 1997

[11] J. F. Nash: Non-Cooperative Games, Annals of Maths, 54:286-295, 1951

[12] R. Selten: Reexamination of the Perfectness Concept for Equilibrium Points in Extensive Games, Int. Journal of Game Theory, 4(1):25-55, 1975

[13] D. Fudenberg, J. Tirole: Perfect Bayesian Equilibrium and Sequential Equilibrium, Journal of Economic Theory, 53:236-260, 1991

[14] M. Helmert: Changes in PDDL 3.1, Unpublished Summary from the IPC-2008 Website, 2008

[15] D. L. Kovacs: BNF Definition of PDDL3.1, Unpublished Manuscript from the IPC-2011 Website, 2011

[16] J. C. Harsanyi: Games with Incomplete Information Played by Bayesian Players, Part I-III., Management Science, 14(3):159-182, 14(5):320-334, 14(7):486-502, 1967-1968

[17] M. Fox, D. Long: Modelling Mixed Discrete-Continuous Domains for Planning, Journal of Artificial Intelligence Research, 27:235-297, 2006

[18] F. Müller, S. Biundo: HTN-Style Planning in Relational POMDPs Using First-Order FSCs, Proc. of 34th Annual German Conference on Artificial Intelligence (KI 2011), Berlin, Germany, 4-7 October 2011, pp. 216-227

[19] H. L. S. Younes, M. L. Littman: PPDDL 1.0: an Extension to PDDL for Expressing Planning Domains with Probabilistic Effects, Techical Report, CMU-CS-04-167, Carnegie Mellon University, Pittsburgh, 2004

[20] F. Teichteil-Königsbuch: Extending PPDDL1.0 to Model Hybrid Markov Decision Processes, Proc. of Workshop on A Reality Check for Plan. and Sched. Under Uncert., ICAPS-08, Sydney, Australia, 15 September 2008

[21] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein: The Complexity of Decentralized Control of Markov Decision Processes, Mathematics of Operations Research, 27(4):819-840, 2002

Appendix 1. Extending MA-PDDL with partial-observability

Appendix 1-3 is best read in conjunction with the BNF (Backus-Naur Form) grammar of MA-PDDL [2] and PDDL3.1 [15]. A new requirement is *added* to MA-PDDL, `:partial-observability`. To capture partial-observability, the *addition* of the following 6 rules is proposed to the BNF of MA-PDDL.

```
<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 the same semantics as events proposed in [17]. *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 and/or incorporated in conditional plans. For the sake of convenience the following 5 rules are also *added* to the grammar.

```
<atomic formula skeleton>  ::=:partial-observability
          (<predicate> <fo> <typed list (variable)>)

<atomic function skeleton> ::=:partial-observability
          (<function-symbol> <fo> <typed list (variable)>)

<fo>                       ::= fo
<fo>                       ::=:multi-agent (fo [<name>])
<fo>                       ::=:multi-agent (fo [<type>])
```

The above 5 rules allow the declaration of full-observability of individual Boolean- and numeric-fluents. An `fo` atom can be placed as the first argument of respective predicate- or function-definitions. The 1st rule for `<fo>` is covers the single-agent case, but if it is used in the multi-agent case, then every agent(object) can observe the corresponding fluent (always, every value). The other two rules for `<fo>` let us specify the type (or even a union of types) of those agents, which always observe every value. If the type is not given, it is assumed to be `object`.

According to our current knowledge, the only work in literature addressing the addition of partial-observability to PDDL is [18]. It adds observations to the probabilistic extension of PDDL (PPDDL1.0) [19] via the addition of observation-predicates that can be referred to in separate observation-effects of actions. Although this concept is simple, it is not clear enough semantically. The partial-observability extension of MA-PDDL proposed in this paper aims to clarify that.

Appendix 2. Further extending MA-PDDL with probabilistic-effects

Now probabilistic-effects are added to MA-PDDL beyond partial-observability based on PPDDL1.0 [19] and its extension to probability-distributions [20]. First the BNF of the effects of discrete actions should be *modified* to the following.

```
<effect>         ::= <p-effect>
<effect>         ::= (and <effect>*)
<effect>         ::=:conditional-effects (forall (<typed list (variable)>) <effect>)
<effect>         ::=:conditional-effects (when <GD> <effect>)
<effect>         ::=:distribution-effects (imply <f-comp> <effect>)
<effect>         ::=:probabilistic-effects (probabilistic <prob-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)
<p-effect>       ::=:rewards (<additive-op> <reward fluent> <f-exp>)
<prob-effect>    ::= <probabilistic list effect>+
<prob-effect>    ::=:distribution-effects + :numeric-fluents <distribution (f-exp)> <effect>
<probabilistic list effect> ::= <probability> <effect>
<probability>               ::= <number>
<probability>               ::=:numeric-fluents <f-exp>
```

PPDDL1.0 [19] is different compared to PDDL3.1 [15] in that it allows `when`-statements to be nested directly into each other. This is the only significant difference. This allows for *conditional probabilistic effects*. Beyond this [20] adds *distribution-effects*, which are also included above in a bit optimized form. They are introduced in the Appendix of [20]. The difference here is that they are parametric (thus grounded or lifted). E.g. the following distributions can be used.

```
<distribution (t)> ::= (gaussian t t <random variate>)
<distribution (t)> ::= (exponential t <random variate>)
<distribution (t)> ::= (uniform t t <random variate>)
<distribution (t)> ::= (poisson t <random variate>)
```

PPDDL1.0 also introduced `:rewards` in form of *reward-fluents* which are not part of the state. Moreover, in PPDDL1.0 `<probability>` was only `<number>`, but in [20] it could be already a function-expression. We used the latter. A further *modification* of the MA-PDDL grammar based on [19,20] is the following.

```
<assign-op>       ::= <additive-op>
<additive-op>     ::= increase
<additive-op>     ::= decrease
```

Similarly to [19] reward-fluents are also *added* to the language in form of the below 2 rules. The second rule allows reward-fluents to be defined for specific agents (refered to as constants, variables or object-fluents) in the multi-agent case. The first rule can be used only in the single-agent case.

```
<reward fluent> ::= total-reward
<reward fluent> ::= :multi-agent (total-reward <term>)
```

A novelty in [20] was the addition of probability-distributions to [19]. A key of this addition was the introduction of *random variates*, which are eventually the actual random values of the distributions, that can be referred in functional-expressions in the below form, after *adding* the below 2 rules to the grammar.

```
<f-exp>          ::=:distribution-effects <random variate>
<random variate>::= #<any char>*
```

PPDDL1.0 and its extension in [20] were discrete, but their ideas can be applied in a straightforward manner to the durative case by *adding* the following 10 rules.

```
<da-effect>       ::=:probabilistic-effects (probabilistic <prob-da-effect>)
<timed-effect>    ::=:continuous-effects + :numeric-fluents + :rewards
                       (<additive-op> <reward fluent> <f-exp-t>)

<f-assign-da>     ::=:numeric-fluents + :rewards
                       (<additive-op> <reward fluent> <f-exp-da>)

<prob-da-effect>  ::= <probabilistic list da-effect>+
<prob-da-effect>  ::=:distribution-effects + :numeric-fluents
                         <distribution (f-exp-da)> <da-effect>
<prob-da-effect>  ::=:distribution-effects + :continuous-effects + :numeric-fluents
                         <distribution (f-exp-t)> <da-effect>

<probabilistic list da-effect> ::= <da-probability> <da-effect>

<da-probability>  ::= <number>
<da-probability>  ::=:numeric-fluents <f-exp-da>
<da-probability>  ::=:numeric-fluents + :continuous-effects <f-exp-t>
```

Probabilistic- and distribution-effects can now define the value of observations with the *addition* of the below 8 rules.

```
<observation-value>   ::=:probabilistic-effects (probabilistic <prob-observation>)

<prob-observation>    ::= <probabilistic list observation>+
<prob-observation>    ::=:distribution-effects + :numeric-fluents
                           <distribution (f-exp)> <observation-value>
<prob-observation>    ::=:distribution-effects + :numeric-fluents + :continuous-effects
                           <distribution (f-exp-t)> <observation-value>

<probabilistic list observation>::= <obs-probability> <observation-value>

<obs-probability>     ::= <number>
<obs-probability>     ::=:numeric-fluents <f-exp>
<obs-probability>     ::=:numeric-fluents + :continuous-effects <f-exp-t>
```

In case of continuous-time probabilities are normalized in runtime to guarantee that they comply with the elementary properties of probability distributions.

```
<problem>      ::= (define (problem <name>)
                         (:domain <name>)
                         [<require-def>]
                         [<object declaration>]
                         <init>
                         <goal>+
                         [<goal-reward>]:rewards
                         [<constraints>]:constraints
                         <metric-spec>*:numeric-fluents
                         [<length-spec>])
```

Above the *modified* rule for describing problems is shown. The only change beyond the previous changes is the *addition* of goal-rewards, similarly to [19].

```
<goal-reward>   ::= (:goal-reward <metric-f-exp>)
<goal-reward>   ::=:multi-agent (:goal-reward
                             [:agent <agent-def>]
                             :reward <metric-f-exp>)
```

If the first rule is used in case of `:multi-agent`, then it refers to all agents. Inheritance/polymorphism of goal-rewards is similar to goals and metric in [2].

```
<metric-f-exp> ::=:rewards <reward fluent>
<metric-f-exp> ::= goal-achieved
<metric-f-exp> ::=:multi-agent (goal-achieved <term>)
```

Similarly to [19] the above 3 rules are *added*. The 3rd rule allows the `goal-achieved` fluent of a specific agent in the metric of any agent. If the 2nd rule is used in MA-case then it is 1 iff the goal of every agent was achieved at least once.

The following collection of rules is based on [20], and *changes* the hitherto description of initial states mainly to optionally include uncertainty.

```
<init-el>          ::= <p-init-el>
<init-el>          ::=:timed-initial-literals <t-init-el>
<init-el>          ::=:probabilistic-effects (probabilistic <prob-init-el>)
<init-el>          ::=:conditional-effects (forall (<typed list(variable)>) <effect>)
<p-init-el>        ::= <literal(name)>
<p-init-el>        ::=:numeric-fluents (= <basic-function-term> <metric-f-exp>)
<p-init-el>        ::=:object-fluents (= <basic-function-term> <metric-o-exp>)
<t-init-el>        ::= (at <number> <p-init-el>)
<t-init-el>        ::= (at <number> (and <p-init-el>*))
<t-init-el>        ::=:probabilistic-effects
                       (at <number> (probabilistic <t-prob-init-el>))
<t-init-el>        ::=:conditional-effects
                       (at <number> (forall (<typed list(variable)>) <effect>))
<prob-init-el>     ::= <probabilistic list init>+
<prob-init-el>     ::=:distribution-effects + :numeric-fluents
                       <distribution (metric-f-exp)> <a-init-el>
<probabilistic list init>   ::= <init-probability> <a-init-el>
<a-init-el>        ::= (and <a-init-el>*)
<a-init-el>        ::= <p-init-el>
<a-init-el>        ::= :timed-initial-literals <t-init-el>
<a-init-el>        ::= (probabilistic <prob-init-el>)
<a-init-el>        ::=:conditional-effects (forall (<typed list(variable)>) <effect>)
<t-prob-init-el>   ::= <probabilistic list t-init>+
<t-prob-init-el>   ::=:distribution-effects + :numeric-fluents
                       <distribution (metric-f-exp)> <t-a-init-el>
<probabilistic list t-init> ::= <init-probability> <t-a-init-el>
<t-a-init-el>      ::= (and <t-a-init-el>*)
<t-a-init-el>      ::= <p-init-el>
<t-a-init-el>      ::= (probabilistic <t-prob-init-el>)
<t-a-init-el>      ::=:conditional-effects (forall (<typed list(variable)>) <effect>)
<init-probability> ::= <number>
<init-probability> ::=:numeric-fluents <metric-f-exp>
```

Appendix 3. Summary of all additional new MA-PDDL requirements

- `:partial-observability` Allows observations in the domain description. It is compatible with multiple agents and probabilistic effects.
- `:probabilistic-effects` Allows discrete probabilistic elements in the effects of durative and non-durative actions and in initial states.
- `:distribution-effects` Allows probability distributions in probabilistic effects of durative and non-durative actions and in initial states.
- `:rewards` Allows reward fluents in effects of durative and non-durative actions and in the metric. It is compatible with `:multi-agent`.
- `:mdp` **=** `:probabilistic-effects + :rewards`
- `:pomdp` **=** `:mdp + :partial-observability`
- `:dec-mdp` **=** `:mdp + :multi-agent`
- `:dec-pomdp` **=** `:pomdp + :multi-agent`

I.e. MA-PDDL with partial-observability+probabilistic-effects can describe DEC-POMDPs (DECentralized Partially Observable Markov Decision Processes) [21].