A Multi-Agent Extension of Hierarchical Task Network

Rafael C. Cardoso[1] **and Rafael H. Bordini**[1]

[1]School of Informatics – FACIN-PPGCC
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brazil

rafael.caue@acad.pucrs.br, rafael.bordini@pucrs.br

Abstract. *Describing planning domains using a common formalism promotes greater reuse of research, allowing a fairer comparison between different approaches. Common planning formalisms for single-agent planning are already well established (e.g., PDDL, STRIPS, and HTN), but currently there is a shortage of multi-agent planning formalisms with clear semantics. In this paper, we propose a multi-agent extension of the Hierarchical Task Network (HTN) planning formalism. Our formalism, the Multi-Agent Hierarchical Task Network (MA-HTN), can be used in the representation of multi-agent planning domains and problems. We provide a grammar for the domain and problem representation, and show a case study with the translation from a JaCaMo system, a multi-agent system development platform, to our MA-HTN formalism.*

1. Introduction

Multi-Agent Systems (MAS) are often situated in dynamic environments where new plans of action need to be constantly devised in order to successfully achieve the system goals. Therefore, employing planning techniques during run-time of a MAS can be used to improve agent's plans using knowledge that was not previously available, or even to create new plans to achieve some goal for which there was no known course of action at design time.

Finding a solution to a planning problem consists of the following process: given a description of the initial states of the world (e.g., agents, environment), a description of the desired goals, and a description of a set of possible actions, the problem consists in finding a set of plans (i.e., sequence of actions) that when executed from any of the initial states will lead to the achievement of a goal. Therefore, it is beneficial to have a planning formalism in order to formally represent these problems, defining the syntax of the languages that are used to describe all of these descriptions.

The choice of a planning formalism is usually dependent on which planner is being used. The reason behind this is that most planners have their own formalism, or at least a variation of a well-defined one previously developed and accepted by the planning community. Because multi-agent planning research has been just recently getting more attention from the planning community, there is no de-facto standard to represent multi-agent planning domains yet.

In this paper we propose the Multi-Agent Hierarchical Task Network (MA-HTN) planning formalism, allowing the representation of multi-agent planning domains and problems for HTN planning. Because we are dealing with dynamic MAS, it means that we need to keep the description of the domain and problem constantly updated, in order to provide the online planning with the best information possible, thus increasing the chances of it finding a viable solution. We show a case study where we developed a JaCaMo MAS of the Rover planning domain, and use a MA-HTN translator to parse information about the world currently available to the JaCaMo system into MA-HTN domain and problem representations.

The rest of this paper is structured as follows. In the next section we cover the necessary background on multi-agent systems and automated planning. Section 3 introduces our Multi-Agent extension to Hierarchical Task Network (MA-HTN), along with the grammar for the representation of domain and problems. Next, in Section 4, we use the Rover domain as our case study, to show the translation process from a MAS into MA-HTN. In Section 5, we discuss related work, and we end the paper with a description of future work and some concluding remarks.

2. Background

In this section we provide a brief background on multi-agent systems and automated planning. We start by describing the different abstraction levels that can be considered for programming multi-agent systems, and give an overview of JaCaMo, the multi-agent system development platform that was used in our case study. Next, we give a succinct description of the Hierarchical Task Network (HTN) planning formalism, discuss multi-agent planning and its different phases, and contextualise where our approach was designed to be used.

2.1. Programming MAS with Multiple Abstraction Levels

According to Bordini and Dix in [Bordini and Dix 2013], “Originally agent programming languages were mostly concerned with programming individual agents, and very little was available in terms of programming abstractions covering the social and environmental dimensions of multi-agent systems as well as the agent dimension”. These multiple abstraction layers are what make multi-agent oriented programming especially suited for solving complex problems that require highly social, autonomous software. We now describe the social and environmental dimensions.

Organisations in a multi-agent system are complex entities in which agents interact in order to achieve some global purpose [Dignum and Padget 2013]. They provide scope for these interactions, reduce or manage uncertainty, and coordinate agents to improve efficiency. This is especially relevant to MAS in complex, dynamic, and distributed domains. These domains are very similar to those that can be found in multi-agent planning.

Environments in agent-based systems can be virtual or physical, or even in some cases both, as it can be beneficial to simulate parts of a physical environment as virtual elements. There are two main different views on the concept of environments in MAS. In classical AI, environment represent the external world that is perceived and acted upon by the agents so as achieve their goals [Russell and Norvig 2009]. A more recent view classify the environment as a first-class abstraction that encapsulates functionalities to support agent activities [Weyns et al. 2007].

2.1.1. JaCaMo

JaCaMo[1] [Boissier et al. 2011] combines three separate technologies into a platform for MAS programming that makes use of multiple levels of abstractions. Each technology (Jason, CArtAgO, and Moise) was developed separately for a number of years and are fairly established on their own when dealing with their respective abstraction level (agent, environment, and organisation). The overview of JaCaMo can be observed in Figure 1.

Moise [Hübner et al. 2007] handles the organisation level. Agents can adopt roles in the organisation, forming groups and sub-groups. Missions are defined to achieve the organisation goals. The behaviour of the agents that adopt roles to execute these missions is guided by norms.

Jason [Bordini et al. 2007] is responsible for the agent level, it is an extension of the AgentSpeak language. Based on the Belief-Desire-Intention architecture, agents in Jason react to events in the system by executing actions on the environment, according to the plans available in each agent's plan library.

CArtAgO [Ricci et al. 2009] is based on the A&A (Agents and Artefacts) model, and deals with the environment level. Artefacts are used to represent the environment, storing information about the environment in observable properties and providing actions that can be executed through operations. When an agent focuses on an artefact, it receives the observable properties as beliefs, and it is able to execute the artefact's operations. Artefacts are grouped in workspaces, which can be distributed across multiple network nodes, providing a natural distribution to the MAS.

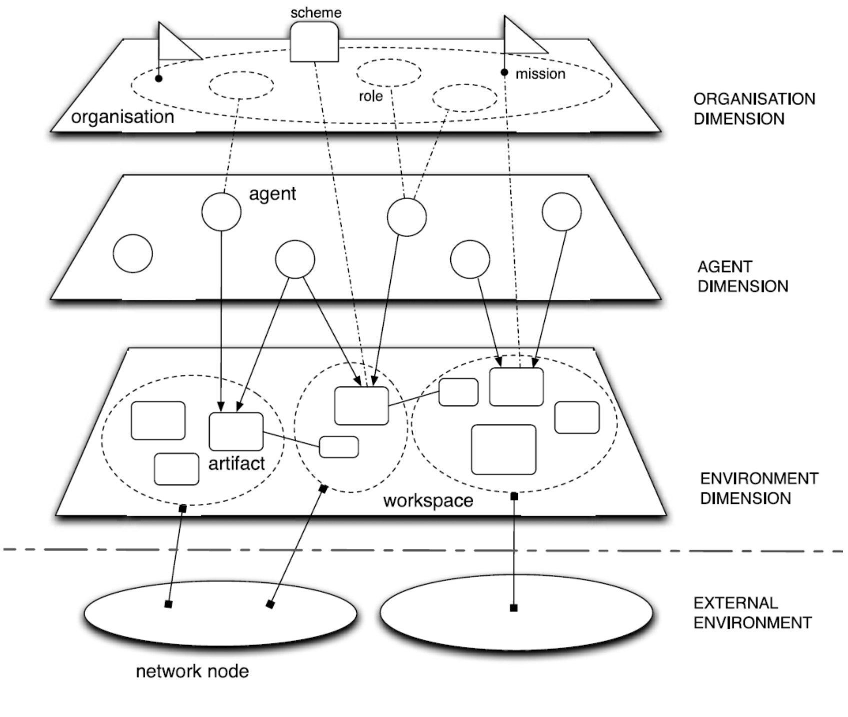

Figure 1. JaCaMo overview [Boissier et al. 2011].

[1]`http://jacamo.sourceforge.net/`.

2.2. Automated Planning

Automated planning is the computational study of planning, which is an abstract deliberation process on choosing and ordering actions in order to achieve some goals in the best way possible [Nau et al. 2004]. This is done by anticipating the outcome of these actions, but not every goal needs to be planned out. The deliberation process can take some time to find the best possible solution, thus sometimes merely reacting is the best approach, and then adapting to the consequences.

2.2.1. Hierarchical Task Network

HTN planning [Nau et al. 2004] is one of the techniques for automated planning. It works by decomposing tasks into subtasks, until arriving at primitive tasks that can solve the planning problem. This convenient way of writing recipes is more closely related to how a human expert would think about solving a planning problem, thus making HTN planning more suited for practical applications. Besides, the extra domain information contained in non-primitive tasks usually results in better performance than with the other types of planners.

An HTN planning domain representation contains a set of operators and a set of methods. Operators are action descriptors that can be executed given some preconditions, causing a list of postconditions to become true. They can cause a state transition to occur in the system, while methods can only decompose tasks into smaller subtasks, which can eventually lead to primitive tasks wherein an operator can be applied. An HTN planning problem representation contains a list of atoms that are true during the initial state of the system, as well as the goals of the system.

2.2.2. Multi-Agent Planning

Multi-Agent Planning (MAP) has often been interpreted as two different things. Either the planning process is centralised and produces distributed plans that can be acted upon by multiple agents, or the planning process itself is multi-agent. Recently, the planning community has been favouring the concept that MAP is actually both of these things, that is, the planning process is done *by* multiple agents, and the solution is *for* multiple agents.

When considering multiple agents the planning process gets increasingly more complicated, giving rise to several problems [Durfee and Zilberstein 2013]. Actions that agents choose to make may cause an impact in future actions that the other agents could take. Likewise, if an agent knows what actions the other agents plan to take, it could change its own current choices. When dealing with multiple agents, concurrent actions are also a possibility and have to be dealt with. These are some of the problems that drive the research on MAP.

Durfee also establishes some useful phases for multi-agent planning in [Durfee 1999], further extended in [Weerdt and Clement 2009]: 1) *global goal refinement*, decomposition of the global goal into subgoals; 2) *task allocation*, use of task-sharing protocols to allocate tasks (goals); 3) *coordination before planning*, coordination mechanisms that prevent conflicts before planning; 4) *individual planning*,

planning algorithms that search for solutions for the problem; 5) *coordination after planning* coordination mechanisms that prevent conflicts after planning; and 6) *plan execution*, the agents carry out the solution found.

3. The MA-HTN Formalism

A planning problem consists of the following process: given a description of the initial states of the world (e.g., agents, environment), a description of the desired goals, and a description of a set of possible actions, the problem consists of finding a set of plans (i.e., sequence of actions) that when executed from any of the initial states will lead to the achievement of a goal. Therefore, it is beneficial to have a planning formalism in order to formally represent these problems, defining the syntax of the languages that are used to describe all of these representations.

We propose the Multi-Agent Hierarchical Task Network (MA-HTN) formalism, which is an extension of the centralised single-agent HTN formalism used in the SHOP2 planner [Nau et al. 2003]. MA-HTN is intended to represent *online* multi-agent planning problems, since domain and problem information are collected during execution. Thus, unlike in offline planning where these two can be specified before execution (e.g., by a system designer or a computer script), here there is a need for a mechanism to collect all of the necessary data and translate it to an input that can be useful to a planner.

We call this mechanism the *translator*, and agents use it to translate their information about the world into domain and problem specifications that can then be passed to a planner. The translator obtains information about the current state of the world from the MAS in execution, using it to generate the problem representation. The domain representation is generated from the possible actions and plans that the agents have access to.

Each agent has their own problem and domain specification. This provides a decent level of privacy on its own, since each planner only has access to their respective agent problem and domain specifications. This means that, unlike some of the other multi-agent planning formalisms, MA-HTN does not need to have privacy or public blocks. Although at some point it might be interesting to add the capability to include private goals into the formalism, for now we are interested only on representing organisational goals.

Actions from other agents can cause conflicts, either at the moment that action is executed (e.g., concurrent actions) or in the future (e.g., durative actions). For this reason, MA-HTN supports the characterisation of actions that can cause *conflict*. The recognition of these actions is not automated, though a mechanism for that purpose could be used. These actions that can cause conflicts have to be annotated by the MAS developer, in order for the translator to be able to identify them. Actions are always translated to operators in HTN, thus, only operators can cause conflicts, methods cannot.

Likewise, dependencies between actions can also exist, either as a concurrent action that requires another agent or actions that depend on actions of other agents to happen first. Similarly to conflicts, MA-HTN also supports the use of *dependency* blocks to identify actions that depend on actions from other agents. These dependency relations also have to be annotated by the MAS developer, so that the translator can add them to the specification. They also cannot be used in methods, only in operations.

The notation usually preferred to specify context-free grammars is the Backus–Naur Form (BNF). We use BNF grammars to define the specifications of MA-HTN domain and problem representations. We provide a simplified grammar to improve readability, where each single quote pair that encloses a symbol is considered a string that is expected by the planner, symbols enclosed by brackets are optional, and symbols preceded by $ are variables obtained from the MAS and represent terminal symbols. Symbols that end with the * signal, represent that zero or more instances are possible. Symbols that end with the $^+$ signal, represent that one or more instances are possible. Because MA-HTN is based on the SHOP2 HTN formalism [Nau et al. 2003], the language itself is LISP-like, though we omitted many of the necessary parenthesis in order to improve readability.

3.1. Domain Representation

In Listing 1, we show our simplified BNF grammar for multi-agent domains. The *$domain-name* variable is defined dynamically by the agent during execution of the MAS, and since there could be multiple calls to the same domain and the specification can be different from the previous call (e.g., a method could be deleted, added, or modified), agents use a counter id that increments each time planning is invoked. The name of the agent, *$agent-name* is included in the specification to represent which agent this domain belongs to. The *conflict-list* and *dependency-list* are added. Conflicts represent actions that can cause negative interactions between agents, while dependencies are actions that need other agents to succeed. The rest of the definitions are similar to SHOP2 HTN, *operators* are *primitive-tasks* while *methods* are *non-primitive-tasks* that can eventually be decomposed into *operators*.

Listing 1. MA-HTN BNF grammar for representing domains.

```
def-domain          ::= 'defdomain' $domain-name ;
agent               ::= 'agent' $agent-name ;

task                ::= primitive-task | non-primitive-task ;
primitive-task      ::= '!'$primitive-task-name '?'$variable* ;
non-primitive-task  ::= $non-primitive-task-name '?'$variable* ;

 def-operator       ::= ':operator' primitive-task precondition-list delete-list add-list conflict-list
        dependency-list ;
precondition-list   ::= precondition* ;
precondition        ::= ('not' $precondition-name '?'$variable*) |
    ($precondition-name '?'$variable*) ;
delete-list         ::= delete* ;
delete              ::= $delete-name '?'$variable* ;
add-list            ::= add* ;
add                 ::= $add-name '?'$variable* ;
conflict-list       ::= conflict* ;
conflict            ::= $action-name $agent-name ;
dependency-list     ::= dependency* ;
dependency          ::= $action-name $agent-name ;

def-method          ::= ':method'" non-primitive-task (precondition-list task-list+) ;
task-list           ::= task+ ;
```

3.2. Problem Representation

Listing 2 shows our simplified BNF grammar for multi-agent problems. Similarly to the domain grammar, the *$problem-name* is specified, along with a reference to its respective *$domain-name*. We also have the explicit agent symbol on line 2. Remember that in our formalism each agent has its own domain and problem representations, which gives some sense of natural privacy to the system. *Facts* can be used to establish types and characteristics of things in the world, for example, *location kitchen* establishes that *kitchen* is a *location* in the world. While *initial-states* are used to represent what is true in the world at that specific moment in time, for example, *kitchen dusty* represents that the *kitchen* is *dusty*. *Goals* can be listed as either *ordered* – when the order in which the goals have to be achieved needs to be strictly followed; or *unordered* – when the order is unknown and the planner is free to find any order between goals.

Listing 2. MA-HTN BNF grammar for representing problems.

```
1  def-problem         ::=  'defproblem' $problem-name $domain-name ;
2  agent               ::=  'agent' $agent-name ;
3
4  def-facts           ::=  fact-list ;
5  fact-list           ::=  fact* ;
6  fact                ::=  $fact-name $fact-parameter+ ;
7
8  def-initial-states  ::=  initial-state-list ;
9  initial-state-list  ::=  initial-state+ ;
10 initial-state       ::=  $initial-state-name $initial-state-parameter+ ;
11
12 def-goals           ::=  (':ordered' | ':unordered') goal-list ;
13 goal-list           ::=  goal+ ;
14 goal                ::=  $goal-name $goal-parameter+ ;
```

4. Case Study

As our case study we use the Rover domain, which has been used in several past IPCs. The Rovers domain was constructed as a simplified representation of the NASA Mars Exploration Rover missions launched in 2003 and other similar missions. This domain involves planning for several rovers, equipped with different, but possibly overlapping, sets of equipment to traverse a planet surface. The rovers must travel between waypoints gathering data and transmitting it back to a lander. The traversal is complicated by the fact that certain rovers are restricted to travelling over certain terrain types and this makes particular routes impassable to some of the rovers. Data collection involves the collection of rock and soil samples located in waypoints, as well as taking images (three different modes are available: high_res, low_res, colour) of certain objectives that are visible from waypoints. Data transmission is also constrained by the visibility of the lander from the waypoints.

The Rover domain was initially conceived as a domain for single-agent planning. Thus, we made a few extensions to turn it into a suitable multi-agent planning domain, such as further specialising rover vehicles into: vehicles for rock analysis, vehicles for soil analysis, and photographer vehicles. We also establish that the action to transmit data to the lander is a possible point of conflict, since the channel might be busy (e.g., another agent is currently transmitting its data). A dependency in this domain is that collections of soil and rock that are located in a waypoint from which an active objective (e.g., a goal for taking images of the objective exists) is visible from, can only be collected after the image of the active objective has been taken.

We implemented this domain as a MAS using the JaCaMo MAS development platform. The *.jcm* configuration file contain all of the information necessary for starting the MAS, and we used it to set up the problem for the Rover domain. We added a *ground_team* agent, which is represented as an agent here just to simulate the input provided by humans, such as creating new dynamic goals for the robots during execution of the MAS. There are three rovers in this instance, one for each specialisation (rock analysis, soil analysis, and photographer). Then, we created four waypoint artefacts, one for each of the four waypoints in this problem, two artefacts for the objectives, and one for the lander. Each agent also has its own personal artefact, containing its starting parameters and the actions that it is able to perform. The goals in this problem are to take images of *objective1* and *objective2*, to collect soil data of *waypoint3*, and to collect rock data of *waypoint4*.

The MA-HTN translator generates problem and domain representations for each agent, as follows:

- **Problem representation:** The name of the problem and the name of the domain are obtained dynamically. The name of the agent is gathered by using a Java function that returns the name of the agent who started the translator. The information collected from the CArtAgO artefacts are parsed into initial states that form the agent's fact list and initial state list. The goal list is created from the organisational goals that were assigned to this particular agent during the goal allocation phase.
- **Domain representation:** The name of the name of the domain is obtained dynamically. The name of the agent is gathered by using a Java function that returns the name of the agent who started the translator. Operators are parsed from all of the artefacts operations that the agent has access to. The precondition list is obtained from any conditional tests in an operation, the delete and add list from the deletion and addition of observable properties, and the conflict and dependency lists need to be previously annotated into the operation in order for them to be able to be parsed. The methods are parsed from all of the plans in the agent's plan library, with the precondition list parsed from the context of the plan and the task list parsed from the body of the plan.

In Listing 3, we show the *.jcm* code with some of the configuration of our case study. For example, agent *rover1* contains the name, code filename of the agent, and all of the actions that agents should take when the system starts. For instance: which workspace they should *join*; what artefacts they should *focus*; and the roles that they should *adopt*.

Listing 3 also shows some of the environment artefacts, such as the *waypoint1* artefact that sets its three observable properties: *visible*, which determines the list of waypoints that can be reached to transmit data, in this case waypoints 2, 3, and 4; *rock_sample*, which determines if there are rock samples available in this waypoint, in this case there is; and *soil_sample*, which determines if there are soil samples available in this waypoint, in this case there is not. We also show as an example the workspace with the artefact for rover1, with its two observable properties: *at*, which waypoint the agent is currently at, in this case the agent is at *waypoint2*; and *can_traverse*, which contains the list of terrains that the vehicle is capable of traversing.

Listing 3. A snippet of the .jcm file for the Rover case study.

```
agent rover1 : rover.asl {
      join: w1, r1
      focus: w1.lander, w1.way1, w1.way2, w1.way3, w1.way4, w1.obj1, w1.obj2, r1.rover1
      roles: rock_analysis in o1.g1
}
workspace w1 {
      artifact way1: rovers.Waypoint(["way2","way3","way4"],"true","false")
      artifact obj1: rovers.Objective(["way1","way2","way3","way4"])
      artifact lander: rovers.Lander("way1","free","")
      ...
}
workspace r1 {
      artifact rover1: rovers.Rover("way2",[par(way2,way1),par(way1,way2),par(way2,way4),par(
      way4,way2)],"obj2")
}
```

In the Moise structural specification, Listing 4 of the organisation, five roles are defined, with three of them being specialisations of the *rover* role. The group specification (omitted from the Listing) defines the cardinality for each role that is required to fully form the group, which reflects the number of agents in the system in our case, 1 ground team, 1 rover for rock analysis, 1 rover for soil analysis, and 1 photographer rover. Group specifications also contain the designation of links. In this case there are two links, an authority link from the *ground_team* to *rover* roles, and an acquaintance link from *rover* to *rover*. The authority link means that the *ground_team* can send goals and command directives to *rovers*, while the acquaintance link allows *rovers* to communicate with each other.

Listing 4. The structural specification of the rover organisation.

```
<structural-specification>
      <role-definitions>
       <role id="ground_team" />
       <role id="rover" />
            <role id="rock_analysis" >             <extends role="rover"/> </role>
            <role id="soil_analysis" >             <extends role="rover"/> </role>
            <role id="photographer" >             <extends role="rover"/> </role>
      </role-definitions>
      <links>
            <link from="ground_team" to="rover" type="authority" scope="
      intra-group" extends-subgroups="false" bi-dir="false"/>
            <link from="rover" to="rover" type="acquaintance" scope="
      intra-group" extends-subgroups="false" bi-dir="false"/>
      </links>
</structural-specification>
```

We show an example of a plan in Jason in Listing 5 of the photographer agent, and an operation from the photographer agent artefact in Listing 6. In the problem representation, CArtAgO observable properties (i.e., the current information about the state of the world) become the initial states. The goals of the organisation become the goal list. Eventually we would like to directly extract these goals from the Moise specification, but we are still investigating the best way to do this. In the domain representation, CArtAgO operations (which are the actions that can be executed in the environment) become operators, and the plans from the Jason agents plan library become methods.

Listing 5. Example of a Jason plan.

```
+!get_image_data(Objective, Mode)
    : visible_from(Objective,Area) & supports(Mode) & .my_name(Name)
<-
    !navigate(Area);
    take_image(Area,Objective)[artifact_id(Name)].
```

Listing 6. Example of a CArtAgO operation.

```
@OPERATION void take_image(String objective, String visible_from) {
    ObsProperty cond1 = getObsProperty("at");
    if ( visible_from.stringValue().contains(cond1.toString()) )
    {
        defineObsProperty("have_image", objective);
    } else {
        failed("Action take_image has failed.");
    }
}
```

5. Related Work

STRIPS is an early automated planning system from 1971 [Fikes and Nilsson 1971], that still provides the basis for many classical planners with its action theory and formalism. In STRIPS, each operator has a precondition list, add list, and delete list. These lists were allowed to contain arbitrary well-formed formulas in first-order logic. However, there were a number of problems with this formulation, such as the difficulty of providing a well-defined semantics for it [Nau et al. 2004]. The PDDL (Planning Domain Definition Language) contains STRIPS-like operators, and has been the formalism of choice in several past IPCs. The latest version of PDDL is 3.1[2].

In HTN planning, a well-established formalism is the one accepted by the SHOP2 planner [Nau et al. 2003]. It differs from PDDL in the sense that it does not necessarily involve state variables. In this formalism, planning domains contain a set of operators, methods, and axioms. Planning problems contain a set of logical atoms (i.e., facts that represent the initial state of the world), and tasks lists (i.e., high-level goals to achieve).

[2]http://ipc.informatik.uni-freiburg.de/PddlExtension

Although it is possible to represent multiple agents using those formalisms, there is no distinction between agents and the other objects of the world. This makes it difficult to represent important features of MAP, such as conflicts, dependencies, privacy, and distribution. Thus, many single-agent formalisms have been expanded to allow for the explicit description of agents. For example, in MA-STRIPS [Brafman and Domshlak 2008] the authors propose a multi-agent extension of STRIPS formalism for cooperative multi-agent systems. Besides adding the notion of agents containing their own set of actions, dependencies can be identified to classify an agent's actions into internal or public.

A multi-agent extension of PDDL 3.1 [Kovacs 2012] was designed to cope with the agents' different abilities and the constructive and destructive nature of concurrent actions. Its design is based on several requirements for multi-agent planning formalisms, those of note are: modelling concurrent actions with interacting effects; agents can have different actions/goals/utilities; straightforward association of agents and actions; distinction between agents and non-agent objects; and inheritance/polymorphism of actions/-goals/utilities. The extension can also be used to represent these problems in temporal, numeric domains.

6. Conclusions

In this paper we described the MA-HTN formalism, a multi-agent variation of the traditional single-agent HTN formalism. We also described a translator that can parse current information about the world available to a JaCaMo system into MA-HTN domain and problem representations. A case study with the Rover domain was used to illustrate the translation process and helped in comparing with the traditional (single-agent) HTN formalism.

By using agents as first-class abstractions in our formalism, we are free of the use of predicates that are usually used to assign tasks between different objects. In turn, this also lowers the number of expansions and inferences required during the planning process, which should improve planning time and performance. At any point during the execution of the MAS, the parameters of the problem may change, new dynamic goals may be created, paths can become obstructed, new agents might join the system, etc. Thus, our formalism serves as a bridge to online planners, allowing access to up-to-date information about the current state of the MAS.

Future work consists of extending the MA-HTN formalism to allow for other types of possible agent interactions, besides conflicts and dependencies as we presented here, such as privacy and distribution. It is also important to make experiments in other domains, as well as to check the suitability of translating MAS into MA-HTN domains (instead of developing MAS from planning domains). Finally, experiments with the use of the MA-HTN formalism in a planner also have to be conducted in order to provide a practical evaluation of the formalism. Another useful extension to MA-HTN is to be able to translate the solution found by a planner into plans that can be added to the respective agent's plan library.

Acknowledgements

We are grateful for the support given by CAPES and by CNPq (grant number 308095/2012-0).

References

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2011). Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*.

Bordini, R. H. and Dix, J. (2013). Programming multiagent systems. In Weiss, G., editor, *Multiagent Systems 2nd Edition*, chapter 11, pages 5870–639. MIT Press.

Bordini, R. H., Wooldridge, M., and Hübner, J. F. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons.

Brafman, R. I. and Domshlak, C. (2008). From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *ICAPS*, pages 28–35.

Dignum, V. and Padget, J. (2013). Multiagent organizations. In Weiss, G., editor, *Multiagent Systems 2nd Edition*, chapter 2, pages 51–98. MIT Press.

Durfee, E. H. (1999). Distributed problem solving and planning. In *Mutliagent systems*, pages 121–164. MIT Press.

Durfee, E. H. and Zilberstein, S. (2013). Multiagent planning, control, and execution. In Weiss, G., editor, *Multiagent Systems 2nd Edition*, chapter 11, pages 485–545. MIT Press.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd international joint conference on Artificial intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2007). Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *Int. J. Agent-Oriented Software Engineering*, 1(3/4):370–395.

Kovacs, D. L. (2012). A multi-agent extension of pddl3.1. In *Proceedings of the 3rd Workshop on the International Planning Competition (IPC)*, ICAPS-2012, pages 19–27, Atibaia, SÃ£o Paulo, Brazil.

Nau, D., Ghallab, M., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nau, D., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009). Environment programming in CArtAgO. In *Multi-Agent Programming: Languages, Tools and Applications*, Multiagent Systems, Artificial Societies, and Simulated Organizations, chapter 8, pages 259–288. Springer.

Russell, S. J. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.

Weerdt, M. d. and Clement, B. (2009). Introduction to Planning in Multiagent Systems. *Multiagent Grid Syst.*, 5(4):345–355.

Weyns, D., Omicini, A., and Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30.