Combining off-line Multi-Agent Planning with a Multi-Agent System Development Framework

Rafael C. Cardoso and **Rafael H. Bordini**
FACIN-PUCRS
Porto Alegre - RS, Brazil
{rafael.caue@acad.pucrs.br, rafael.bordini@pucrs.br}

Abstract

Automated planning is an important capability to have in multi-agent systems. Extensive research has been done for single-agents, but so far it has not been fully explored in multi-agent systems mainly because of the computational costs of multi-agent planners. With the increasing availability of distributed systems, and more recently multi-core processors, there have been several novel multi-agent planning algorithms developed, such as the MAP-POP algorithm, which in this work we integrate with the JaCaMo multi-agent system framework. Our work provides off-line multi-agent planning capabilities as part of a multi-agent system development framework that supports the development of systems for complex multi-agent problems. In summary, our approach is to provide to developers an initial multi-agent system implementation for the target scenario, based on solutions found by the MAP-POP multi-agent planner, and on which the developer can work further towards a fully-fledged multi-agent system.

1 Introduction

In this paper, we provide an approach to combine off-line Multi-Agent Planning (MAP) algorithms with a Multi-Agent Systems (MAS) platform. Our work serves two purposes: it provides an execution stage for off-line MAP, and it provides an initial MAS on which developers can further work. This process is done with a translator, which receives as input the problem instances and the solution found by the multi-agent planner and generates a MAS program as output.

Automated planning is an interesting and desirable capability to have in intelligent agents and MAS, which so far has not been fully explored because of the computational costs of MAP algorithms (Jonsson and Rovatsos 2011; Crosby, Jonsson, and Rovatsos 2014). Recent algorithms have managed to improve performance, which was one of the main incentives for pursuing this topic.

Although there is an increase in interest in theoretical research on multi-agent planning, as evidenced in (Witwicki and Durfee 2011; Jr. and Durfee 2011; Planken, de Weerdt, and Witteveen 2010), current implemented multi-agent planning algorithms and planners are mostly application specific, such as in (Mao et al. 2007; van Leeuwen and Witteveen 2009).

A distributed multi-agent planning problem involves the development and execution of joint plans through cooperation (agents on the same team) and competition (agents on opposing teams) without centralised control. These off-line planning algorithms normally stop at the planning stage, providing a solution but with no means of executing it.

The term multi-agent planning has been used in a variety of contexts through the years, and as such, its concept can mean widely different things. For the purposes of this paper, we use the multi-agent planning definition found in (Durfee and Zilberstein 2013), which states that the planning process itself is multi-agent (i.e. multiple agents cooperatively generate plans), and the solution can be distributed across and acted upon by multiple agents. In other words, multi-agent planning by multiple agents and multi-agent planning for multiple agents.

We use the Multi-Agent Planning based on Partial-Order Planning (MAP-POP) (Lerma 2011; Torreño, Onaindia, and Sapena 2014a; 2014b; Sapena, Onaindia, and Torreño 2015) as an example of a multi-agent planner, and provide a basic grammar for it. This grammar is then used in the translation algorithms that we describe. These algorithms can be easily adapted to work with the grammars of other multi-agent planners.

For the MAS development platform we use the JaCaMo framework (Boissier et al. 2011). JaCaMo is composed of three technologies, each representing a different abstraction level that is required for the development of sophisticated MAS. JaCaMo is the combination of Jason, CArtAgO, and Moise, each of these technologies are responsible for a different programming dimension. Jason is used for programming the agent level, CArtAgO is responsible for the environment level, and Moise for the organisation level.

The translator takes as input the problem instances and the solution provided by a multi-agent planner, MAP-POP in this paper. As output, the translator generates a coordination scheme in Moise, followed by the respective agent plans in Jason, and CArtAgO artefacts for organisation control and environment representation. All of these come together to form an initial multi-agent system in JaCaMo.

Our goal with this work was to check if current general-purpose MAP algorithms could be easily integrated with a MAS framework in order to execute the solution, and if the resulting MAS in JaCaMo was complex enough to be of any help to developers. Our results are encouraging, the coordination aspect that is needed for the distributed planning stages of a MAP algorithm is a natural fit for the specification of a Moise organisation. The plans generated in Jason are parsed from the solution presented by each algorithm, and generally were a simple conversion from a step to a plan, maintaining its pre-conditions and effects.

The rest of this paper is structured as follows. In Section 2 we cover the background. Section 3 provides some of the related work. In Section 4 we describe the grammar that we made for the input and the solution of MAP-POP and the translation algorithms. Section 5 presents two case studies and we conclude in Section 6.

2 Background

It is common for MAP algorithms to use and improve upon single-agent planning techniques, as single-agent planning has been extensively researched over the years and has also been the focus of several International Planning Competitions (IPC). As a consequence, single-agent planners generally have an excellent performance. Usually, in distributed MAP algorithms, agents plan locally using adaptations of single-agent planners, which means that at some point they will need to exchange information to be able to arrive at a global solution plan. Therefore in order to properly exchange information the agents need some kind of coordination mechanism.

The input of a MAP algorithm refers to instances of the formalism chosen to represent MAP planning problems. Similarly to single-agent planners, problem formalisms have also been extensively researched over the years. PDDL for example has been the standard formalism to represent single-agent problems for quite some time, and it can be easily adapted to comply with the needs that arise when dealing with multi-agent planning problems. The output of a MAP algorithm is the solution it generates, and contrary to the input, the output does not have any standard representation. However, as we are dealing with multi-agent plans that can cause interference with each other, coordination constraints are needed to guarantee that during the execution stage the partial plans will be executed in the correct order so as to achieve the global goal.

The MAP-POP (Lerma 2011) planner builds upon the concept of refinement planning, where agents propose successive refinements to a base plan until a solution is obtained. It uses the PDDL 3.1 formalism with some ad-hoc adaptations to make it work with their multi-agent planner. MAP-POP is based on partial-order planning, establishing partial order relations between the actions in the plan.

The MAP-POP algorithm starts with an initial communication stage in which the agents exchange some information on the planning domain, in order to generate data structures that will be useful in the subsequent planning process. The next step comprises of two different stages that are interleaved, they repeat themselves until a solution plan is found:

- **an internal planning process**, through which the agents refine the current base plan individually with an internal POP system. In order to guide the search the SUM heuristic is applied, it is based on the sum of the costs of the open goals found in the initial communication stage.
- **and a coordination process**, that allows agents to exchange the refinement plans that were generated in the previous stage and to select the next base plan, using the SUM heuristic to estimate the quality of a refinement. A leadership baton is passed among the agents, following a round-robin order. If the current base plan does not have any open goals it is a solution, and if not the baton agent selects the next most costly open goal to be solved. With a new subgoal to be solved, the next internal planning stage starts.

According to (Wooldridge 2002), an agent is a computer system that is capable of autonomous action in the environment that it is situated in order to meet its objectives. In other words, agents receive perceptions through sensors in the environment, and respond to these events with actions that affect the environment. Systems that require the use of the Agent Model will seldom need only a single-agent. Albeit obvious, a MAS then is composed of multiple agents.

Many agent-oriented programming languages have been developed over the years. Some examples of these include Jason (Bordini, Wooldridge, and Hübner 2007), JACK (Busetta et al. 1999), 2APL (Dastani 2008), GOAL (Hindriks et al. 2000), and more recently ALOO (Ricci 2014).

Several studies indicate that Jason has an excellent performance when compared with other agent-oriented programming languages. For example, Jason is included in a qualitative comparison of features alongside with Erlang and Java (Jordan et al. 2011); in a universal criteria catalog for agent development artefacts (Braubach, Pokahr, and Lamersdorf 2008); in a quantitative comparison between Jason and two actor-oriented programming languages (Erlang and Scala) using a communication benchmark (Cardoso, Hübner, and Bordini 2013); and finally a performance evaluation of several benchmarks between agent programming languages (Jason, 2APL, and GOAL) and actor programming languages (Erlang, Akka, and ActorFoundry) (Cardoso et al. 2013). In those cases where performance was considered, Jason typically showed excellent results.

A JaCaMo[1] (Boissier et al. 2011) MAS (i.e. a software system programmed in JaCaMo) is defined by an agent organisation programmed in Moise, responsible for the organisation of autonomous agents programmed in Jason. Those agents work in a shared distributed artefact-based environment programmed in CArtAgO. JaCaMo integrates these three platforms by defining a semantic link among concepts of the different programming dimensions (agent, environment, and organisation) at the meta-model and programming levels, in order to obtain a uniform and consistent programming model that simplifies the combination of those dimensions for the development of MAS.

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

Jason (Bordini, Wooldridge, and Hübner 2007) focuses on the agent programming level, it is a programming language for the development of MAS based on the BDI (Belief-Desire-Intention) model, inspired by the AgentSpeak language (Rao 1996). In Jason an agent is an entity composed of a set of beliefs — agent's current state and knowledge about the environment in which it is situated; a set of goals — tasks the agent has to achieve; a set of intentions — tasks the agent is committed to achieve; and a set of plans — courses of actions triggered by events (can be related to changes in either the agent's belief base or its goals).

CArtAgO (Ricci et al. 2009) is a framework and infrastructure for environment programming and execution in multi-agent systems. In CArtAgO the environment is used as a first-class abstraction for designing MAS, a computational layer encapsulating functionalities and services that agents can explore at runtime. These software environments can be designed and programmed as a dynamic set of computational entities called artefacts, that are collected into several workspaces, possibly distributed among various nodes of a network.

Finally, the Moise (Hübner, Sichman, and Boissier 2007) model is used to program the organisational dimension. This approach includes an organisation modelling language, an organisation management infrastructure, and support for organisation-based reasoning mechanisms at the agent level. The organisation model is divided into three layers: the structural specification, where the groups, roles, and links between roles are specified; the functional specification, where the schemas are specified, containing a group of goals and missions, along with information on which goals will be executed in parallel and which will be executed in sequence; and the normative specification, where obligations and permissions towards certain missions are assigned to certain roles.

3 Related Work

A survey (Meneguzzi and De Silva 2013) presents a collection of recent techniques used to integrate automated planning in BDI-based agent-oriented programming languages. It focuses mostly on efforts to generate new plans at runtime, while as with our work we translate the output of MAP algorithms into a MAS that is then able to execute the solution plan, i.e. the MAP algorithms are not involved during runtime. There are at least two other surveys on multi-agent planning, they can be found in (Weerdt, Mors, and Witteveen 2005; de Weerdt and Clement 2009).

In (Mao et al. 2007), decommitment penalties and a Vickrey auction mechanism are proposed to solve a multi-agent planning problem in the context of an airport — deicing and anti-icing aircrafts during winter — where the agents are self-interested and often have conflicting interests. The experiments showed that the former ensures a fairer distribution of delay, while the latter respects the preferences of the individual agents. Both mechanisms outperformed a first come, first served mechanism, but were specifically tailored to the airport problem.

CANPLAN2 (Sardiña and Padgham 2007) is a BDI-based formal language that incorporate an HTN planning mechanism. This approach was further extended in (Sardiña and Padgham 2011) to address previous limitations such as failure handling, declarative goals, and lookahead planning. It is important to note that the CAN family are not implemented programming languages, although its features could be used to augment some BDI-based Agent Oriented Programming (AOP) languages.

The TAEMS framework (Decker 1996) provides a modelling language for describing task structures — the tasks that the agents may perform. Such structures are represented by graphs, containing goals and sub-goals that can be achieved, along with methods required to achieve them. Each agent has its own graph, and tasks can be shared between graphs, creating relationships where negotiation or coordination may be of use. Coordination in TAEMS is identified using the language's syntax, and then the developer choose or create an ad-hoc coordination mechanism by using the commitment constructs that are available. The TAEMS framework does no explicit planning, its focus is on coordinating tasks of agents where specific deadlines may be required. Its development has been discontinued since 2006.

In (Clement, Durfee, and Barrett 2007), multi-agent planning algorithms and heuristics are proposed to exploit summary information during the coordination stage in order to speed up planning. The key idea is to annotate each abstract operator with summary information about all of its potential needs and effects. That often resulted in an exponential reduction in planning time compared to a flat representation. This approach depends on some specific conditions and assumptions, and therefore cannot be used in all domains.

4 Combining MAP with MAS

In order to allow the JaCaMo framework to execute the solution generated by the MAP algorithm, we define a grammar for MAP-POP and a set of algorithms for a translator. The translator is used to help bridge the planning and execution stages of multi-agent planning problems. Off-line MAP algorithms usually ignore the execution stage of planning, ending up with just a set of plans that has to be implemented by the user. On the other hand, we have AOP languages and MAS development frameworks that usually have some kind of planning capabilities available during runtime (online), but provide no sophisticated way to solve complex multi-agent planning problems.

The translator needs as input the definition of a multi-agent planning problem and the solution for the problem found by a MAP algorithm. It then provides as output a MAS specified in JaCaMo that is able to execute the solution found during the planning stage. If the MAP algorithm being used during the planning stage also supports single-agent planning, then the translator should still be able to provide a valid output, but it will not use all of the abstraction levels that JaCaMo provides, such as Moise organisations, which are not necessary in single-agent systems.

A standard input would be ideal for the translation process, but in this case it means that we would need to change the source code of the MAP algorithms. If we develop a standard input, each new algorithm would need to be adapted to accept this new input, while if we choose to use the inputs of

the MAP algorithms, we then have to adapt the grammar and algorithms of the translator to accept them. Unfortunately, at the time of writing there is no standard formalism for the representation of multi-agent planning problems that is accepted by the MAP community, therefore we chose to adapt the translator to accept multiple inputs.

The input depends on which MAP algorithm is used, as each multi-agent planner usually makes its own adaptations to a planning problem formalism. The MAP-POP algorithm uses its own extension of PDDL 3.1 for multi-agent planning. We use this input from the PDDL files of the MAP-POP algorithm to define the name of the agents in the JaCaMo project file and the roles in the Moise organisation. We use the PDDL problem file to build CArtAgO artefacts that represent the initial state of the environment.

The output of MAP algorithms consists of a solution that solves the global goal of the problem. The MAP-POP algorithm requires coordination constraints alongside the actions in order to establish the partial order in which the actions should be executed. This resulted in MAP-POP providing a solution that allows the organisation in Moise to use the coordination constraints in order to construct a MAS with parallel execution of plans.

Both the input and output of the MAP algorithm are given as input for the translator. The translator then generates a MAS specified in JaCaMo, containing: JaCaMo project file, Jason agent's files, Moise XML specifications, and Java codes for the CArtAgO artefacts. This standard output provides generic classes that can be used to integrate new MAP algorithms. In order to integrate new MAP algorithms, one would have to develop input and solution grammars (similar to what we made for MAP-POP), and simply adapt our translation algorithms accordingly.

A summary of how the translator works is available in the diagram of Figure 1. The solution provided by the MAP algorithm is translated into AgentSpeak plans, and added to the respective agent's plan library in Jason. Because plan representation and action theory in Jason differs from the basics of the planning formalisms used by the MAP algorithms (STRIPS-like), we had to use simple transitions, that is, every action in the solution would translate to a plan in Jason with the preconditions at the context, and the effects of the action at the body of the plan. After executing the action, the effects will change the environment, i.e. the CArtAgO artefacts.

Due to space constraints, in this paper we show only the grammar of the problem file for MAP-POP. For the full grammar, all the algorithms, the domain, problem, solution, and resulting MAS of the case studies present in the next section check `http://bit.ly/1DFqveG`.

In Listing 1, we present a simplified BNF grammar based on the official PDDL 3.1 definition, which can be found in `http://bit.ly/1BRcTC8`. Each single quote pair encloses a string that is expected to appear in the file, brackets are optional, and the rest are non-terminal symbols. For example the non-terminal symbol `name` represents a terminal string of characters $a..z|A..Z$.

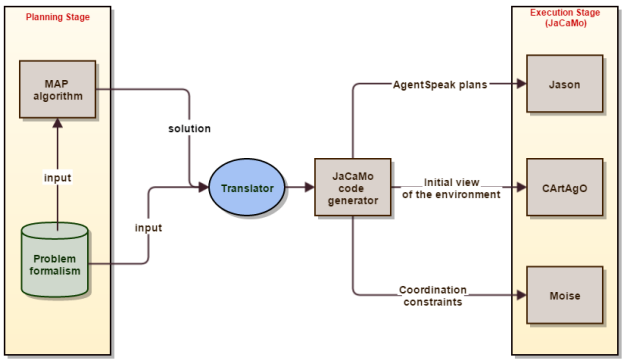

Figure 1: An overview of the translation process.

Listing 1: Initial lines from the grammar for the problem file.

```
problem ::= '((define (problem' name ')'
              '(:domain name ')'
               objectsDef
               [sharedData]
               initDef
               globalGoals ')' ;

objectsDef ::= '(:objects' typedList+ ')' ;

sharedData ::= '(:shared-data' pf+ '- (
    either' name+ ') )' ;

pf ::= predicate | func ;

initDef ::= '(:init' literal* ')' ;

literal ::= term | '(not' term ')' ;

term ::= ( '(' litName first* name* ')' ) |
    ( '(= (' litName first ')' name ')' ) ;

litName ::= name ;

first ::= name ;

globalGoals ::='(:global-goal (and' literal*
     ') )' ;
```

Similarly, for the translation we show only the main translation algorithm in Algorithm 1. The translation function receives as parameters the information contained in the domain, problem, and solution files, which are in accordance with their respective grammar. For example, the notation in `DomainSpec.domain.typesDef.typedList` means that we look in the domain information and inside typesDef for any typedList, as specified in the domain grammar. The translation starts by getting the agent types from the PDDL domain file, and the agents names from the PDDL problem file. With this information it then calls the rest of the algorithms, starting with the algorithm for the translation of the organisation, the algorithm for the plans in Jason, and finally returning and calling the algorithm for the CArtAgO artefacts.

To demonstrate part of the translation process consider

Algorithm 1 Main translation algorithm.

```
1: function TRANSLATE(DomainSpec, ProblemSpec, SolutionSpec)
2:     for (n:name, t:type) in DomainSpec.domain.typesDef.typedList do
3:         if t = 'agent' then
4:             agentsTypes ← agentsTypes ∪ n
5:         end if
6:     end for
7:     for t1 in agentsTypes do
8:         for (n:name, t2:type) in ProblemSpec.problem.objectsDef.typedList do
9:             if t1 = t2 then
10:                agents ← agents ∪ (n, t2)
11:            end if
12:        end for
13:    end for
14:    organisation ← organisation ∪ createOrg(SolutionSpec, agentsTypes, agents)
15:    agentCode ← agentCode ∪ createAgentCode(SolutionSpec, agents)
16:    artefacts ← artefacts ∪ createEnv(DomainSpec, ProblemSpec)
17:    return (agents ∪ organisation ∪ agentCode ∪ artefacts)
18: end function
```

Listing 2 and Listing 3, a step (action) from the solution and its translation to a plan in Jason. The parameters from the step of the solution are used in the context of the resulting plan in Jason — these parameters are used to access and update the artefacts, and are also used to check preconditions. The context (note that the context of a plan starts after the colon) contains the information to access the necessary artefacts, all subsequent lines are each a precondition specified in that step of the solution. Preconditions that involves only predicates pertaining the agent that is responsible for executing that plan can be checked directly in that agent's belief base. The remaining preconditions access the artefacts and make the necessary tests.

Finally, at the body of a Jason plan (the body starts after the left arrow), the effects of the step are translated into Jason actions. The translation can generate two types of actions: an action that can change the belief base of the agent that is running that action — this happens if the predicate in question involves only that same agent; or an action can result in a change in the environment — i.e. an update to observable properties of the artefacts that represent the environment.

A simple print mechanism is added using the syntax for detecting plan failure in Jason, `-!`, that provides basic feedback on which plans failed. If a plan fails and it has any subsequent dependent plans in the Moise organisation schema, the organisation will prevent the execution of those plans as the previous goals were not achieved. If there were no errors during the translation process, then these plans should never fail. However, they may fail because of two different reasons: new plans were added or translated plans were edited by the developer; or there may be other agents that may cause some kind of interference during execution, resulting in plan failure. Regardless, the mechanism for handling plan failure is present only to inform the user of the failure, it is not possible for the translator to call for replanning mechanisms as the MAP algorithms do not have any kind of interaction with the execution stage, this is part of future work.

Listing 2: A step from the solution of a driverlog problem.

```
3 // step id
driver2 // agent executing this step
Action: board
Parameters: driver2 truck1 street0
Precond:
pos truck1
street0

at driver2
street0

empty truck1
true
Effect:
at driver2
truck1

empty truck1
false
```

Listing 3: A Jason translated plan for a driverlog problem.

```
+!board1: V1 = ``truck1'' & V2 = ``street0''
     & id(V1,Id1) & id(V2,Id2) & at(V2) &
    pos(L)[artifact_id(Id1)] & processList(L
    ,V2) & empty(E)[artifact_id(Id1)] & E
          <- -at(V2);
             +at(V1);
             updateEmpty(false)[artifact_id(
    Id1)].
-!board1 <- .print(``Plan board1 failed,
    check solution plan.'').
```

The roles of the organisation are acquired from the instances of the formalism used to represent the problem, which in this case with MAP-POP are the PDDL files. By checking for agent types in Listing 4, and then checking the objects that use those types in Listing 5, the translator generates the roles present in Listing 6. The coordination constraints from the solution found by the MAP-POP algorithm are instantiated in a Moise specification file as a new schema to be followed by the agents. The plans for adopting this schema are also added to each agent's plan library. The conversion of coordination constraints into schemas is exemplified in the next section, along with the descriptions of the driverlog do main and problem that were used as examples.

Listing 4: Types of the driverlog domain.

```
(:types location truck obj - object
        driver - agent)
```

Listing 5: Objects from the problem file that use types in the driverlog domain.

```
(:objects
   driver1 driver2 - driver
   truck1 truck2 - truck
   package1 package2 - obj
   s0 s1 s2 p1-0 p1-2 - location
)
```

Listing 6: Example of translated PDDL types into Moise roles.

```
<role-definitions>
   <role id="driver" />
      <role id="driver1"> <extends role="
    driver"/> </role>
      <role id="driver2"> <extends role="
    driver"/> </role>
   </role-definitions>
```

Now for the environment, we obtain the names of the artefacts by looking at the objects that are not of type agent in the problem file, for example, in Listing 5 they are truck, obj, and location. Next, we create one artefact for each of these types. Information about these objects are stored in observable properties — when an agent focuses an artefact, the observable properties of that artefact will be directly represented as beliefs in that agent's belief base. In Listing 7 the truck object has two observable properties, `empty` (whether or not there are packages inside) and `pos` (where the truck is). For each observable property the artefact also has an operation that allows agents to execute it as an action in order to update its value.

Listing 7: Example of a CArtAgO artefact representing a truck object from the driverlog domain.

```
  defineObsProperty("empty");
  defineObsProperty("pos");

  @OPERATION public void updateEmpty(Boolean
     newEmpty) {
    ObsProperty opEmpty = getObsProperty("
    empty");
    opEmpty.updateValue(newEmpty);
```

At the end of this process all files necessary for the execution stage are available and the user can run the system as any normal JaCaMo system, by running the MAS project file that was also generated during the translation.

5 Case Studies

In this section we describe two multi-agent adaptations of single-agent planning problems from previous IPCs: the driverlog domain and the depots domain. We also discuss the solution and coordination constraints found by the MAP-POP algorithm and the output of the translation process, that is, the MAS that was generated as output.

Performance is not an issue discuss here since there is no purpose in benchmarking the translation, as the planning stage is separated from the execution stage. Instead, we focus on a more qualitative evaluation, analysing the input and output during the planning and execution stages.

Driverlog Domain

The Driverlog domain is a simple problem of logistics. There are several streets and passageways that may contain packages, trucks, and drivers. A driver cannot directly walk through streets, it can only walk through passageways that have paths between a street and a passageway. When driving a truck, a driver can then drive through streets that are linked with each other.

In this domain we only have one type of agent, the driver, as it is the only object that can perform actions. The agent can perform the following actions:

- **load truck**: loads a package from a location into a truck;
- **unload truck**: unloads a package from a truck into a location;
- **board truck**: the driver enters the truck at a location;
- **disembark truck**: the driver leaves the truck at a location;
- **drive truck**: the driver drives the truck from a street to another;
- **walk**: the driver walks from a location that contains a path to another location.

The initial state of the problem can be observed in Figure 2. All the streets are linked, but note that only a truck can move through the linked streets. The drivers, when not driving a truck, can only move through passageways that have paths to streets. The global goal is to have driver1 at s1, and t1 at s1. That is, driver1 and truck1 should be at street1.

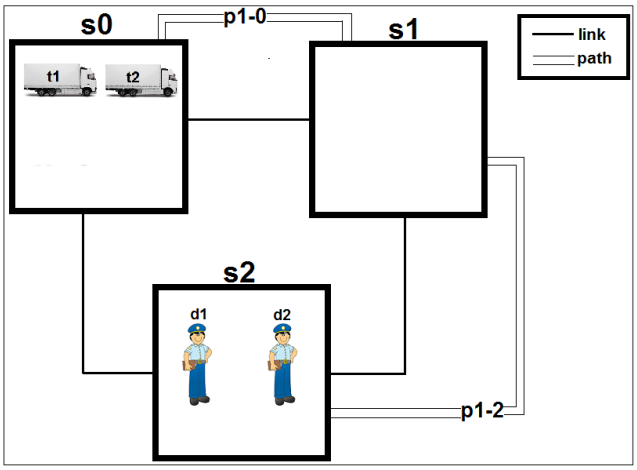

Figure 2: Initial state of the problem for the Driverlog domain.

The solution found by MAP-POP for the Driverlog problem contained the following steps:

- Id 0 — Initial Step
- Id 1 — Final Step
- Id 2 — agent driver2: drive t1 to s1
- Id 3 — agent driver2: board t1 at s0

- Id 4 — agent driver2: walk from p1-0 to s0
- Id 5 — agent driver2: walk from s1 to p1-0
- Id 6 — agent driver2: walk from p1-2 to s1
- Id 7 — agent driver2: walk from s2 to p1-2
- Id 8 — agent driver1: walk from p1-2 to s1
- Id 9 — agent driver1: walk from s2 to p1-2

The partial order in which these steps need to be executed can be obtained from the ordering constraints, also provided in the solution. The ordering constraints are represented in pairs of Ids, the first Id is the step that must come before the second, e.g. 0 — 1 means that the step with Id 0 must come before the step with Id 1. We can also use this order to set the plan operators, i.e. if it will be executed in parallel or sequentially, in the Moise schema. The execution order for the solution of this problem is (numbers between commas can be executed in parallel): 0 — 7,9 — 6,8 — 5 — 4 — 3 — 2 — 1.

Depots Domain

The Depots domain is more complex than the previous domain, as it involves different types of agents. In this domain trucks are used to transport crates between warehouses, with the help of hoists that are present in each warehouse.

There are two types of agents: trucks and locations. Note here that a location (depots or distributors) is a type of agent, since each location has control over a hoist. A truck can perform the following actions:

- **drive**: move the truck from a place to another;
- **load**: loads a crate that a hoist has into the truck;
- **unload**: unloads a crate from the truck to a hoist.

A location can perform the following control actions with its hoist:

- **liftP**: lifts a crate that is on top of a pallet;
- **liftC**: lifts a crate that is on top of another crate;
- **dropP**: drops a crate on top of a pallet;
- **dropC**: drops a crate on top of another crate.

The initial state of the problem can be observed in Figure 3. Truck `t1` is located at `depot0`, and truck `t2` is located at `distributor1`. A truck agent is able to move freely between any of the locations. The global goal is to have c0 on p2, and c1 on p1, i.e. crate0 must be moved to distributor1 and crate1 must be moved to distributor0.

Next we have the solution found by MAP-POP for the Depots domain:

- Id 0 — Initial Step
- Id 1 — Final Step
- Id 2 — agent ditributor1: drop c0 on p2 at distributor1
- Id 3 — agent distributor0: drop c1 on p1 at distributor0
- Id 4 — agent distributor0: lift c0 from p1 at distributor0
- Id 5 — agent truck2: unload c0 to h2 at distributor1
- Id 6 — agent truck2: load c0 from h1 at distributor0

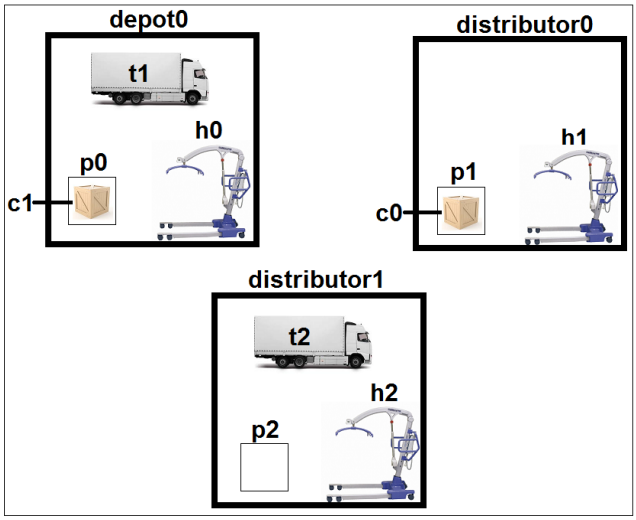

Figure 3: Initial state of the problem for the Depots domain.

- Id 7 — agent truck1: unload c1 to h1 at distributor0
- Id 8 — agent truck1: load c1 from h0 at depot0
- Id 9 — agent truck2: drive from distributor0 to distributor1
- Id 10 — agent depot0: lift c1 from p0 at depot0
- Id 11 — agent truck1: drive from depot0 to distributor0
- Id 12 — agent truck2: drive from distributor1 to distributor0

Once again, we find the partial order of actions by retracing all the ordering constraints, resulting in the order: 0 — 4,10,12 — 6,8 — 9,11 — 5,7 — 2,3 — 1.

Translation

The translator extracts from the solution the steps and the ordering constraints. Each agent directly represents a role in the Moise organisation, e.g. objects driver1 and driver2 are translated as roles that extend a driver role in the Moise specification file under the structural specification. For future work we expect to implement, for example, only the role of driver with a maximum cardinality of 2. In the Moise functional specification we translate steps into goals, with the plan operators (sequence or parallel) that were extracted from the ordering constraints.

Every role (agent) has its mission, and that mission contains all the goals that need to be executed by that particular role. Links and formation constraints, two Moise features, are not considered in our translation algorithms, but they could be expressed by making a few adaptations in the planning formalism. However, since we are using the default input of the MAP algorithms we chose not to make use of these features.

As for the environment, the translator checks the initial state provided by the input of the MAP algorithms. If one of the variables in an initial state is an agent, then that state will be represented as a belief in that particular agent's belief

base. If not, then it will be stored as an observable property in its respective artefact.

The information about initial states is also used to instantiate initial values for the observable properties, that are defined by the predicates and functions of the problem domain. An artefact is created for each type declared as an object in the domain file, to represent the initial state of the environment. For the driverlog domain we have artefacts for `location`, `truck`, and `obj`. For the depots domain we have artefacts for `hoist` and `surface`. When an agent executes the operation of an artefact, it updates the observable properties of the artefact that is involved by using the effects of that particular action.

For the Jason plans, each step is converted to a plan that is added to the agent's plan library, with that step's respective preconditions and effects. In the end of this process we obtain a MAS that can execute the solution for a driverlog problem and a MAS for a depots problem.

6 Conclusion

We integrated a multi-agent planner into JaCaMo through the use of a translator. JaCaMo provided practical solutions for some of the problems that appeared in the execution stage, such as the coordination of agents using Moise organisations, representation of the environment with CArtAgO artefacts, and execution of the solution using Jason agents.

The execution stage of planning is often overlooked when dealing with off-line planning. Our work tries to bridge this gap by using translation algorithms to create a MAS, using the input and output of a multi-agent planner. Our goal with this work was to provide the developers with an initial multi-agent system implementation for a target scenario, based on the solutions found by the multi-agent planner, and to provide a basis for extending other MAP algorithms to work with JaCaMo.

We are investigating two other possible choices of MAP algorithms to be integrated with JaCaMo, the Planning-First (Nissim, Brafman, and Domshlak 2010) and the MAD-A* (Nissim and Brafman 2012). Planning-First is a general, distributed multi-agent planning algorithm that uses Distributed Constraint Satisfaction Problem to coordinate the agents. The MAD-A* is an adaptation for multi-agent planning of one of the best known heuristic search algorithm, A*.

During our work we identified a few downsides:

- although the translation can be used to fill the gap between planning and execution stages, it is not a seamless transition such as the one present in online planning;
- plans in Jason are different from the PDDL formalism used by the three MAP algorithms, which resulted in a simplified conversion of steps to plans;
- the performance was strictly dependent on the performance of the MAP algorithm used during the planning stage.

For future work we would like to use JaCaMo agents not only during the execution stage, but also during the planning stage, which would allow most of the translation to be done directly, and make the transition between planning and execution stages much more seamless. For example, an HTN planner would be able to provide agents in Jason with much more robust plans than previously, and also allows it to make use of current plans present in the agent's plan library prior to the planning stage. This may lead to performance gains and possibly some kind of planning and/or replanning during runtime.

Another line for future work includes the standardisation of input used by the algorithms, so that the translator accepts a standard input file. This input could be a completely new formalism or, for example, the PDDL 3.1 Multi-Agent extension introduced in (Kovacs 2012). This extension allows planning for agents in temporal, numeric domains and copes with many of the already discussed open problems in multi-agent planning, such as the exponential increase in the number of actions, but it also approaches new problems such as the constructive and destructive synergies of concurrent actions. This would also make the process of including a new MAP algorithm easier and at the same time promote a standard formalism to represent domains and problems in multi-agent planning, which at the time of writing does not exist.

Finally, we would also like to test our work on real world applications, such as robotics. Specifically, the scenario we have in mind is the use of Unmanned Aerial Vehicles (UAVs) to monitor, control, and mitigate flash flood occasioned by heavy rain when associated with severe thunderstorms. The planner could be used to generate possible trajectories in flash flood locations, while JaCaMo is used to coordinate the UAVs and reason about possible courses of actions.

7 Acknowledgments

We are grateful for the support given by CAPES (grant number 23038.006826/2014-64) 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.; Wooldridge, M.; and Hübner, J. F. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons.

Braubach, L.; Pokahr, A.; and Lamersdorf, W. 2008. A universal criteria catalog for evaluation of heterogeneous agent development artifacts. In Jung, B.; Michel, F.; Ricci, A.; and Petta, P., eds., *From Agent Theory to Agent Implementation (AT2AI-6)*, 19–28.

Busetta, P.; Ronnquist, R.; Hodgson, A.; and Lucas, A. 1999. JACK Intelligent Agents - Components for Intelligent Agents in Java. AgentLink News, Issue 2.

Cardoso, R. C.; Zatelli, M. R.; Hübner, J. F.; and Bordini, R. H. 2013. Towards Benchmarking Actor- and Agent-Based Programming Languages. In *AGERE! @ SPLASH 2013*.

Cardoso, R. C.; Hübner, J. F.; and Bordini, R. H. 2013. Benchmarking Communication in Agent- and Actor-Based Languages. In *Proceedings of the EMAS '13, held with AAMAS-2013*, 81–96.

Clement, B. J.; Durfee, E. H.; and Barrett, A. C. 2007. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research (JAIR)* 28:453–515.

Crosby, M.; Jonsson, A.; and Rovatsos, M. 2014. A single-agent approach to multiagent planning. In *21st European Conf. on Artificial Intelligence (ECAI'14)*.

Dastani, M. 2008. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16(3):214–248.

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

Decker, K. 1996. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. *Foundations of Distributed Artificial Intelligence, Chapter 16* 429–448.

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

Hindriks, K. V.; de Boer, F. S.; van der Hoek, W.; and Meyer, J.-J. C. 2000. Agent Programming with Declarative Goals. In *Proceedings of the 7th International Workshop on Agent Theories, Architectures, Boston, MA, USA*, 228–243. Springer.

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.

Jonsson, A., and Rovatsos, M. 2011. Scaling Up Multiagent Planning: A Best-Response Approach. In *Procs. ICAPS 2011*, 114–121. AAAI Press.

Jordan, H.; Botterweck, G.; Huget, M.-P.; and Collier, R. 2011. A feature model of actor, agent, and object programming languages. In *Proceedings of the SPLASH '11 Workshops*, 147–158. New York, NY, USA: ACM.

Jr., J. C. B., and Durfee, E. H. 2011. Distributed algorithms for solving the multiagent temporal decoupling problem. In *AAMAS 2011, Taipei, Taiwan*, 141–148.

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, 19–27.

Lerma, A. T. 2011. Design and implementation of a Multi-Agent Planning system. Master's thesis, Polytechnic University of Valencia, Valencia, Spain.

Mao, X.; Mors, A.; Roos, N.; and Witteveen, C. 2007. Coordinating Competitive Agents in Dynamic Airport Resource Scheduling. In *Proceedings of the 5th German conference on Multiagent System Technologies*, 133–144.

Meneguzzi, F., and De Silva, L. 2013. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review* FirstView:1–44.

Nissim, R., and Brafman, R. I. 2012. Multi-Agent A* for Parallel and Distributed Systems. In *Proceedings of the Heuristics for Domain-Independent Planning Workshop, held with ICAPS 12*.

Nissim, R.; Brafman, R. I.; and Domshlak, C. 2010. A General, Fully Distributed Multi-Agent Planning Algorithm. In *AAMAS 10*, 1323–1330.

Planken, L.; de Weerdt, M.; and Witteveen, C. 2010. Optimal temporal decoupling in multiagent systems. In *AAMAS 2010, Toronto, Canada*, 789–796.

Rao, A. S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, MAAMAW '96, 42–55.

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. Springer. chapter 8, 259–288.

Ricci, A. 2014. From actor event-loop to agent control-loop: Impact on programming. In *AGERE! '14*, 121–132. New York, NY, USA: ACM.

Sapena, O.; Onaindia, E.; and Torreño, A. 2015. FLAP: applying least-commitment in forward-chaining planning. *AI Commun.* 28(1):5–20.

Sardiña, S., and Padgham, L. 2007. Goals in the context of BDI plan failure and planning. In *AAMAS 2007, Honolulu, Hawaii, USA*.

Sardiña, S., and Padgham, L. 2011. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems* 23(1):18–70.

Torreño, A.; Onaindia, E.; and Sapena, O. 2014a. A flexible coupling approach to multi-agent planning under incomplete information. *Knowl. Inf. Syst.* 38(1):141–178.

Torreño, A.; Onaindia, E.; and Sapena, O. 2014b. FMAP: distributed cooperative multi-agent planning. *Appl. Intell.* 41(2):606–626.

van Leeuwen, P., and Witteveen, C. 2009. Temporal decoupling and determining resource needs of autonomous agents in the airport turnaround process. In *Proceedings of the International Conference on Intelligent Agent Technology, IAT 2009, Milan, Italy*, 185–192.

Weerdt, M. D.; Mors, A. T.; and Witteveen, C. 2005. Multi-agent planning: An introduction to planning and coordination. Technical report, Handouts of the European Agent Summer.

Witwicki, S. J., and Durfee, E. H. 2011. Towards a unifying characterization for quantifying weak coupling in dec-POMDPs. In *AAMAS 2011, Taipei, Taiwan*, 29–36.

Wooldridge, M. 2002. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 1st edition.