

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**INTEGRATING AUTOMATED PLANNING
WITH A MULTI-AGENT SYSTEM
DEVELOPMENT FRAMEWORK**

RAFAEL CAUÊ CARDOSO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Supervisor: Prof. Dr. Rafael Heitor Bordini

**Porto Alegre
2014**

C268i Cardoso, Rafael Cauê

Integrating automated planning with a multi-agent system development framework / Rafael Cauê Cardoso. - Porto Alegre, 2014. 98 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Rafael Heitor Bordini.

1. Informática. 2. Sistemas Multiagentes. 3. Agente Inteligente (Software). I. Bordini, Rafael Heitor. II. Título.

CDD 006.3

**Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS**

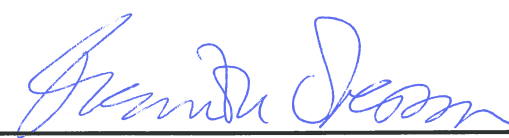


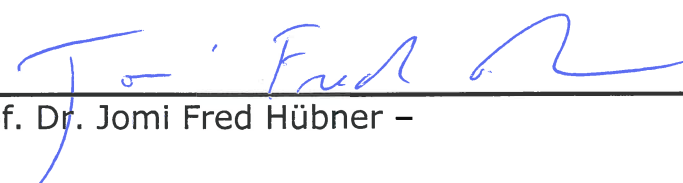
TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "*Integrating Automated Planning With a Multi-Agent System Development Framework*" apresentada por Rafael Cauê Cardoso como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 21/03/2014 pela Comissão Examinadora:



Prof. Dr. Rafael Heitor Bordini – PPGCC/PUCRS
Orientador


Prof. Dr. Felipe Rech Meneguzzi – PPGCC/PUCRS


Profa. Dra. Renata Vieira – PPGCC/PUCRS


Prof. Dr. Jomi Fred Hübner – UFSC

Homologada em 11/03/2015, conforme Ata No. 001 pela Comissão Coordenadora.


Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS

Campus Central

Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900

Fone: (51) 3320-3611 – Fax (51) 3320-3621

E-mail: ppgcc@pucrs.br

www.pucrs.br/facin/pos

INTEGRANDO PLANEJAMENTO AUTOMATIZADO COM UM FRAMEWORK DE DESENVOLVIMENTO DE SISTEMAS MULTI-AGENTE

RESUMO

Planejamento automatizado é uma capacidade importante de se ter em agentes inteligentes. Já foi realizada uma extensa pesquisa em planejamento para um único agente, porém planejamento multiagente ainda não foi totalmente explorado, principalmente por causa do alto custo computacional encontrado nos algoritmos para planejamento multiagente. Com o aumento na disponibilidade e o avanço tecnológico de sistemas distribuídos, e mais recentemente de processadores multinúcleos, novos algoritmos de planejamento multiagente tem sido desenvolvidos, como por exemplo o algoritmo MAP-POP, que neste trabalho é integrado com o *framework* de sistemas multiagente JaCaMo. Este trabalho fornece capacidades para planejamento multiagente *offline* como parte de um *framework* para desenvolvimento de sistemas multiagentes. Esse *framework* suporta problemas multiagente complexos baseados em programação orientada a agentes. Em síntese, a principal contribuição deste trabalho é fornecer aos desenvolvedores uma implementação inicial do sistema multiagente para um determinado cenário, baseado nas soluções encontradas pelo planejador MAP-POP, a qual pode ainda ser expandida pelo desenvolvedor para se tornar um sistema multiagente completo e bem desenvolvido.

Palavras Chave: agentes inteligentes; programação orientada a agentes; planejamento multiagente; sistemas multiagentes; JaCaMo.

INTEGRATING AUTOMATED PLANNING WITH A MULTI-AGENT SYSTEM DEVELOPMENT FRAMEWORK

ABSTRACT

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

Keywords: intelligent agents; agent-oriented programming; multi-agent planning; multi-agent systems; JaCaMo.

LIST OF FIGURES

Figure 2.1	Diagram representing a simple agent architecture.	25
Figure 2.2	A more complex representation of an agent architecture [75].	26
Figure 2.3	Generic BDI model, adapted from [88].	27
Figure 2.4	Overview of a JaCaMo MAS, highlighting its three dimensions [9].	30
Figure 2.5	Typical overview of a planner [89].	33
Figure 2.6	Initial state of the Blocks World problem of Listing 2.3.	36
Figure 2.7	Goal state of the Blocks World problem of Listing 2.3.	36
Figure 3.1	Stages of the MAP-POP algorithm [50].	46
Figure 3.2	An overview of the translation process.	49
Figure 4.1	Initial state of our problem for the Driverlog domain.	60
Figure 4.2	Initial state of our problem for the Depots domain.	62

LIST OF ALGORITHMS

Algorithm 3.1 Main translation algorithm.	52
Algorithm 3.2 Organisational specification translation algorithm.	52
Algorithm 3.3 Structural specification translation algorithm.	53
Algorithm 3.4 Functional specification translation algorithm.	53
Algorithm 3.5 Normative specification translation algorithm.	54
Algorithm 3.6 Agent code translation algorithm.	55
Algorithm 3.7 Environment translation algorithm.	55

LIST OF LISTINGS

2.1	Example of plans in AgentSpeak.	28
2.2	Domain file for the Blocks World example in PDDL.	34
2.3	Problem file for the Blocks World example in PDDL.	35
3.1	Simplified BNF grammar for the syntax definition of the PDDL 3.1 domain description.	49
3.2	Simplified BNF grammar for the syntax definition of the PDDL 3.1 problem description.	50
3.3	BNF grammar for solution generated by MAP-POP.	50
3.4	Main translation algorithm.	52
3.5	Organisational specification translation algorithm.	52
3.6	Structural specification translation algorithm.	53
3.7	Functional specification translation algorithm.	53
3.8	Normative specification translation algorithm.	54
3.9	Agent code translation algorithm.	55
3.10	Environment translation algorithm.	55
3.11	A step from the solution of a driverlog problem.	56
3.12	A Jason translated plan for a driverlog problem.	56
3.13	Types of the driverlog domain.	57
3.14	Objects from the problem file that use types in the driverlog domain.	57
3.15	Example of translated PDDL types into Moise roles.	57
A.1	Driverlog MAP-POP PDDL domain.	75
A.2	Driverlog MAP-POP PDDL problem.	76
A.3	Driverlog JaCaMo project file.	77
A.4	Driverlog Moise organisation.	77
A.5	Driverlog location artefact.	79
A.6	Driverlog obj artefact.	79
A.7	Driverlog truck artefact.	80
A.8	Driverlog init agent.	80
A.9	Driverlog common code.	81
A.10	Driverlog driver1 agent.	82
A.11	Driverlog driver2 agent.	83
B.1	Depots MAP-POP PDDL domain for location agent.	85
B.2	Depots MAP-POP PDDL domain for truck agent.	86
B.3	Depots MAP-POP PDDL problem.	87
B.4	Depots JaCaMo project file.	88
B.5	Depots Moise organisation.	88

B.6	Depots hoist artefact.	91
B.7	Depots surface artefact.	91
B.8	Depots init agent.	92
B.9	Depots common code.	93
B.10	Depots truck1 agent.	94
B.11	Depots truck2 agent.	95
B.12	Depots depot0 agent.	96
B.13	Depots distributor0 agent.	97
B.14	Depots distributor1 agent.	98

LIST OF ABBREVIATIONS AND ACRONYMS

AFC	<i>Asynchronous Forward Checking</i>
AFME	<i>Agent Factory Micro Edition</i>
AFSE	<i>Agent Factory Standard Edition</i>
AI	<i>Artificial Intelligence</i>
AOP	<i>Agent-Oriented Programming</i>
AOS	<i>Agent Oriented Software</i>
BDI	<i>Belief-Desire-Intention</i>
BRF	<i>Belief Revision Function</i>
DisCSP	<i>Distributed Constraint Satisfaction Problem</i>
disRPG	<i>distributed Relaxed Planning Graph</i>
GOAL	<i>Goal-Oriented Agent Language</i>
HTN	<i>Hierarchical Task Network</i>
IPC	<i>International Planning Competition</i>
MAD A*	<i>Multi-Agent Distributed A*</i>
MA-FD	<i>Multi-Agent Fast Downward</i>
MAP	<i>Multi-Agent Planning</i>
MAP-POP	<i>Multi-Agent Planning based on Partial-Order Planning</i>
MAS	<i>Multi-Agent Systems</i>
MDP	<i>Markov Decision Process</i>
PDDL	<i>Planning Domain Definition Language</i>
PRS	<i>Procedural Reasoning System</i>
STP	<i>Simple Temporal Problem</i>
STRIPS	<i>Stanford Research Institute Problem Solver</i>

CONTENTS

1. INTRODUCTION	21
1.1 Motivation	22
1.2 Objectives	22
1.3 Dissertation Outline	23
2. BACKGROUND	25
2.1 Intelligent Agents	25
2.1.1 Agent-Oriented Programming	27
2.1.2 Agent-Oriented Programming Languages	28
2.1.3 JaCaMo	30
2.2 Planning	32
2.2.1 Classical Planning	33
2.2.2 Multi-agent Planning	36
2.3 Related Work	38
3. INTEGRATING MAP INTO MAS	41
3.1 The Algorithms	41
3.1.1 Planning-First Algorithm	42
3.1.2 MAD-A* Algorithm	43
3.1.3 MAP-POP Algorithm	44
3.2 The Translation	47
4. CASE STUDIES	59
4.1 Driverlog Domain	59
4.2 Depots Domain	60
4.3 Planning Stage	62
4.4 Execution Stage	64
5. CONCLUSION	65
REFERENCES	67

A. Driverlog domain codes

75

B. Depots domain codes

85

1. INTRODUCTION

The Agents research area evolved rapidly through the mid 90s, and since then the agent community has been expecting intelligent agents to become a key technology in computer systems [42,90]. With the increasing advance of distributed systems, and more recently multi-core processors, there are evidence that both academia and industry are starting to look for concurrent and parallel programming languages. The Erlang¹ programming language, for example, is already used in many companies like Amazon, Yahoo!, and Facebook. Erlang is based on the Actor Model [2], a predecessor of the Agent Model [90]. Recently, the Actor Model has been receiving increased attention, both from academia and industry, that can be taken as a clue that the Agent Model is soon to follow.

Planning is the act or process of making or carrying out plans, thinking about the actions that are required to achieve a desired goal and organizing them; it is considered to be a characteristic of intelligent behaviour. In psychology, cognitive planning is one of the main executive functions; it encompasses the neurological processes involved in the formulation, evaluation and selection of a sequence of thoughts and actions to achieve a desired goal [68]. Thus, 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 [24,85]. Recent algorithms have managed to significantly improve performance, which was one of the main incentives for pursuing this topic.

Single-agent planning has been extensively researched over the years [4, 8, 36, 49, 59, 62, 77]. One of its more common models is classical planning, which refers to planning for restricted state-transition systems where a state is a collection of variables [62]. Although restrictive and unrealistic, classical planning serves as baseline for other models, such as temporal and probabilistic planning. One of the earliest classical planners is the STanford Research Institute Problem Solver (STRIPS) [31], it uses a model of the world and a set of action schemata that describes the preconditions and effects of all the actions available to the agent. This action formalism presented in STRIPS continues to be used in modern-day planners, for example, it is part of the Planning Domain Definition Language (PDDL) [55], a language proposed to standardise the syntax used for representing planning problems.

PDDL has been used as the standard language for the International Planning Competition (IPC)² since 1998, and has been constantly updated and extended during this period of time. The latest major version is PDDL 3.0 [34], further extended in PDDL 3.1³. PDDL is not limited to classical planning. Besides the STRIPS-like extension there are other models available such as temporal planning, and recently even a MAP extension [48] was proposed.

¹<http://www.erlang.org/>.

²<http://www.icaps-conference.org/>

³<http://ipc.informatik.uni-freiburg.de/PddlExtension>.

We use the JaCaMo framework as the MAS development platform. It 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.

In this work we provide an approach to integrate Multi-Agent Planning (MAP) algorithms with a Multi-Agent Systems (MAS) platform. We discuss three implemented MAP algorithms recently proposed by the planning community: Planning-First [64]; Multi-Agent Distributed A* (MAD A*) [63]; and Multi-Agent Planning based on Partial-Order Planning (MAP-POP) [50]. Our goal with this work is 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.

The implementation is done in the form of a translator, which will take as input the name of the MAP algorithm to be used and the chosen algorithm's related problem input. As output, the translator will generate a coordination scheme in Moise, followed by the respective agent plans in Jason to adopt the roles in the organisation, the solution that will be added to the agents plan library, and CArtaGo artefacts for both organisation control and environment representation. All of these come together to form a multi-agent system in JaCaMo.

1.1 Motivation

The 1st Workshop on Distributed and Multi-Agent Planning (DMAP) [73] was held in 2013 during the International Conference on Automated Planning and Scheduling (ICAPS), suggesting an increase in the research interest of the automated planning community towards MAP.

Although there is an increase in interest in theoretical research on multi-agent planning, as evidenced by [21, 44, 69, 87], current implemented multi-agent planning algorithms and planners are mostly application specific, such as in [54, 83]. 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 planning algorithms normally stop at the planning stage, providing a solution plan but with no means to execute that plan. By integrating JaCaMo with MAP algorithms we are able to cover both the planning and the execution stages.

1.2 Objectives

Here we present the main objectives of our work:

1. Implement a translator that receives as input the solution plan of a problem, and generates as output a multi-agent system in JaCaMo that is able to execute the solution plan.
2. Provide a grammar and translation algorithms that can be used to make the process of integrating other MAP algorithms with JaCaMo easier.
3. A qualitative evaluation of the resulting system during both stages, planning and execution.

1.3 Dissertation Outline

The rest of the dissertation is structured as follows: Chapter 2 provides the relevant background information on Intelligent Agents, the BDI model, multi-agent systems, agent-oriented programming languages (namely Jason and JaCaMo), planning (both single-agent and multi-agent planning), and finally the related work; Chapter 3 contains the description of three MAP algorithms, and the semi-formal description of the translator that was implemented; in Chapter 4 we present two case studies, and provide a brief qualitative evaluation of both planning and execution stages; finally, in Chapter 5 some final considerations and future work are discussed.

2. BACKGROUND

In this Chapter we briefly discuss some of the fundamentals of Intelligent Agents, the Procedural Reasoning System (PRS) and its foundational Belief-Desire-Intention (BDI) model, along with one of its popular representative, the AgentSpeak language. Afterwards, we present some Agent-Oriented Programming (AOP) languages, focusing on the platform that was chosen for this work, JaCaMo. Finally, some relevant concepts in planning techniques, and more importantly Multi-Agent Planning techniques, are tackled.

2.1 Intelligent Agents

According to [89] “An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives”. In other words, agents receive perceptions through sensors in the environment, and respond to these events with actions that affect the environment. Therefore, sensor data, i.e. perceptions, are the input of an agent and the actions its output, as can be seen in Figure 2.1.

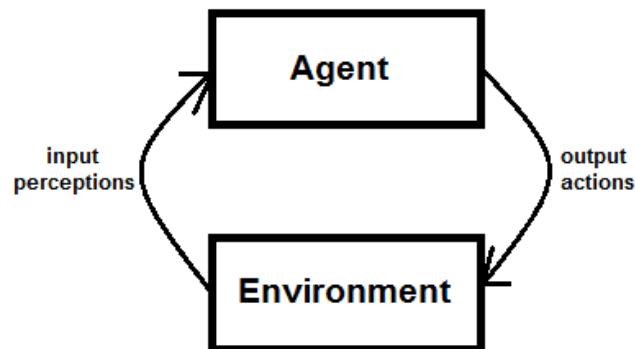


Figure 2.1: Diagram representing a simple agent architecture.

Similarly to the above concept, we have a more refined one presented in [75] that states “An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators”. Although both concepts express practically the same thing, the latter resembles the similar concept of robots, which are also known to be equipped with sensors and actuators. This concept is illustrated in Figure 2.2, where the question mark represents the reasoning that the agent performs with the input from the sensors to generate an appropriate action output.

The reasoning mechanism can be based on, for example, the PRS [32] architecture. In this architecture an agent has a library of pre-compiled plans that are composed of: goal — postcondition of the plan; context — precondition of the plan; and body — the course of action to carry out. Another important feature of the PRS is the intention stack, as it contains all of

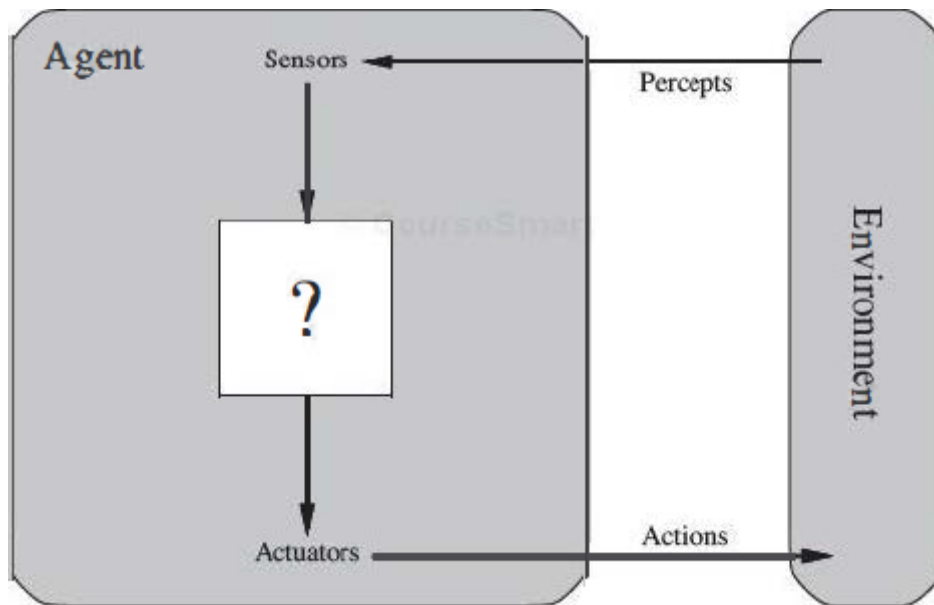


Figure 2.2: A more complex representation of an agent architecture [75].

the goals not yet achieved, and it is used by the agent to search its library to see what plans match the goal on the top of the stack as their postcondition. After that it is time to check if the precondition is satisfied, and if so then these plans become possible options to be executed by the agent.

The PRS was largely based in the BDI model, first described in [12] and further extended by [72]. The BDI model has its roots in philosophy, as it tries to understand the process of deciding which action to perform in order to achieve certain goals — also known as practical reasoning. This model has three primary mental attitudes: belief — what the agent believes that is true about its environment, and about the other agents present in it; desire — the desired states that the agent hopes to achieve; and intention — a sequence of actions that an agent has to carry out in order to achieve a state. Those mental attitudes respectively represent the information, motivational, and deliberative states of the agent. Figure 2.3 illustrate how the BDI model works: the Belief Revision Function (BRF) receives the input information from the sensors, and updates the belief base. This update will generate more options that can become current desires based on the belief and intention bases. The filter is responsible for updating the intentions base, taking into consideration its previous state, and the current belief and desire bases. Finally an intention is chosen that will be carried out as an action by the agent.

Systems that require the use of the Agent Model will seldom need only a single-agent. Albeit obvious, a MAS then has multiple agents, or as more formally defined in [89] “Multiagent systems are systems composed of multiple interacting computing elements, known as agents”.

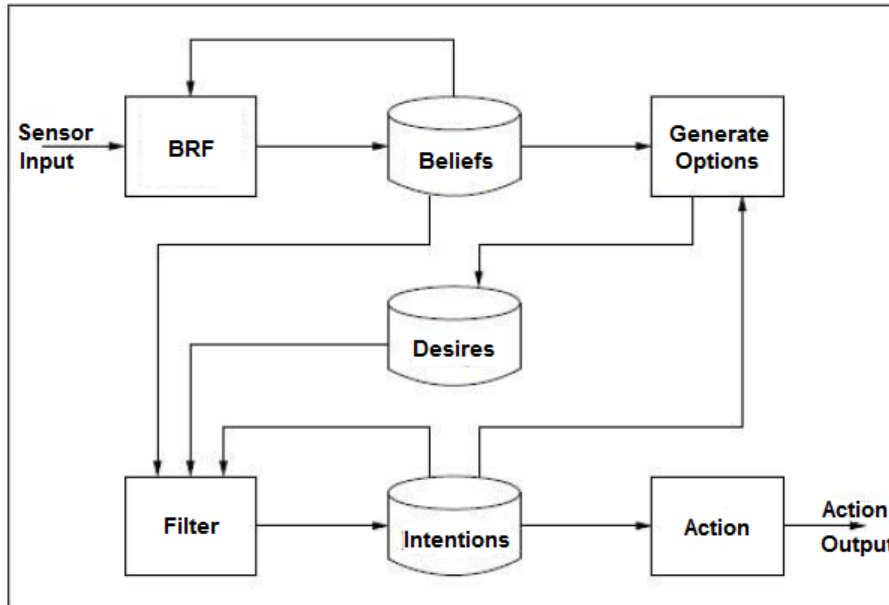


Figure 2.3: Generic BDI model, adapted from [88].

2.1.1 Agent-Oriented Programming

As previously discussed, MAS is recently obtaining more popularity, and it can already be found in a wide variety of applications, as for example in health care [5, 14, 46], smart grids [29, 47], smart homes [1, 27, 41, 60], smart cities [20, 53], among others.

Based on the BDI model, many AOP languages were created, including the AgentSpeak language. Initially conceived by Rao [71], it was later much extended in a series of publications by Bordini, Hübner, and colleagues, so as to make it suitable as a practical agent programming language known as Jason. AgentSpeak represents an abstraction of implemented BDI systems like PRS which allows agent programs to be written and interpreted much like horn-clause logic programs, hence the resemblance to the logic programming language Prolog. A plan in AgentSpeak is triggered when events (internal or external) occur, followed by checking if the context of the plan is applicable; the rest of the plan is composed of basic actions and/or subgoals that need to be achieved in order for the plan to be successful. The following is from AgentSpeak syntax: **!** for achievement goal, **?** for test goal, **;** for sequencing and **<-** to separate the plan context and its body. An achievement goal, for example **!g(t)**, is used when the agent wants to achieve a state where **g(t)** is a belief. A test goal, for example **?g(t)**, is used when the agent wants to test if **g(t)** is in the belief base. The convention regarding case sensitivity is similar as the one used in Prolog: variables start in upper-case and constants in lower-case.

The program for an agent is basically a set of initial goals, beliefs, and plans. While beliefs and goals are represented by predicates (the latter, preceded by **!**), plans have the form **te : ct <- b**, where **te** is a trigger event the plan can react to, **c** is a context (a formula that must be true for the plan to be applicable), and **b** is a sequence of actions and subgoals.

We present a simple example of AgentSpeak plans in Listing 2.1. In this example, we have an agent that can book tickets to a movie for the user. The plan `+movie(M,T)` is activated when the agent receives a perception that a new movie `M` is available in a movie theatre `T`; the perception could be received either by another agent or a newsfeed. When activated, it checks its context for a couple of preconditions: if the movie theatre is located close by to where the user lives, and if the rating of the movie is higher than seven. The rating could be obtained by agents from friends or from a website that ranks movies such as IMDb. If the preconditions are all true, then the agent proceeds to execute the body of the plan; the action `suggest(M,T)` is used to suggest the movie to the user, if they accept the plan continues, otherwise it fails and the agent stops pursuing it. As the plan continues, the agent will now have a new goal of booking tickets for the user to go watch the movie, this means that the plan `+!book_tickets(M,T)` will be activated by the internal event generated by the adoption of a subgoal. The agent checks in the context of the plan for the website `W` of the particular movie theatre that will screen the movie, the agent then proceeds to execute the body of the plan, with actions to check when the movie will screen, checking it against the user’s schedule, suggesting times and seats to the user, and finally booking the ticket by choosing a seat.

Listing 2.1: Example of plans in AgentSpeak.

```

1 +movie(M,T) : closeby(T) & rating(M,R) & R > 7
2             <- suggest(M,T);
3             !book_tickets(M,T).
4
5 +!book_tickets(M,T) : website(T,W)
6                     <- getSchedule(W,M,S);
7                     checkSchedule(S);
8                     ...;
9                     choose_seat(M,T,C).

```

2.1.2 Agent-Oriented Programming Languages

This section presents a collection of brief descriptions of some AOP languages that are all free software, with the exception of JACK. The 2APL¹ [22] — A Practical Agent Programming Language — is known for its abstraction into two different levels: the multi-agent level — which provides programming constructs for a MAS in terms of a set of individual agents and a set of environments in which actions can be performed; and the individual agent level — which provides programming constructs to implement cognitive agents based on the BDI model. It supports the implementation of both reactive and pro-active agents.

The Agent Factory Framework² [65] is a collection of tools, platforms, and languages that

¹<http://apapl.sourceforge.net/>.

²<http://www.agentfactory.com/>.

support the development and deployment of MAS. It is essentially split into two parts: the Agent Factory Standard Edition (AFSE), which deploys agents on laptops, desktops and servers; and the Agent Factory Micro Edition (AFME), which deploys agents on constrained devices such as mobile phones and sensors.

Goal-Oriented Agent Language (GOAL)³ [37] is partly based on the BDI model, but it is also influenced by the UNITY [19] language. In UNITY a set of actions executed in parallel constitutes a program, however whereas UNITY is based on variable assignment, GOAL uses more complex notions such as beliefs, goals, and agent capabilities. Message passing is based on mailboxes, each message that arrives is inserted as a fact in the receiver agent message base. GOAL's agent communication is based on speech acts, represented by moods: indicative, declarative, and interrogative.

JACK⁴ [15] is developed by the Agent Oriented Software (AOS) company and is a commercial AOP language that is based on the BDI model. One of the most successful industry AOP language, it is written in Java, which means it has natural portability. JACK's downside is that it uses mainly one thread, for security purposes.

Jadex⁵ [70] is another AOP language based on the BDI model, which allows the programming of intelligent software agents in XML and Java. It also provides a framework including a runtime infrastructure for agents, the agent platform, and an extensive runtime tool suite.

Several studies indicate that Jason has an excellent performance when compared with other AOP languages. For example, Jason is included in a qualitative comparison of features alongside with Erlang, and Java [43]; in a universal criteria catalog for agent development artefacts [13]; in a quantitative analysis of 2APL, GOAL, and Jason regarding their similarity and the time it takes them to reach specific states [7]; a performance evaluation of Jason when used for distributed crowd simulations [30]; an approach to query caching and a performance analysis of its usage in Jason, 2APL and GOAL [3]; an implementation of Jason in Erlang and a benchmark for evaluating its performance [26]; a quantitative comparison between Jason and two actor-oriented programming languages (Erlang and Scala) using a communication benchmark [16, 17]; and finally a performance evaluation of several benchmarks between agent programming languages (Jason, 2APL, and GOAL) and actor programming languages (Erlang, Akka, and ActorFoundry) [18]. In those cases where performance was considered, Jason typically showed excellent results. In the next section we present JaCaMo and its three technologies: Jason, CArtAgO, and Moise.

³<http://mmi.tudelft.nl/trac/goal>.

⁴<http://aosgrp.com/products/jack>.

⁵<http://www.activecomponents.org/>.

2.1.3 JaCaMo

A JaCaMo⁶ [9] MAS (i.e., a software system programmed in JaCaMo) is given by an agent organisation programmed in Moise, responsible for the organisation of autonomous agents programmed in Jason, and those agents work in a shared distributed artefact-based environment programmed in CArtAgO. An overview of how JaCaMo combines these different levels of abstraction can be seen in Figure 2.4.

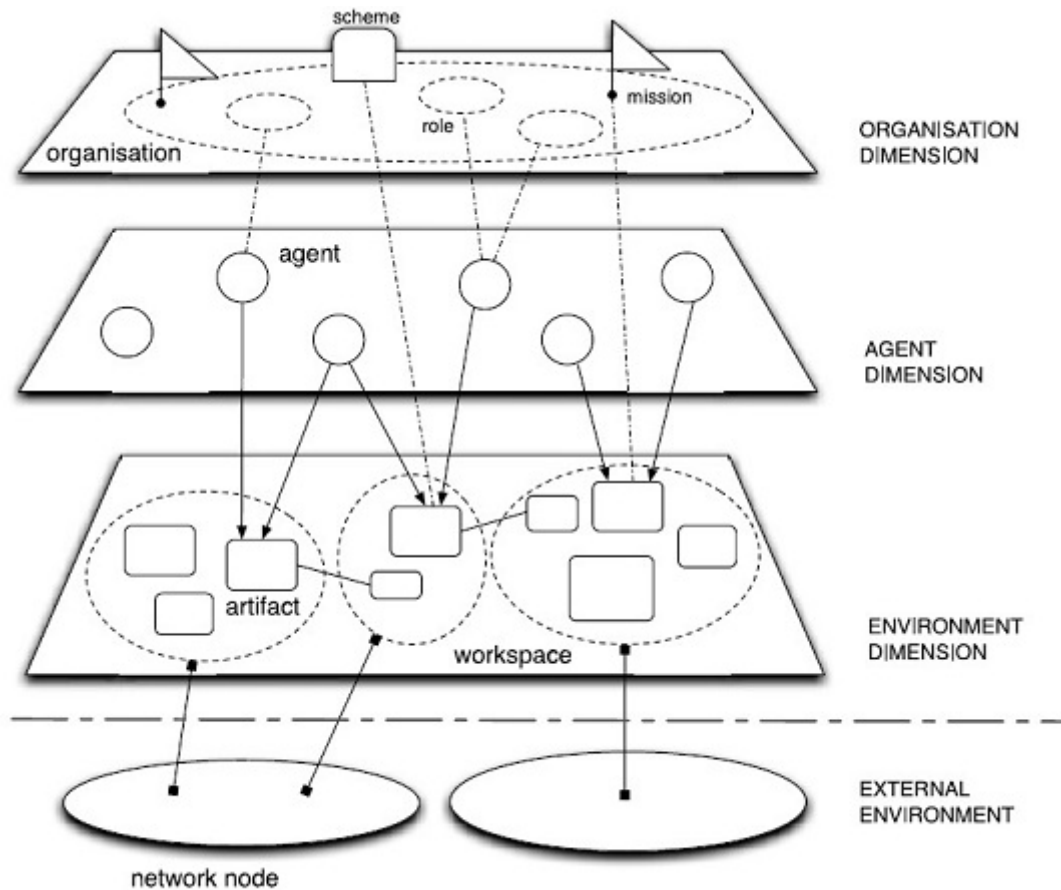


Figure 2.4: Overview of a JaCaMo MAS, highlighting its three dimensions [9].

Jason⁷ [10] is a platform for the development of MAS based on the BDI model, inspired by the AgentSpeak language, Jason focuses on the agent programming level, in Jason an agent is an entity composed of a set of *beliefs*, representing agent's current state and knowledge about the environment in which it is situated, a set of *goals*, which correspond to tasks the agent has to achieve, a set of *intentions*, which are tasks the agent is committed to achieve, and a set of *plans* which are courses of *actions* triggered by *events*.

Events in Jason can be related to changes in either the agent's belief base or its goals. The agent reacts to the events creating new intentions, provided there is an applicable plan for that

⁶<http://jacamo.sourceforge.net/>.

⁷<http://jason.sourceforge.net/>.

event. Therefore, each intention represents a particular “focus of attention” for the various tasks currently being done by the agent: they all compete for the agent’s choice of intention to be further executed in a given execution step.

A project file is used to choose particular settings of the platform (e.g., whether it will run distributed on various hosts or in a single machine) and also to specify all the agents that will take part in the system, for example in “**agents : ag #2**”, two instances of agent **ag** are created and named **ag1** and **ag2**.

In Jason, as in most agent platforms, the communication between agents is based on speech-act theory; formal semantics of speech-act for AgentSpeak can be found in [84]. Messages produce changes on the receivers’ state that are richer than just a new entry in an inbox. For instance, the action `.send(X, achieve, token(5))` sends a message with content `token(5)` to agent `X`. The performative `achieve` is a directive speech-act, it means that the content will be a new goal to be adopted by the receiver, represented by the event `+:token(5)[source(a)]`, where `a` is the sender of the message.

Last, it should be mentioned the “pool of threads” functionality of Jason, declared by using `(pool, x)` next to the infrastructure of choice in the project file. Enabling a pool of threads means that rather than creating one thread for each agent, Jason creates only a fixed number of threads that agents compete for, unless they have nothing to do. Generally its best to choose `x` to be the number of cores of the processor where Jason is running, but programmers are free to set it to any number they want.

CARTAgO⁸ [74] is a framework and infrastructure for environment programming and execution in multi-agent systems. The underlying idea is that the environment can be used as a first-class abstraction for designing MAS, as a computational layer encapsulating functionalities and services that agents can explore at runtime. CARTAgO is based on the A&A meta-model [66], which means that in CARTAgO such software environments can be designed and programmed as a dynamic set of computational entities called artefacts, collected into several workspaces, possibly distributed among various nodes of a network.

Artefacts in CARTAgO are programmed in Java. Artefacts contain a set of operations that can be used to represent the changes that can occur in the environment. The current state of the environment is represented by observable properties; when an agent observes a particular artefact it means that the observable properties of this artefact will be directly represented as beliefs in that agent’s belief base. The agent’s actions in the environment can be mapped to operations in the artefact that can change the observable properties and dynamically update the belief base of all agents that are observing the artefact.

Finally, the Moise⁹ [40] model is used to program for 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. An

⁸<http://CARTAgO.sourceforge.net/>.

⁹<http://moise.sourceforge.net/>.

organisation model is represented by an Extensible Markup Language (XML) file that 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.

The organisation management infrastructure in JaCaMo works through CArtaGo — another example of the natural synergy between the JaCaMo components — creating organisational artefacts that are used to help coordinate the agents that belong to a particular organisation. For example, an organisational artefact maintains a list of which agents should pursue which goals, and this is represented by the obligations that were defined in the normative specification of the XML file of an organisation. When an agent joins an organisation and adopts a role, it acquires the obligations pertaining to that role by observing the respective organisational artefact.

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. The end result is the JaCaMo conceptual framework and platform, that provides high-level first-class support for developing agents, environments, and organisations in synergy, allowing for the development of more complex MAS.

At this point we have presented the necessary background in intelligent agents and multi-agent systems along with several AOP languages, including an extended description of the MAS development framework we chose, JaCaMo. In the next section of this chapter we discuss automated planning, planning in intelligent agents, and finally multi-agent planning.

2.2 Planning

This section presents a brief introduction to planning, planning formalisms such as STRIPS and PDDL, the Graphplan planner, multi-agent planning including early approaches to it, and multi-agent planning problems.

As briefly discussed in Chapter 1, planning uses some form of deliberation to achieve goals by anticipating the effect of possible actions and organising these actions in a way that they can lead to the achievement of those goals. According to [62]: “Automated planning is an area of Artificial Intelligence (AI) that studies this deliberation process computationally”. Next, we take a look at how this deliberation process has been computationally realised over the years.

A planner (see Figure 2.5), as described in [4, 89], is an algorithm that given a description of the initial state of the environment, a goal state, and the actions available to an agent, will generate a plan, i.e. a sequence of actions, that when executed will guarantee the achievement

of the goal. A collection of several of these planning algorithms can be found in [49]. These planning algorithms assume that the environment where the plan was being executed is static, and while it is possible to have static environments in MAS, its more common for the environments to be dynamic, even more so when discussing real world applications. While the planner is generating a solution plan, the beliefs the agents had about the environment could change, rendering the plan ineffective when it is finally conceived, which can result in plan failure. These problems can be addressed with look-ahead planning [35] or plan failure handling [78].

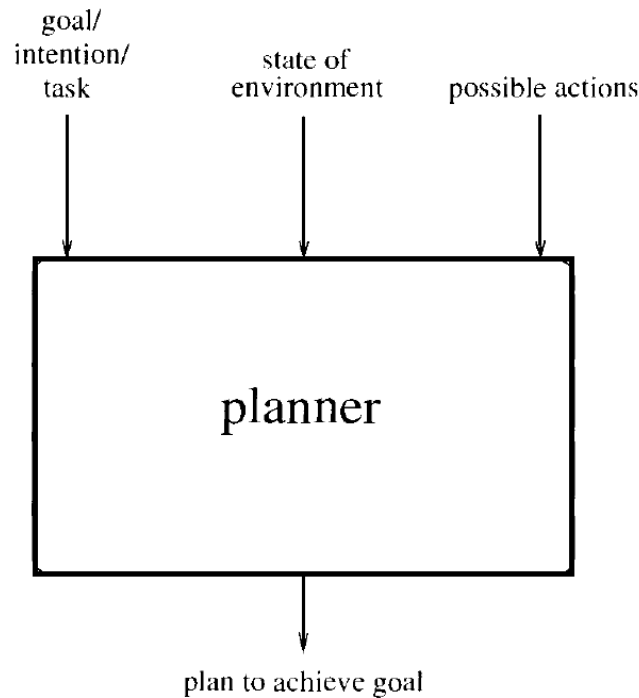


Figure 2.5: Typical overview of a planner [89].

2.2.1 Classical Planning

Classical planning, according to [62], “is a deterministic, static, finite, and fully observable state-transition system with restricted goals and implicit time”. Next, we will describe several planning formalisms and techniques frequently used in the literature for planning with single-agents.

Stanford Research Institute Problem Solver (STRIPS) [31] uses the same steps from the planning algorithm explained above, where the initial state, goal, and actions are characterised using a subset of first-order logic. Accordingly, STRIPS has two basic components: a model of the world, and a set of action schemata that describes the preconditions and effects of all the actions available to the agent. The second component of STRIPS still persists in current AI planning research, and it is still used in some current planners.

A different approach was presented in [8], representing the planning problem in a graph structure using the Graphplan algorithm. A planning problem in Graphplan is described as

the following: a STRIPS-like domain (set of operators); a set of objects; a set of propositions (initial conditions); and a set of problem goals that are required to be true at the end of a plan. The algorithm runs in stages, in a stage i it takes the planning graph from stage $i - 1$ and extends it (i.e. takes the next action level), and then searches for a valid plan of length i . In each iteration, the algorithm either discovers a valid plan or else proves that no plan with that number of time steps or fewer is possible. As stated in [75], Graphplan has problems in domains with lots of objects because too many stages would need to be processed.

The first version of the Planning Domain Definition Language (PDDL) [55] was developed as an attempt to standardise syntax for representing planning problems, and it has been used as the standard language for the International Planning Competition (IPC¹⁰) since 1998. PDDL is able to represent a planning problem in a particular domain by using the following components: objects, things of interest in the domain; predicates, properties of the objects; initial state, the state that the problem starts in; goal state, the result of solving the problem; and actions (operators), define changes to the states of the problem. The latest major version is PDDL 3.0 [34], which has been further extended to PDDL 3.1¹¹.

Planning problems described in PDDL are composed of two files: a domain file with predicates and actions; and a problem file with objects, initial state, and goal state. The Blocks World example — see Listing 2.2 for the domain file, and Listing 2.3 for the problem file — is a popular planning domain in AI; the goal is to move blocks around a surface in order to build one or more vertical stacks of blocks. The Blocks World domain has the following restrictions: only one block may be moved at a time; a block can either be placed on the table or on top of another block; and any blocks that are under another block cannot be moved. Figure 2.6 represents the initial state for the listed problem, and Figure 2.7 represents the goal state. The domain file for this example specifies three predicates, i.e. rules, that can be used in actions: *on* to check if a block is on top of another block or the table; *block* to check if the object is a block; and *clear* to check if an object has nothing on top of it, i.e. it is possible to move another block on top of the object. In this example of the Blocks World domain there are two possible actions, *move*(b, x, y) that corresponds to moving a block b , that is atop a block x , to the top of the block y , and *moveToTable*(b, x) in order to move a block b , that is atop a block x , to the *table*.

Listing 2.2: Domain file for the Blocks World example in PDDL.

```

1 (define (domain blocks)
2   (:requirements :strips)
3   (:constants-def table)
4   (:predicates (on ?a ?b)
5               (block ?b)
6               (clear ?b)

```

¹⁰<http://ipc.icaps-conference.org/>.

¹¹<http://ipc.informatik.uni-freiburg.de/PddlExtension>.

```

7      )
8      (:action move
9          :parameters (?b ?x ?y)
10         :precondition (and (on ?b ?x) (clear ?b)
11                             (clear ?y) (block ?b) (block ?y)
12                             )
13         :effect (and (on ?b ?y) (clear ?x)
14                     (not (on ?b ?x)) (not (clear ?y))
15                     )
16     )
17     (:action moveToTable
18         :parameters (?b ?x)
19         :precondition (and (on ?b ?x) (clear ?b) (block ?b) (block ?x))
20         :effect (and (on ?b table) (clear ?x) (not (on ?b ?x)))
21     )
22 )

```

Listing 2.3: Problem file for the Blocks World example in PDDL.

```

1 (define (problem pb1)
2     (:domain blocks)
3     (:objects a b c table)
4     (:init (on a table)
5             (on b table)
6             (on c a)
7             (block a)
8             (block b)
9             (block c)
10            (clear b)
11            (clear c)
12        )
13     (:goal (and (on a b)
14                (on b c)
15                )
16    )
17 )

```

Problem formalisms are an important part of automated planning, as a planner can use the formalism provided by PDDL to formulate a solution plan for the problem. PDDL is constantly updated with new extensions, resulting in an extensive range of possibilities to represent all kinds of domains and, as a result, only a small subset of PDDL is usually used by the planners.

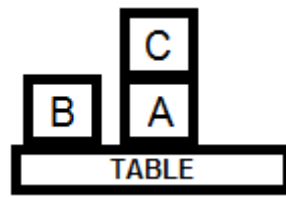


Figure 2.6: Initial state of the Blocks World problem of Listing 2.3.

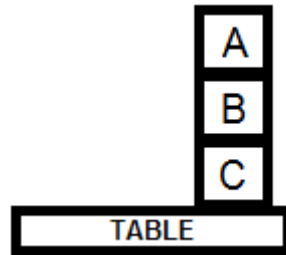


Figure 2.7: Goal state of the Blocks World problem of Listing 2.3.

2.2.2 Multi-agent Planning

Single-Agent planning is concerned with the task of finding a plan, for a single-agent, suitable for a problem. Consequently, Multi-Agent Planning is the process of finding a solution plan for multiple agents. The task of finding a plan can also be done by multiple agents, which is known as Distributed MAP. According to [89], “Multi-agent planning must take into consideration the fact that the activities of agents can interfere with one another — their activities must therefore be coordinated”, thus the coordination of multiple agents can be considered as one of the main problems in MAP.

In [28], planning is classified as follows:

- **Centralised planning for distributed plans:** based on the master-slave model of communication, a plan is developed in which the division and ordering of labour is defined, the master agent then distributes the plan to the slaves who execute their part of the plan. This can be considered centralised planning.
- **Distributed planning:** agents cooperate to form a centralised plan, however the agents that form the plan will not execute it, their role is only to generate the plan. This can be considered distributed single-agent planning.
- **Distributed planning for distributed plans:** agents cooperate to form individual plans, dynamically coordinating their activities along the way. Potential coordination problems may arise, in particular for self-interested agents. Possible solutions for coordination problems are the use of negotiation techniques or an organisational control. This can be considered distributed multi-agent planning.

These classifications are in a top-to-bottom order in complexity, the last one being the most difficult case to consider, as in that case there may never be a global plan representation, the agents may only have pieces of the plan they are interested in. The three MAP algorithms that we used in this work are considered as Distributed planning for distributed plans, or also known as Distributed Multi-Agent Planning.

Now we will discuss some of the earlier approaches to multi-agent planning. In [33], a plan merging algorithm was proposed, where a planner takes a set of plans generated by single-agents and produces a conflict free, not necessarily optimal, multi-agent plan. These set of plans are specified using a modified STRIPS notation, resulting in a synchronised multi-agent plan after the following three stages [89]:

1. **Interaction analysis:** search the agents plans to detect any interactions between them. These interactions are then described using notions of satisfiability, commutativity, and precedence.
2. **Safety analysis:** all actions where there is no interaction are removed from the plan, the remaining actions are considered the set of harmful interactions.
3. **Interaction resolution:** in order to resolve conflicts the harmful interactions are treated as critical sections, therefore mutual exclusion of the critical sections must be guaranteed. Although Hoare’s Communicating Sequential Processes [38] paradigm was used to enforce mutual exclusion, simpler mechanisms, such as semaphores, could be used to achieve the same result.

A similar approach was implemented in [80], which takes a set of unsynchronised single-agent plans and generates a synchronised multi-agent plan. Both approaches use Hoare’s Communicating Sequential Processes to guarantee mutual exclusion of critical sections, and both of them represent actions using an extended STRIPS notation. The main idea of this second approach is that the plans are represented as a set of formulae of temporal logic, and agents attempt to derive a synchronised plan using a temporal logic theorem prover. According to [89] this approach is considered computationally expensive, as the temporal theorem prover has to solve a PSPACE-complete problem.

As stated in [75], decoupling has been a major focus of research on multi-agent planning in order to make the complexity of the problem expand linearly instead of exponentially. This is considered an important topic not only in automated planning, but also in many other areas of AI. Most of the problems in a MAS are loosely coupled¹², and the standard approach to decoupling is to pretend they are completely decoupled and then fix up the interactions. There are many ways of fixing these interactions — e.g. concurrent action list, Hierarchical Task

¹²Any part of a system contains interdependent elements that vary in the number and strength of their interdependencies [67].

Network (HTN), partial observability, replanning, etc. We describe some approaches that use them in the next section.

Research in MAP is generally branched as follows: Distributed MAP is, at the time of this writing, the focus of research in the automated planning community, while coordinating and controlling the actions of the agents in a MAP environment is the focus of research in the MAS community. Bringing those two topics of research together can provide further insight into online (also known as continual) MAP.

These are some basic steps that can be taken for solving a Distributed MAP problem: refinement of the global goal; allocation of planning tasks to agents; coordination before planning; individual planning; coordination after planning; and execution of the solution plan. Not all steps must be necessarily included, some of them can also be combined into the same one.

Usually, agents in a MAP environment do not have access to all of the domain information; instead, this information is distributed among agents. This results in two different approaches for MAP, those that allow the initial state of a planning problem to be incomplete, and those that requires planning with complete information. Coordination mechanism, search algorithm, and information type are only some of the features that have to be considered for an effective multi-agent planning algorithm. Others include how planning information is distributed, which information about the problem each agent has, how common goals are distributed, how private goals are treated, etc.

Therefore, the most difficult multi-agent problems involves the development and execution of joint plans through cooperation (agents on the same team) and competition (agents on opposing teams) without centralised control [75]. In the next section, we take a look at current research in multi-agent planning, from theoretical to practical, that tries to solve some of the problems previously discussed, as well as some of the AOP languages that integrate some form of automated planning, highlighting their limitations.

2.3 Related Work

This section describes some important and relevant research in multi-agent planning, including current approaches, and also some of the related research in single-agent planning.

A survey [57] presents a collection of recent techniques used to integrate automated planning in BDI-based agent-oriented programming languages. The survey focuses mostly on efforts to generate new plans at runtime, while as with our work we translate the output of distributed MAP algorithms into a MAS that is then able to execute the solution plan, the MAP algorithms are not involved during runtime, i.e. the execution stage.

In [54], 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 re-

spects the preferences of the individual agents. Both mechanisms outperformed a first come, first served mechanism. The downside is that the approaches developed were application specific, while in our work we use multi-agent planners for domain independent solutions.

IndiGolog [35], a programming language for autonomous agents, opted not to use classical planning based on the premise that it often ends up being computationally costly. Instead they proposed using a high-level program execution [51], posing as a middle ground between classical planning and normal programming. The general idea is to give the programmer more control over the search effort: if a program is almost deterministic then very little search effort is required; on the other hand the more a program has non-determinism, the more it resembles traditional planning. IndiGolog is an extension of Golog [52], both based on a situation calculus action theory that can be used to perform planning/lookahead. This programming language is not BDI-based, so the plans are not associated with goals and events, which, for example, makes it difficult to find an alternative plan when a selected plan fails [35], while in our work we use the BDI-based language Jason for programming the agents.

CANPLAN2 [78] is a modular extension of CANPLAN [77], both BDI-based formal languages that incorporate an HTN planning mechanism. One of the improvements in CANPLAN2 was to prevent an agent from blindly persisting with a blocked subgoal when an alternative plan is available for achieving a higher-level goal. This approach was further extended in [79] 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 AOP languages. Similarly, in [23] the authors proposed an approach to obtain new abstract plans in BDI systems for hybrid planning problems — where both goal states and the high-level plans already programmed are considered — bringing classical planning into BDI hierarchical structures. This approach was directed to single-agent planning in the context of AOP languages, it does not address the problems of multi-agent planning such as the ones that are approached in our work.

A modification of the X-BDI agent model [61] is presented in [58, 59] that describes the relationship between propositional planning and means-end reasoning of BDI agents; Graphplan was used as the propositional planning algorithm, and applied to a scenario of a production cell controlled by agents. It is common to use Markov Decision Process (MDP) for modelling non-deterministic environments, and so in [81, 82] Earley graphs are used to show that it is possible to bridge the gap between HTNs and MDPs, allowing the use of probabilistic HTN for planning in an MDP environment. These approaches are also related to single-agent planning, different from our approach of domain independent multi-agent planning.

It is claimed in [21] that by associating summary information with plans' abstract operators it can ensure plan correctness, even in multi-agent planning, while still gaining efficiency. 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 compared to a flat representation. This approach depends on some specific conditions and assumptions, and

therefore cannot be used in all domains.

Temporal decoupling refers to the idea that the threads in a parallel system keep their own local time, and only synchronize when they need to communicate with each other. There have been several recent studies regarding the temporal decoupling problem in MAP: in [83], with an algorithm that can be used to determine the minimum number of resources the agent requires to accomplish its ground handling task in an airport application; later in [69] the Simple Temporal Problem (STP) is discussed, showing that finding an optimal decoupling of the STP is NP-hard, and that it can only be solved efficiently if all the agents have linear valuation functions; in [87], the authors propose a characterisation of weakly-coupled problems and quantify the benefits of exploiting different forms of interaction structure between agents; and finally, in [44], a new algorithm is proposed for temporal decoupling and its efficiency is evaluated against current Multiagent Simple Temporal Problem solution algorithms, showing that it still maintains the same computational complexity as the others and even surpasses them in some cases where efficiency is concerned. These studies regarding temporal decoupling are important for the coordination aspect of MAP, in both offline and online planning, even more so in online planning as it often requires time constraints and more tightly coupled systems.

A recent extension for PDDL3.1 enables the description of multi-agent planning problems [48]. 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. If this extension becomes consolidated in the community it will be an important step towards standardisation in the field of multi-agent planning, along with the multi-agent planning track at the 2013 IPC.

Finally, the TAEMS framework [25] 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, and its development has been discontinued since 2006.

In this section we discussed some of the current research on planning and online planning with agent programming or multi-agent systems more generally. While most of it is related to single-agent planning and/or domain specific, in our work we deal with distributed multi-agent planning and domain independent solutions. In this chapter we presented a brief background on intelligent agents and automated planning, with a focus on multi-agent systems and multi-agent planning. In the next chapter we define the details of the translator and the multi-agent planning algorithms.

3. INTEGRATING MAP INTO MAS

In this chapter we present an extension to the JaCaMo framework, allowing it to carry out coordinated plans generated by MAP algorithms. This extension is done by a translator, for which we propose a grammar and translation algorithms, facilitating the introduction of new MAP algorithms into the framework. Initially we considered providing translation algorithms for three MAP algorithms that are available in the literature: Planning-First, MAD-A*, and MAP-POP. However, during our experiments we noticed that MAP-POP had the best interaction with JaCaMo. MAP-POP generates partial order plans allowing parallel goals, while the other two algorithms use total order. The first section of this chapter describes these three algorithms, providing more details for the MAP-POP algorithm. The second section details the translator, its PDDL grammar, its solution grammar, and a set of algorithms that make use of these grammars in order to create a MAS in JaCaMo.

In order to allow the JaCaMo framework to execute the solution generated by the MAP algorithm, we define a grammar and a set of algorithms for a translator, which is used to help bridge the planning and execution stages of multi-agent planning problems. As discussed throughout this dissertation, MAP algorithms usually ignore the execution stage of planning, ending up with just a set of solution 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.

3.1 The Algorithms

In this section we describe three domain independent MAP algorithms, providing details about their input, output, planner, and coordination mechanism. Before we go into details about each algorithm, it is important to define the concepts above. 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 IPCs. As a consequence, single-agent planners 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. A distributed MAP algorithm then, is composed of a planner and a coordination mechanism.

A planner can be classified by the type of node representation: the search can be either state-space or plan-space based. In state-space search algorithms, each node represents a state of the world, and each arc represent a state transition [62]. In plan-space search algorithms,

each node represents partially specified plans rather than states of the world, and arcs are plan-refinement operations; through these refinements open goals can be achieved or inconsistencies removed while further completing partial plans [62]. Besides these differences, the approaches also differ on the definition of a solution: because in state-space algorithms use a sequential transition of states, the resulting order will be just a sequence of actions; while in plan-space algorithms it also needs to provide a partial ordering for those actions. Thus, a plan in a plan-space search is defined as a set of planning operators, a set of ordering constraints, and a set of binding constraints. These constraints are explained in detail in the MAP-POP algorithm section, which uses plan-space representation.

Another classification that can be applied to planners is its plan representation: total-order or partial-order. Total-order consists of strictly linear sequences of actions. Partial-order, on the other hand is used when the planner adopts a plan-space representation, with the use of constraints to control the order of actions when needed [75].

A planner can also be classified by its chaining type: forward chaining or backward chaining. In forward chaining the search starts from the initial state of the world — in plan-space representation the initial node is an empty plan, called initial plan — and tries to find a state that satisfies the global goal. In backward chaining the search starts from the goal state and applies in reverse order the planning operators to generate subgoals, stopping when a set of subgoals satisfy the initial state. Both approaches generate partial solutions at each iteration, the collection of all partial solutions results in the final solution plan [62].

The input of a MAP algorithm refers to 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 to represent 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.

Now that we covered some of the basic aspects of distributed MAP algorithms, we are ready to understand how they work in the three algorithms.

3.1.1 Planning-First Algorithm

Planning-First [64] is a general, distributed MAP algorithm that uses Distributed Constraint Satisfaction Problem (DisCSP) [91] to coordinate the agents, and for local planning it uses the Fast-Forward planning system [39] to search for a plan. In other words, the public aspects that require coordination between agents are dealt with using CSP, while the internal aspects are dealt with using the Fast-Forward planning system.

The Planning-First algorithm takes as problem input PDDL files using a MA-STRIPS [11]

subset. It works similarly to STRIPS, except there are a few extensions such as internal and public actions. Public actions are available to all agents while internal ones are specific to some agents. Another addition was the use of dependencies to denote if the effects of an action affects others. A solution to a MA-STRIPS problem is equivalent to its STRIPS counterpart — an ordered sequence of actions taking the system from its initial state to a state containing all goal propositions.

In [11], the CSP+planning methodology is introduced to separate the public and private aspects of planning problems. This methodology is used to quantify the coupling of a multi-agent planning problem and to facilitate the coordination during the planing stage. It quantifies coupling by using two parameters: the tree-width of the agent interaction graph; and the number of coordination points required per agent.

The Asynchronous Forward Checking (AFC) [56] algorithm for DisCSP uses a middle ground between forward-checking and back-jumping. There is a trade-off that must be addressed between the computational effort invested in generating domains, and the efficiency of the procedure used. This trade-off depends on the coupling level that is required by the particular problem.

The planner uses a state-space based node representation, and therefore, total-order of actions. The Fast-Forward planning system relies on that forward state-space search, using a heuristic to estimate goal distances by ignoring deleted lists, this heuristic information can then be used to prune the search space, improving performance. The system uses a Graphplan-like algorithm to find an explicit relaxed solution to each search state.

An adaptation of the agent’s local planner (Fast-Forward planning system) is also necessary in order to make the search as goal-oriented as possible. This is done by trying to solve the local planning problem where public sub-goals are encoded as private ones, and removing them one by one if the search fails. Another option is using rewards or negative costs so that the planner prefers public-goal achieving plans. The planner was also adapted to store failed plans, so that it does not assign that same action sequence again.

3.1.2 MAD-A* Algorithm

The MAD-A* [63] algorithm is a multi-agent planning adaptation of one of the best known search algorithm, A*. The algorithm also has centralised and parallel versions which will not be discussed here, since our focus is on distributed MAP algorithms. The algorithm distributes the work among different agents, and reduces the overall workload. Moreover, it reduces exactly to A* when there is only a single agent.

The algorithm uses heuristics to help coordinate the agents during the planning stage, and a multi-agent extension of the single-agent planner Fast Downward [36], which has been the basis for the winners of three past IPCs. This algorithm is from the same authors as the previous one, and uses the same formalism to represent MAP problems, MA-STRIPS, that was explained in the previous section.

In MAD-A*, each agent maintains two lists: an “open” list of states that are candidates for expansion and a “closed” list of already expanded states. When an agent expands a state s it uses its own operators only, this means that two agents expanding the same state will generate different successor states.

Since no agent has complete knowledge of the entire search space, messages informing agents of open search nodes relevant to them must be sent. These messages sent between agents contain both public and private variable values, as well as the cost of the best plan from the initial state to s found so far, and the sender’s heuristic estimate of s . The private data sent by an agent is merely used as an ID by other agents, for coordination purposes, and may be encrypted arbitrarily if needed, but only the agent itself may change the value of its own private state.

Once an agent expands a solution state s , it sends it to all agents and initiates the process of verifying its optimality. When the solution is verified as optimal, the agent starts the traceback of the solution plan. When the trace-back phase is done, a terminating message is broadcast.

In order to implement the algorithm the authors developed the Multi-Agent Fast Downward (MA-FD) framework, based on the Fast Downward [36] framework. Minor changes were made to allow the distribution of operators to agents, such as: in addition to PDDL files, MA-FD receives a file detailing the number of agents, their names, and their IP addresses; because agents do not have shared memory, all exchange of information between agents is done by message passing, and inter-agent communication is performed using the TCP/IP protocol.

3.1.3 MAP-POP Algorithm

MAP-POP [50] is a MAP model devised to cope with cooperative planning. The model builds upon the concept of refinement planning [45], where agents propose successive refinements to a base plan until a consistent joint plan that solves the problem’s goals is obtained.

MAP-POP is based on partial-order planning [6], establishing partial order relations between the actions in the plan. POP-based planners work over all the planning goals simultaneously, maintaining partial order relations between actions without committing to a precise order among them until the plan’s own structure determines that it is necessary to do so. The two previous MAP algorithms performed state-space search, while POP models adopt a plan-state search approach, that is, each node in the search represents a different partial plan.

In MAP-POP refinement planning, an agent proposes a plan Π that typically contain some open goals, then the rest of the agents collaborate on the refinement of Π by trying to solve some of its open goals. This means that MAP-POP uses plan-space search, where agents refine an initially empty partial plan, until arriving at a solution. MAP-POP is also classified as backward chaining, since it begin the search by satisfying the problem goals, and build the plan backwards.

The search procedure of refinement planning consists in looking for a flaw in the plan, making additions to the plan in order to correct the flaw, or if there is no correction to be made then the search backtracks and tries something different. A flaw in this context is anything

that is keeping the partial plan in question from being a solution, or part of it [75]. At every step the least commitment [86] approach is used to fix the flaw, meaning that only the essential ordering decisions are recorded.

An argumentation approach is proposed as the coordination mechanism [50] at a theoretical stage only, and so it will not be discussed here. Instead, the coordination mechanism that is implemented uses an ad-hoc heuristic function to validate plan refinements and choose the most appropriate plan to be the next base plan.

To adapt POP to multi-agent planning, the authors introduced several modifications. For initial plan, instead of using a void plan, any partial plan can be used. For subgoal resolution, only the open goal selected by the agents as the current subgoal is solved, the rest of the open goals in the initial plan are ignored until a current subgoal is supported by a causal link, which the planner will then try to solve in a cascading approach. For solution checking, in order to a partial-order plan to be validated as a refinement plan it needs to be threat free (no conflicts), it needs causal links that support the current subgoal, and if the plan has added new open goals over the current base plan, each one of them must be solvable by another agent or group of agents. In regards to planning restarting, instead of finishing when a solution is found, MAP-POP allows the planning process to be continued at will if the agent wants to obtain more refinements to the base plan.

The formalism adopted for specifying a MAP problem, i.e. its input, is PDDL 3.1, with a few modifications to deal with specific MAP requirements. The first one is the addition of the `shared-data` construct defined in the problem file; it indicates the objects that are shared by each agent. The second one is the separation of global (`global-goal`) and local (`private-goal`) goals in the problem file, although only global goals are implemented at the time of writing. The third one is the `multi-functions` construct, which can be used to simplify the specification of objects in the initial state of an agent. Each agent must have a problem file for itself where shared and private data can be specified, and each different type of agent must have its own domain file to specify the actions that are available to that particular agent type.

The 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 two different stages that are interleaved: an internal planning process through which the agents refine the current base plan individually with an internal POP system, and a MAS coordination process that allows agents to exchange the generated refinement plans and to select the next base plan. These stages repeat themselves until a solution plan is found. This process is illustrated in Figure 3.1.

During the initial communication stage a distributed Relaxed Planning Graph (disRPG) [92] is constructed, and it is used to provide agents with important planning information that is used throughout the planning process. The agents only have access to their own RPG, i.e. none of them has the complete representation of the disRPG, this respects agents' privacy. It uses the `shared-data` constructs from the domain definition files to effectively exchange literals

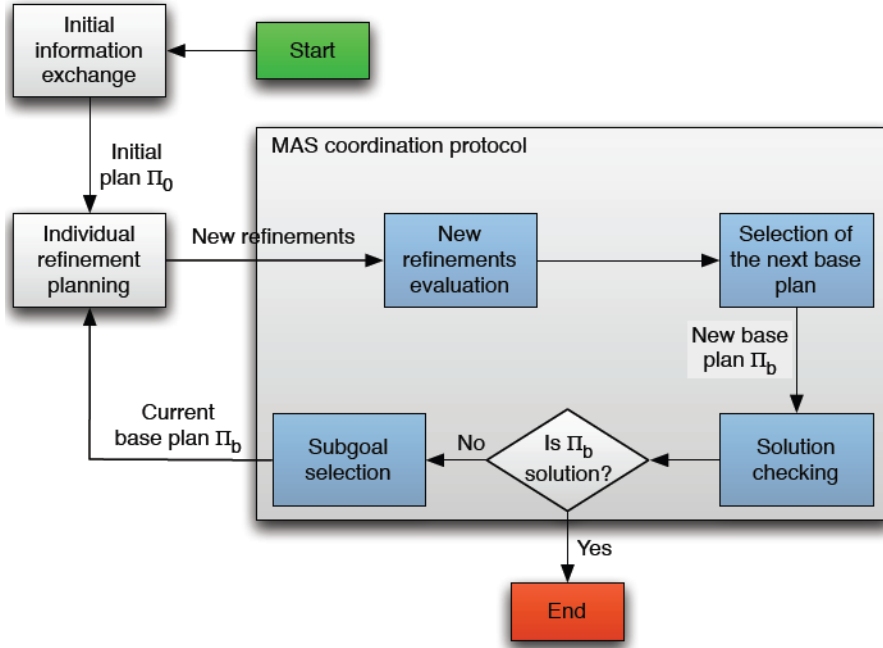


Figure 3.1: Stages of the MAP-POP algorithm [50].

between agents; if a literal in the agent’s RPG matches with a predicate in the `shared-data` construct, it will be sent to the agents specified in this construct.

The disRPG also computes an estimate of the best cost to achieve each literal state, which can be used as heuristic to guide the problem-solving process. After this initial exchange, each agent updates its own RPG: if a literal is not yet in the RPG it is stored according to its cost, otherwise the agent checks the cost of the received literal, if it is lower than the cost registered it is updated in the agent’s RPG. After the RPG is updated the agent expands it, checking if the new literals trigger the appearance of new actions in the RPG, which will then be shared in the next literal exchange stage. The process finishes when there are no new literals in the system.

In the internal planning process each agent in the system executes an individual POP process in order to refine the current base plan. A heuristic function is applied, the SUM heuristic, in order to guide the search. This heuristic is based on the sum of the costs of the open goals found in the previous stage by the RPG. The current subgoal is solved, and all the subgoals that arise from this initial resolution are solved in a cascading fashion, leaving the rest of preconditions unsolved. The valid refinements obtained by the agents during this process are evaluated in the next stage, the coordination stage, in order to select the next base plan among them.

Finally, in the coordination stage a leadership baton is passed among the agents, following a round-robin order. The baton is passed to the next agent when a coordination stage is completed. A coordination stage consists of the following steps:

1. **New refinements evaluation:** the SUM heuristic (same as the one used in the internal planning stage) is used to estimate the quality of a refinement, i.e. to what extent it can

be refined to a solution.

2. **Selection of the next base plan:** agents select the best valued (lower value equals better quality) refinement plan as the new base plan.
3. **Solution checking:** if the current base plan does not have any open goals it is a solution. However, since open goals might not be visible to some agents, all agents are required to send their confirmations to the baton agent to reach a consensus.
4. **Subgoal selection:** the baton agent selects the next most costly open goal to be solved, according to the baton's agent RPG. If the baton agent is not able to find any remaining open goals in the base plan, then it passes the baton to the next agent in order to complete this step. After this step there is a new subgoal to be solved, which starts the next internal planning stage, where the agents will refine the new base plan.

The coordination algorithm can be seen as an A* search in which refinement plans are nodes in the search tree, expanding it during each planning stage. As previously mentioned, these stages repeat themselves until a solution is found.

3.2 The Translation

The translator needs as input the solution plan from a MAP algorithm and the definition of a multi-agent planning problem. 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. Both the Planning-First algorithm and the MAD-A* receives as input a PDDL file using the MA-STRIPS formalism, and the MAP-POP algorithm uses its own extension of PDDL 3.1 for multi-agent planning. This input from the PDDL files of the MAP-POP algorithm is used to define the agents in the MAS project file and the roles in the Moise organisation, while both Planning-First and MAD-A* have a separate file with the name of the agents. We use the PDDL problem file to build CArtaGo

artefacts that represent the initial state of the environment, and also use the global goals to check if the execution stage was successful.

The output of the MAP algorithms consists of a solution that solves the global goal of the problem. Both Planning-First and MAD-A* perform state-space searches, meaning that the solution plan will already contain an ordered sequence of actions, while for the MAP-POP algorithm it is also necessary to provide 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. For this reason we focus mostly on the MAP-POP algorithm when describing the grammar and algorithms for the translator.

Our challenge, then, was to develop a standard JaCaMo output for the translator. The output is a MAS specified in JaCaMo, resulting in multiple files as the output: Jason project and 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 did with MAP-POP), and simply adapt our translation algorithms accordingly.

The solution is translated into AgentSpeak plans, and added to the respective agent's plan library in Jason, but 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 force simple transitions, that is, every action in the set of solution plans 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. A summary of how the translator works is available in the diagram of Figure 3.2.

We now detail the grammar used by the translator. In Listing 3.1 and Listing 3.2 we present a simplified grammar based on the official PDDL 3.1 definition¹. In this simplified version we omit some of the terminals, such as the non-terminal symbol `name` which represents a terminal string of characters `a..z|A..Z`, and the symbol `digit` which represents a sequence of numbers with digits 0..9. Similarly, we present in Listing 3.3 a grammar for the solution generated by MAP-POP.

¹<http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/Resources/kovacs-pddl-3.1-2011.pdf>

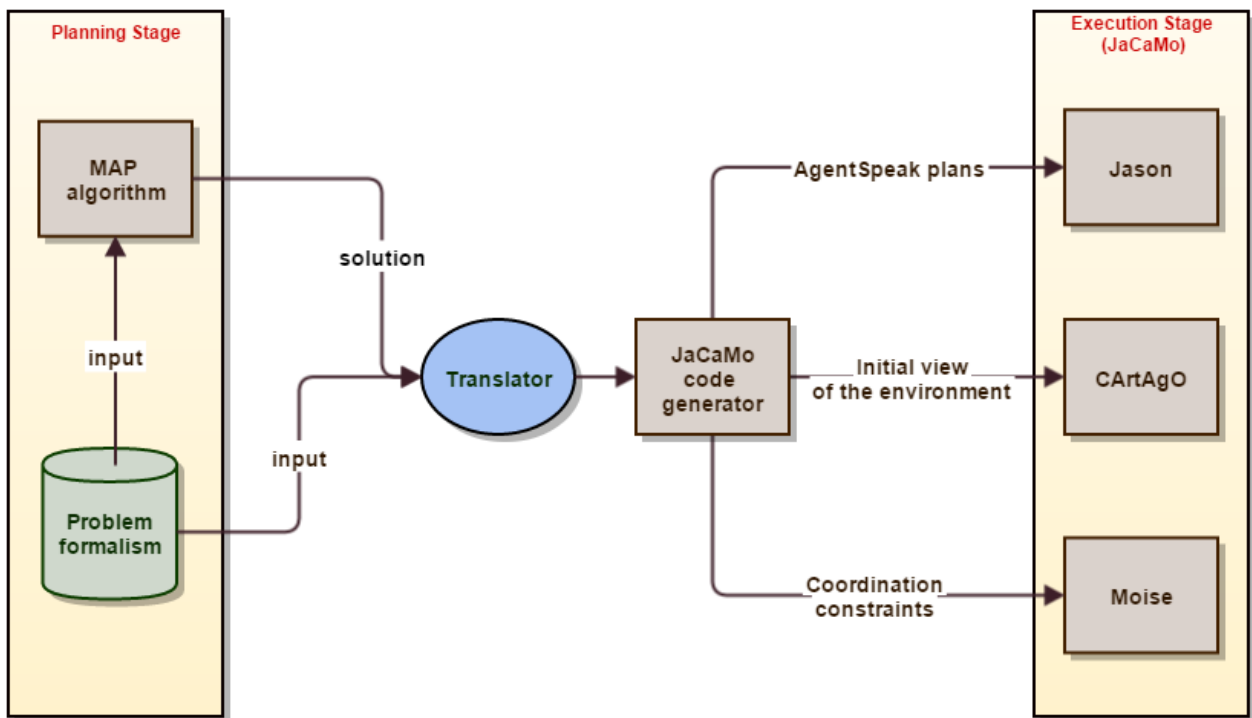


Figure 3.2: An overview of the translation process.

Listing 3.1: Simplified BNF grammar for the syntax definition of the PDDL 3.1 domain description.

```

1  domain          ::= '(define (domain' name '))'
2                      [requireDef]
3                      typesDef
4                      [constantsDef]
5                      [predicatesDef]
6                      [functionsDef]
7                      actionsDef+ '))' ;
8
9  requireDef      ::= '(:requirements' requireKey+ '))' ;
10 requireKey     ::= 'strips' | 'typing' | 'fluents' | 'multi-agent' |
11                   'equality' | 'negative-preconditions' ;
12
13 typesDef        ::= '(:types' typedList+ '))' ;
14 constantsDef   ::= '(:constants' typedList+ '))' ;
15 typedList       ::= (name+ '-' type) | name+ ;
16 type           ::= ( ' (either' name+ '))' ) | name ;
17
18 predicatesDef  ::= '(:predicates' predicate+ '))' ;
19 predicate      ::= '( ' name variable ['- ' type] '))' ;

```

```

20 variable      ::= '?' name ;
21
22 functionsDef ::= '(:functions' textitfunc+ ')' ;
23 func         ::= '(' predicate '-' type ')' '-' type ;
24
25 actionsDef   ::= '(:action' name
26                  ':parameters (' (variable ['- type])* ') ' actionBody ') ' ;
27 actionBody   ::= [ ':precondition (and' atomic* ') ' ]
28                  [ ':effect (and' atomic* ') ' ] ;
29 atomic       ::= atomicFormula | '(not' atomicFormula ') ' ;
30 atomicFormula ::= '(' predicate ') ' | '(=' '(' predicate ') ' variable ') ' ;

```

Listing 3.2: Simplified BNF grammar for the syntax definition of the PDDL 3.1 problem description.

```

1 problem      ::= '((define (problem' name '))'
2                  '(:domain' name '))'
3                  objectsDef
4                  [sharedData]
5                  initDef
6                  globalGoals '))' ;
7
8 objectsDef   ::= '(:objects' typedList+ '))' ;
9
10 sharedData  ::= '(:shared-data' pf+ '- (either' name+ '))' ;
11 pf         ::= predicate | func ;
12
13 initDef     ::= '(:init' literal* '))' ;
14 literal     ::= term | '(not' term ') ' ;
15 term        ::= ( '(' litName first* name* ') ' ) |
16                  ( '(=' '(' litName first ') ' name ') ' ) ;
17
18 litName     ::= name ;
19 first       ::= name ;
20
21 globalGoals ::= '(:global-goal (and' literal* '))' ;

```

Listing 3.3: BNF grammar for solution generated by MAP-POP.

```

1 solution    ::= 'Steps:'
2                initial
3                final

```

```

4         steps+
5         oc ;
6
7 initial ::= '0
8         null
9         Action:' ;
10
11 final   ::= '1
12         null
13         Action: '
14
15 steps   ::= id
16         agent
17         'Action:'
18         name
19         'MinTime:'
20         min
21         'Parameters:'
22         param*
23         'Precond:'
24         precond*
25         'Effect:'
26         atomic* ;
27
28 precond ::= 'Type (1 equals 2 distinct):'
29         digit
30         atomic ;
31
32 oc      ::= 'Ordering Constraints:'
33         ord* ;
34 ord     ::= 'Step 1:' digit+ 'Step 2:' digit+ ;
35
36 id      ::= digit ;
37 agent   ::= name ;
38 min     ::= digit ;
39 param   ::= name ;

```

We now detail the translation algorithms that make use of the previous grammars. Algorithm 3.1 represents the main translation process, 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.typeDL`

means that we will 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 (Algorithm 3.2), which subsequently calls Algorithm 3.3 for the structural specification, Algorithm 3.4 for the functional specification, and Algorithm 3.5 for the normative specification. It then returns to the main algorithm which calls Algorithm 3.6 for the plans in Jason, finally returning and calling Algorithm 3.7 for the CArtAgO artefacts.

Listing 3.4: Main translation algorithm.

```

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

```

Listing 3.5: Organisational specification translation algorithm.

```

1 function createOrg(SolutionSpec, agentsTypes, agents)
2   structural := ∅
3   functional := ∅
4   normative := ∅
5   structural := createStructural(agentsTypes, agents)
6   functional := createFunctional(SolutionSpec, agents)
7   normative := createNormative(agents, functional)
8   return (structural ∪ functional ∪ normative)
9 end function

```

Listing 3.6: Structural specification translation algorithm.

```

1 createStructural(agentsTypes, agents)
2   roles := ∅
3   extendedRoles := ∅
4   for each t1 in agentsType do
5     roles := roles ∪ t1
6   end for
7   for each (a,t2) in agents do
8     extendedRoles := extendedRoles ∪ {(a, t2)}
9   end for
10  return (roles ∪ extendedRoles)
11 end function

```

Listing 3.7: Functional specification translation algorithm.

```

1 createFunctional(SolutionSpec, agents)
2   count := 1
3   minTime := 0
4   goals := ∅
5   missions := ∅
6   for each (g:name,o:min) in SolutionSpec.solution.steps do
7     if (g ∈ goals) then
8       count := count + 1
9     end if
10    if (o ≠ minTime) then
11      goals := goals ∪ {(g,count,'notParallel')}
12      minTime := o
13    end if
14    else goals := goals ∪ {(g,count,'parallel')}
15  end for
16  for each {(a1, _)} in agents do
17    for each (g:name,a2:agent) in SolutionSpec.solution.steps do
18      if (a2 = a1) then
19        m := addFirstChar(a2, 'm')
20        missions := missions ∪ {(g,m)}
21      end if
22    end for
23  end for
24  return (goals ∪ missions)
25 end function

```

Listing 3.8: Normative specification translation algorithm.

```

1 createNormative(agents, functional)
2   norms := ∅
3   for each {(_, m)} in functional.missions do
4     role := deleteFirstChar(m, 'm')
5     norms := norms ∪ {(role,m)}
6   end for
7   return (norms)
8 end function

```

To demonstrate this process consider Listing 3.11 and Listing 3.12, 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 from line 1 in Listing 3.12 (note that the context of a plan starts after the colon) contains the information to access the artefacts, all subsequent lines are each a precondition specified in 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 (line 2 in Listing 3.12). The remaining preconditions access the artefacts and make the necessary tests, lines 3 and 4 in Listing 3.12. Finally, at the body of a Jason plan (remember that 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 (as explained before, 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 that is discussed in Chapter 5.

Listing 3.9: Agent code translation algorithm.

```

1 createAgentCode(SolutionSpec, agents)
2   agentCode := ∅
3   for each (i:id,ag:agent,g:name,pr:param,pc:precond,e:atomic)
4   in SolutionSpec.solution.steps do
5     for each p in pr do
6       if (p ∈ agents) then
7         pr := pr ∪ removeP(pr, p)
8       end if
9     end for
10    agentCode := agentCode ∪ {(i,ag,g,pr,pc,e)}
11  end for
12  return (agentCode)
13 end function

```

Listing 3.10: Environment translation algorithm.

```

1 createEnv(DomainSpec, ProblemSpec)
2   obsProp := ∅
3   updateOp := ∅
4   initAg := ∅
5   arts := ∅
6   for each (pred:name,t:type) in DomainSpec.domain.predicatesDef.predicate do
7     if (t ≠ 'agent') then
8       obsProp := obsProp ∪ {(type,pred)}
9     end if
10  end for
11  for each(func:name,t:type)in DomainSpec.domain.functionsDef.func.predicate do
12    if (t ≠ 'agent') then
13      obsProp := obsProp ∪ {(type,func)}
14    end if
15  end for
16  for each (type, obs) in obsProp do
17    updateOp := updateOp ∪ createOp(type, obs)
18  end for
19  for each n:name in ProblemSpec.problem.objectsDef.typedList do
20    arts := arts ∪ n
21  end for
22  for each art1 in arts do
23    for each (lit:litName,art2:first,objs:name)
24      in ProblemSpec.problem.initDef.literal.term do
25        if (art1 = art2) then
26          initAg := initAg ∪ createArt(art2, lit, objs)
27        end if
28      end for
29    end for
30  return (obsProp ∪ updateProp ∪ initAg)
31 end function

```

Listing 3.11: A step from the solution of a driverlog problem.

```

1 3 // step id
2 driver2 // agent responsible for executing this step
3 Action:
4 board
5 Parameters:
6 driver2 truck1 street0
7 Precond:
8 pos truck1
9 street0
10
11 at driver2
12 street0
13
14 empty truck1
15 true
16 Effect:
17 at driver2
18 truck1
19
20 empty truck1
21 false

```

Listing 3.12: A Jason translated plan for a driverlog problem.

```

1 +!board1 : V1 = "truck1" & V2 = "street0" & id(V1,Id1) & id(V2,Id2) &
2         at(V2) &
3             pos(L)[artifact_id(Id1)] & processList(L,V2) &
4             empty(E)[artifact_id(Id1)] & E
5     <- -at(V2);
6         +at(V1);
7         updateEmpty(false)[artifact_id(Id1)].
8 -!board1 <- .print("Plan board1 failed, check solution plan.").

```

The roles of the organisation are acquired from the formalism used to represent the problem, in case of MAP-POP we use the PDDL files. By checking for agent types in Listing 3.13, and then checking the objects that use those types in Listing 3.14, the translator generates the roles present in Listing 3.15. The coordination constraints from 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. This conversion of coordination constraints into schemas is exemplified in the next chapter, along with the

descriptions of the domains and problems that were used as examples.

Listing 3.13: Types of the driverlog domain.

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

Listing 3.14: Objects from the problem file that use types in the driverlog domain.

```
1 (:objects
2     driver1 driver2 - driver
3     truck1 truck2 - truck
4     package1 package2 - obj
5     s0 s1 s2 p1-0 p1-2 - location
6 )
```

Listing 3.15: Example of translated PDDL types into Moise roles.

```
1 <role-definitions>
2   <role id="driver" />
3     <role id="driver1"> <extends role="driver"/> </role>
4     <role id="driver2"> <extends role="driver"/> </role>
5 </role-definitions>
```

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 with the `.mas2j` extension that was also generated in the translation.

This chapter presented in detail the MAP-POP algorithm, the one that better integrated into JaCaMo, the specifics of the grammars and translation algorithms used in the implementation of the translator, and a brief demonstration of the translation process.

4. CASE STUDIES

In this chapter, two multi-agent planning problems are presented. Both are adaptations of single-agent versions from previous IPCs: the driverlog domain and the depots domain. We explain the domains and present the input used by MAP-POP. As explained in the previous chapter, MAP-POP provided better integration with JaCaMo than the other two algorithms. In the planning stage, Section 4.3, we discuss the solution and coordination constraints found by the MAP-POP algorithm. While in the execution stage, Section 4.4 we discuss the output of the translation and excerpts of the code that was generated.

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.

4.1 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.

A problem in this domain normally consists of having objects in certain locations. Now we specify a simple problem in the Driverlog domain, containing the following objects:

- two agents: driver1 and driver2;
- two trucks: t1 and t2;

- five locations: s_0 , s_1 , s_2 , p_1-0 , and p_1-2 ;

The initial state of the problem can be observed in Figure 4.1. 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 $driver_1$ at s_1 , and t_1 at s_1 . That is, $driver_1$ and $truck_1$ should be at street1.

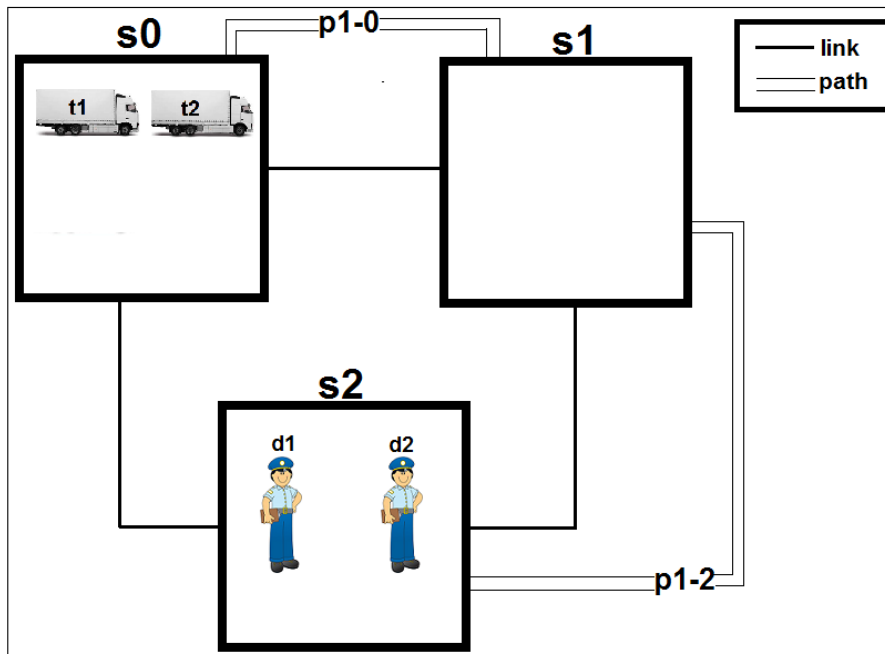


Figure 4.1: Initial state of our problem for the Driverlog domain.

The PDDL files for the driverlog domain and problem are shown in Appendices A.1 and A.2. There are two agents in this example, therefore there are two problem files, one for each agent, but we do not show the problem file for the second agent, `driver2`, as it is the same as for the first agent, `driver1`. Although they are the same in this example, this may not always be the case as the `shared-data` construct can be different, especially when dealing with multiple types of agents. This is not relevant to the translation as the `shared-data` construct is only useful during the planning stage, and therefore only one problem file is needed as input to the translator.

4.2 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.

We define a simple problem in the Depots domain, using the following objects:

- three location agent: depot0, distributor0, and distributor1;
- two truck agents: t1 and t2;
- three pallets: p0, p1, and p2;
- two crates: c0 and c1;
- three hoists: h0, h1, and h2;

The initial state of the problem can be observed in Figure 4.2. 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`.

The PDDL files for the depots domain and problem are shown in Appendices B.1, B.2, and B.3. In this example there is one domain file for the locations and one for the trucks, that is, one domain file for each different type of agent (each type has access to different actions). As in the previous example, we do not show the problem file for the rest of the agents as the translator only requires one problem file.

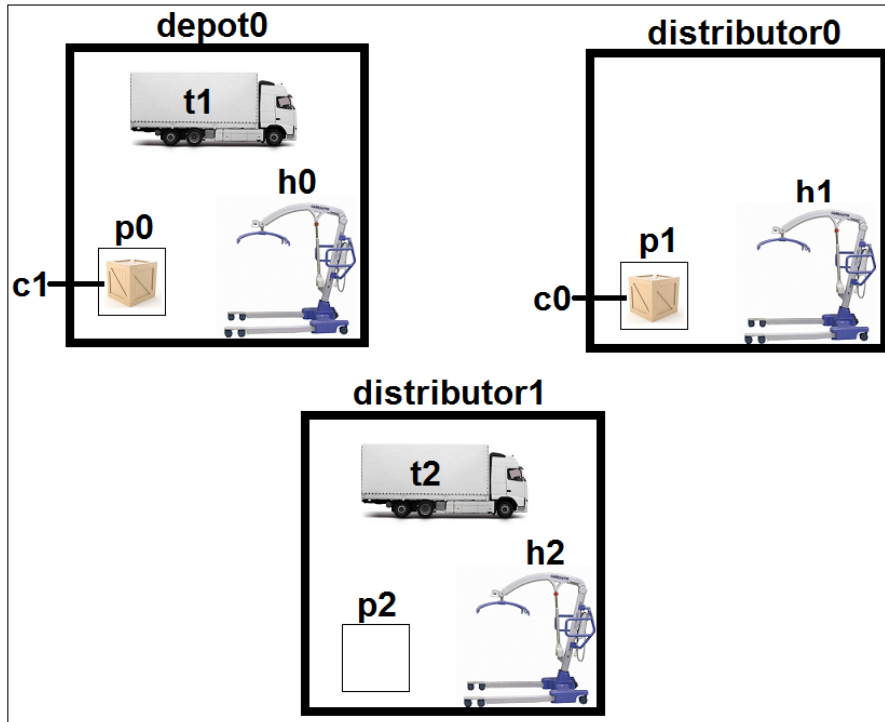


Figure 4.2: Initial state of our problem for the Depots domain.

4.3 Planning Stage

The solution found by MAP-POP for the Driverlog problem contained the following steps¹:

- 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

¹The initial and final steps are always defined previously to the planning stage of the algorithm. This is necessary so that the agents can come up with new plan refinements. Because MAP-POP uses plan-based search, there are no initial and final states.

The order of the Ids of the actions depicts the backwards-chaining aspect of MAP-POP. The first refinements are made to accommodate the final global goals, and then it keeps making refinements until arriving at the conditions that satisfy the subgoals of the initial step.

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): $0 - 7,9 - 6,8 - 5 - 4 - 3 - 2 - 1$.

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

- Id 0 — Initial Step
- Id 1 — Final Step
- Id 2 — agent distributor1: 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
- 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: $0 - 4,10,12 - 6,8 - 9,11 - 5,7 - 2,3 - 1$. Now that we have the solution from both problems we can proceed to the execution stage and check the output of the translation.

4.4 Execution Stage

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. For the Moise functional specification we translate steps into goals, with the plan operators (sequence or parallel) that was extracted from the ordering constraints.

Now we have enough information to create the Moise XML file, which we show in Appendices A.4 for the driverlog domain, and Appendices B.5 for the depots domain. 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. The codes for these artefacts are shown in Appendices A.5, A.6, and A.7 for driverlog, and Appendices B.6 and B.7 for depots.

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. These driverlog translations are located in Appendices A.10 and A.11, and for depots in Appendices B.10, B.11, B.12, and B.13. The necessary plans for agents to join the organisation, focus on artefacts, and commit to goals (i.e., the `common.asl` file) are also available in Appendices A.9 and B.9. The `init` agent, which is responsible for creating the environment and organisation artefacts, is expected to be automated in a configuration file in future versions of JaCaMo. The source code for the translator is available at <https://github.com/rafaelcaue/MAP-JaCaMo/translator>.

5. CONCLUSION

We integrated distributed MAP into JaCaMo through the use of a translator, and detailed its grammar and translation algorithms. JaCaMo provided practical solutions for some of the problems that appeared, 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 translator takes as input the name of the MAP algorithm to be used and its associated input. As output, it generates an organisation in Moise, Jason plans that are added to the agents plan library, and CArtAgO artefacts that represent the initial state of the environment.

There are many experiments that can be done with this translator for multi-agent planning problems. For example, it can be further extended to cope with the problems in multi-robot planning. ROS¹ is an open-source, meta-operating system for robots that provides hardware abstraction, low-level device control implementation of commonly-used functionalities, message-passing between processes, and package management. ROS can be interfaced with JaCaMo, which can be operate along with the MAP algorithms as a cognitive high-level control mechanism.

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, for example, the PDDL 3.1 Multi-Agent extension introduced in [48], or a completely new formalism. 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.

Some performance adaptations could also be made to the MAP algorithms, for example, the Planning-First algorithm has a low performance on tightly-coupled systems, but by using temporal decoupling algorithms it could improve its performance by reducing the coupling in the system. By using the grammar and translation algorithms that we provided, the process of adding new MAP algorithms into JaCaMo also becomes easier.

Our goal with this research 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.

There are a few downsides that we identified during this work:

¹<http://www.ros.org/>.

- although our translator can be used to fill the gap between planning and execution stages, it does not provide a seamless transition;
- 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. From the three algorithms discussed, MAP-POP is the one with the best computational performance according to [50].

These downsides all have possible solutions. One of them is to develop a new MAP algorithm based on the Hierarchical Task Network (HTN) [76] formalism, which is 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.

As for the translator, JaCaMo agents can be used 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.

REFERENCES

- [1] S. Abras, S. Pesty, S. Ploix, and M. Jacomino. Advantages of MAS for the Resolution of a Power Management Problem in Smart Homes. In *Advances in Practical Applications of Agents and Multiagent Systems*, volume 70 of *Advances in Intelligent and Soft Computing*, pages 269–278. Springer Berlin Heidelberg, 2010.
- [2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
- [3] N. Alechina, T. Behrens, K. Hindriks, and B. Logan. Query Caching in Agent Programming Languages. In *Proceedings of the 10th International Workshop on Programming Multiagent Systems (ProMAS-2012), held with AAMAS-2012*, pages 117–131, Valencia, Spain, 2012.
- [4] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Mateo, CA, 1990.
- [5] R. S. Alonso, D. I. Tapia, Ó. García, D. Sancho, and M. Sánchez. Improving context-awareness in a healthcare multi-agent system. In *Trends in Practical Applications of Agents and Multiagent Systems*, volume 90 of *Advances in Intelligent and Soft Computing*, pages 1–8. Springer Berlin Heidelberg, 2011.
- [6] A. Barrett and D. S. Weld. Partial-Order Planning: Evaluating Possible Efficiency Gains. *Artificial Intelligence*, 67(1):71–112, 1994.
- [7] T. M. Behrens, K. Hindriks, J. Hübner, and M. Dastani. Putting APL Platforms to the Test: Agent Similarity and Execution Performance. Technical Report IfI-10-09, Clausthal University of Technology, 2010.
- [8] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.
- [9] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 2011.
- [10] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
- [11] R. I. Brafman and C. Domshlak. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *ICAPS*, pages 28–35, 2008.

- [12] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning, 1988. *Computational Intelligence*, vol. 4, 349-355.
- [13] L. Braubach, A. Pokahr, and W. Lamersdorf. A universal criteria catalog for evaluation of heterogeneous agent development artifacts. In B. Jung, F. Michel, A. Ricci, and P. Petta, editors, *From Agent Theory to Agent Implementation (AT2AI-6)*, pages 19–28, 2008.
- [14] S. Bromuri, M. I. Schumacher, and K. Stathis. Pervasive healthcare using self-healing agent environments. In *Highlights in Practical Applications of Agents and Multiagent Systems*, volume 89 of *Advances in Intelligent and Soft Computing*, pages 159–166. Springer Berlin Heidelberg, 2011.
- [15] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java, 1999. AgentLink News, Issue 2.
- [16] R. C. Cardoso, J. F. Hübner, and R. H. Bordini. Benchmarking communication in actor- and agent-based languages. In *12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013), extended abstract, Saint Paul, Minnesota, USA, 2013*.
- [17] R. C. Cardoso, J. F. Hübner, and R. H. Bordini. Benchmarking Communication in Agent- and Actor-Based Languages. In *Proceedings of the EMAS '13, held with AAMAS-2013*, pages 81–96, Saint Paul, Minnesota, USA, 2013.
- [18] R. C. Cardoso, M. R. Zатели, J. F. Hübner, and R. H. Bordini. Towards Benchmarking Actor- and Agent-Based Programming Languages. In *Proceedings of the 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE! @ SPLASH 2013)*, Indianapolis, Indiana, USA, 2013.
- [19] K. M. Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing, Boston, MA, USA, 1988.
- [20] M. Chen. Towards smart city: M2M communications with software agent intelligence. *Multimedia Tools and Applications*, pages 1–12, 2012.
- [21] B. J. Clement, E. H. Durfee, and A. C. Barrett. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research (JAIR)*, 28:453–515, 2007.
- [22] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
- [23] L. de Silva, S. Sardiña, and L. Padgham. First principles planning in BDI systems. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, Volume 2*, pages 1105–1112, 2009.

- [24] M. de Weerd and B. Clement. Introduction to Planning in Multiagent Systems. *Multiagent Grid Syst.*, 5(4):345–355, 2009.
- [25] K. Decker. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. *Foundations of Distributed Artificial Intelligence, Chapter 16*, pages 429–448, 1996.
- [26] Á. F. Díaz, C. B. Earle, and L.-A. Fredlund. eJason: an implementation of Jason in Erlang. In *In Proceedings of the 10th International Workshop on Programming Multi-Agent Systems (ProMAS-2012), held with AAMAS-2012*, pages 7–22, Valencia, Spain, 2012.
- [27] A. J. Dinusha Rathnayaka, V. M. Potdar, and S. J. Kuruppu. Energy resource management in smart home: State of the art and challenges ahead. In N. M’Sirdi, A. Namaane, R. Howlett, and L. Jain, editors, *Sustainability in Energy and Buildings*, volume 12 of *Smart Innovation, Systems and Technologies*, pages 403–411. Springer Berlin Heidelberg, 2012.
- [28] E. H. Durfee. Distributed problem solving and planning. In *Multiagent systems*, pages 121–164. MIT Press, 1999.
- [29] A. Feliachi and R. Belkacemi. Intelligent Multi-agent System for Smart Grid Power Management. In *Smart Power Grids 2011*, Power Systems, pages 515–542. Springer Berlin Heidelberg, 2012.
- [30] V. Fernández, F. Grimaldo, M. Lozano, and J. M. Orduña. Evaluating Jason for Distributed Crowd Simulations. In *ICAART 2010*, pages 206–211, 2010.
- [31] R. E. Fikes and N. J. Nilsson. 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, 1971.
- [32] M. Georgeff and A. Lansky. Procedural knowledge. *Proceedings of the IEEE (Special Issue on Knowledge Representation)*, 74:1383–1398, 1986.
- [33] M. P. Georgeff. Communication and interaction in multi-agent planning. In *Distributed Artificial Intelligence*, pages 200–204. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
- [34] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition. Technical report, Department of Electronics for Automation, University of Brescia, Italy, 2005.

- [35] G. Giacomo, Y. Lespérance, H. J. Levesque, and S. Sardina. IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. In *Multi-Agent Programming: Languages, Tools and Applications*, chapter 2, pages 31–72. Springer, 2009.
- [36] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.
- [37] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent Programming with Declarative Goals. In *Proceedings of the 7th International Workshop on Agent Theories Architectures and Languages, Boston, MA, USA*, pages 228–243. Springer, 2000.
- [38] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
- [39] J. Hoffmann and B. Nebel. The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302, 2001.
- [40] J. F. Hübner, J. S. Sichman, and O. Boissier. 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, 2007.
- [41] P. Jaszczyk and D. Król. Updatable Multi-agent OSGi Architecture for Smart Home System. In *Agent and Multi-Agent Systems: Technologies and Applications*, volume 6071 of *Lecture Notes in Computer Science*, pages 370–379. Springer Berlin Heidelberg, 2010.
- [42] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.
- [43] H. Jordan, G. Botterweck, M.-P. Huet, and R. Collier. A feature model of actor, agent, and object programming languages. In *Proceedings of the compilation of the co-located workshops on DSM’11, TMC’11, AGERE!’11, AOOPEs’11, NEAT’11, & VMIL’11, SPLASH ’11 Workshops*, pages 147–158, New York, NY, USA, 2011. ACM.
- [44] J. C. B. Jr. and E. H. Durfee. Distributed algorithms for solving the multiagent temporal decoupling problem. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan*, pages 141–148, 2011.
- [45] S. Kambhampati. Refinement Planning as a Unifying Framework for Plan Synthesis. *AI Magazine*, 18(2):67–97, 1997.
- [46] S. Khanna, T. Cleaver, A. Sattar, D. Hansen, and B. Stantic. Multiagent based scheduling of elective surgery. In *Principles and Practice of Multi-Agent Systems*, volume 7057 of *Lecture Notes in Computer Science*, pages 74–89. Springer Berlin Heidelberg, 2012.

- [47] J. Ko, I.-H. Shin, G.-L. Park, H.-Y. Kwak, and K.-J. Ahn. Design of a multi-agent system for personalized service in the smart grid. In *Security-Enriched Urban Computing and Smart Grid*, volume 78 of *Communications in Computer and Information Science*, pages 267–273. Springer Berlin Heidelberg, 2010.
- [48] D. L. Kovacs. A Multi-Agent Extension of PDDL3.1. In *Proceedings of the 3rd Workshop on the International Planning Competition (IPC), held in the 22nd International Conference on Automated Planning and Scheduling, ICAPS-2012*, pages 19–27, Atibaia, São Paulo, Brazil, 2012.
- [49] S. M. LaValle. *Planning algorithms*. Cambridge University Press, 2006.
- [50] A. T. Lerma. Design and implementation of a Multi-Agent Planning system. Master’s thesis, Polytechnic University of Valencia, Valencia, Spain, 2011.
- [51] H. Levesque and R. Reiter. High-level Robotic Control: Beyond Planning. A Position Paper. In *AIII 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap*, 1998.
- [52] H. J. Levesque, R. Reiter, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:54–84, 1997.
- [53] A. Luberg, T. Tammet, and P. Järv. Smart City: A Rule-based Tourist Recommendation System. In *Information and Communication Technologies in Tourism 2011*, pages 51–62. Springer Vienna, 2011.
- [54] X. Mao, A. Mors, N. Roos, and C. Witteveen. Coordinating Competitive Agents in Dynamic Airport Resource Scheduling. In *Proceedings of the 5th German conference on Multiagent System Technologies*, pages 133–144, Berlin, Heidelberg, 2007.
- [55] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control, 1998.
- [56] A. Meisels and R. Zivan. Asynchronous Forward-checking for DisCSPs. *Constraints Journal*, 12(1):131–150, 2007.
- [57] F. Meneguzzi and L. De Silva. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review*, FirstView:1–44, 2013.
- [58] F. Meneguzzi, A. F. Zorzo, M. da Costa Móra, and M. Luck. Incorporating planning into BDI agents. *Scalable Computing: Practice and Experience*, 8:15–28, 2007.

- [59] F. R. Meneguzzi, A. F. Zorzo, and M. D. C. Móra. Propositional planning in BDI agents. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 58–63, Nicosia, Cyprus, 2004.
- [60] I. Mocanu and A. M. Florea. A Multi-agent System for Human Activity Recognition in Smart Environments. In *Intelligent Distributed Computing V*, volume 382 of *Studies in Computational Intelligence*, pages 291–301. Springer Berlin Heidelberg, 2012.
- [61] M. C. Mora, J. G. Lopes, R. M. Viccariz, and H. Coelho. BDI Models and Systems: Reducing the Gap. In *Intelligent Agents V: Agents Theories, Architectures, and Languages*, volume 1555 of *Lecture Notes in Computer Science*, pages 11–27. Springer Berlin Heidelberg, 1999.
- [62] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
- [63] R. Nissim and R. I. Brafman. Multi-Agent A* for Parallel and Distributed Systems. In *Proceedings of the Heuristics for Domain-Independent Planning Workshop, held with ICAPS 12*, 2012.
- [64] R. Nissim, R. I. Brafman, and C. Domshlak. A General, Fully Distributed Multi-Agent Planning Algorithm. In *AAMAS 10*, pages 1323–1330, 2010.
- [65] G. M. P. O’Hare. Foundations of distributed artificial intelligence. In *Agent factory: an environment for the fabrication of multiagent systems*, pages 449–484. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [66] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.
- [67] J. D. Orton and K. E. Weick. Loosely Coupled Systems: a reconceptualization. *Academy of Management Review*, 15:203–223, 1990.
- [68] A. M. Owen. Cognitive planning in humans: Neuropsychological, neuroanatomical and neuropharmacological perspectives. *Progress in Neurobiology*, 53(4):431 – 450, 1997.
- [69] L. Planken, M. de Weerd, and C. Witteveen. Optimal temporal decoupling in multiagent systems. In *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada*, pages 789–796, 2010.
- [70] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85, 2003.

- [71] A. S. Rao. 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 : agents breaking away: agents breaking away*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996.
- [72] A. S. Rao and M. P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the first International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, 1995.
- [73] R. B. Raz Nissim, Daniel L. Kovacs, editor. *Proceedings of the 1st Workshop on Distributed and Multi-Agent Planning*, Rome, Italy, 2013.
- [74] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. 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, 2009.
- [75] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.
- [76] E. D. Sacerdoti. A Structure For Plans and Behavior. Technical Report 109, AI Center, SRI International, Menlo Park, CA, USA, 1975.
- [77] S. Sardiña, L. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: a formal approach. In *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*, Hakodate, Japan, pages 1001–1008, 2006.
- [78] S. Sardiña and L. Padgham. Goals in the context of BDI plan failure and planning. In *6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, Honolulu, Hawaii, USA, page 7, 2007.
- [79] S. Sardiña and L. Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.
- [80] C. Stuart. An implementation of a multi-agent plan synchronizer. In *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2*, IJCAI'85, pages 1031–1033, San Francisco, CA, USA, 1985.
- [81] Y. Tang, F. Meneguzzi, S. Parsons, and K. Sycara. Planning over MDPs through Probabilistic HTNs. In *Proceedings of the AAAI-11 Workshop on Generalized Planning*, 2011.
- [82] Y. Tang, F. Meneguzzi, S. Parsons, and K. Sycara. Probabilistic Hierarchical Planning over MDPs. In *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems*, pages 1143–1144, 2011.

- [83] P. van Leeuwen and C. Witteveen. Temporal decoupling and determining resource needs of autonomous agents in the airport turnaround process. In *Proceedings of the 2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2009, Milan, Italy*, pages 185–192, 2009.
- [84] R. Vieira, Á. F. Moreira, M. Wooldridge, and R. H. Bordini. On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. *Journal of Artificial Intelligence Research (JAIR)*, 29:221–267, 2007.
- [85] M. D. Weerd, A. T. Mors, and C. Witteveen. Multi-agent planning: An introduction to planning and coordination. Technical report, Handouts of the European Agent Summer, 2005.
- [86] D. S. Weld. An Introduction to Least Commitment Planning. *AI Magazine*, 15(4):27–61, 1994.
- [87] S. J. Witwicki and E. H. Durfee. Towards a unifying characterization for quantifying weak coupling in dec-POMDPs. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan*, pages 29–36, 2011.
- [88] M. Wooldridge. Intelligent agents. In *Multiagent systems: a modern approach to distributed artificial intelligence*, pages 449–484. John Wiley & Sons, Inc., New York, NY, USA, 1999.
- [89] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 1st edition, 2002.
- [90] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.
- [91] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
- [92] J. F. Zhang, X. T. Nguyen, and R. Kowalczyk. Graph-based Multiagent Replanning Algorithm. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '07*, pages 122:1–122:8. ACM, 2007.

A. Driverlog domain codes

Listing A.1: Driverlog MAP-POP PDDL domain.

```

1 (define (domain driverlog)
2 (:requirements :typing :equality :fluents)
3 (:types location truck obj - object
4     driver - agent)
5 (:predicates (link ?x ?y - location)
6             (path ?x ?y - location) ; list
7             (empty ?v - truck) ; boolean
8             (myAgent ?a - agent))
9 (:functions (at ?d - driver) - (either location truck)
10            (pos ?t - truck) - location
11            (in ?o - obj) - (either location truck))
12 (:action load
13   :parameters (?obj - obj ?truck - truck ?loc - location)
14   :precondition (and (= (pos ?truck) ?loc) (= (in ?obj) ?loc))
15   :effect       (assign (in ?obj) ?truck))
16 (:action unload
17   :parameters (?obj - obj ?truck - truck ?loc - location)
18   :precondition (and (= (pos ?truck) ?loc) (= (in ?obj) ?truck))
19   :effect       (assign (in ?obj) ?loc))
20 (:action board
21   :parameters (?driver - driver ?truck - truck ?loc - location)
22   :precondition (and (myAgent ?driver) (= (pos ?truck) ?loc)
23                  (= (at ?driver) ?loc) (empty ?truck))
24   :effect       (and (assign (at ?driver) ?truck) (not (empty ?truck))))
25 (:action disembark
26   :parameters (?driver - driver ?truck - truck ?loc - location)
27   :precondition (and (myAgent ?driver) (= (pos ?truck) ?loc)
28                  (= (at ?driver) ?truck))
29   :effect       (and (assign (at ?driver) ?loc) (empty ?truck)))
30 (:action drive
31   :parameters (?truck - truck ?loc-from - location ?loc-to - location
32              ?driver - driver)
33   :precondition (and (myAgent ?driver) (= (pos ?truck) ?loc-from)
34                  (= (at ?driver) ?truck) (link ?loc-from ?loc-to))
35   :effect       (assign (pos ?truck) ?loc-to))
36 (:action walk

```

```

37  :parameters (?driver - driver ?loc-from - location ?loc-to - location)
38  :precondition (and (myAgent ?driver) (= (at ?driver) ?loc-from)
39                (path ?loc-from ?loc-to))
40  :effect      (assign (at ?driver) ?loc-to))

```

Listing A.2: Driverlog MAP-POP PDDL problem.

```

1  (define (problem DLOG-2-2-2)
2  (:domain driverlog)
3  (:objects
4  driver1 driver2 - driver
5  truck1 truck2 - truck
6  package1 package2 - obj
7  street0 street1 street2 p10 p12 - location
8  )
9  (:shared-data
10   (empty ?v - truck)
11   ((at ?d - driver) - (either location truck))
12   ((pos ?t - truck) - location)
13   ((in ?o - obj) - (either location truck)) -
14   driver2
15  )
16  (:init (myAgent driver1)
17   (= (at driver1) street2)
18   (= (at driver2) street2)
19   (= (pos truck1) street0)
20   (empty truck1)
21   (= (pos truck2) street0)
22   (empty truck2)
23   (= (in package1) street0)
24   (= (in package2) street0)
25   (link street0 street1)
26   (link street0 street2)
27   (path street0 p10)
28   (link street1 street0)
29   (link street1 street2)
30   (path street1 p10)
31   (path street1 p12)
32   (link street2 street0)
33   (link street2 street1)
34   (path street2 p12)
35   (path p10 street0)

```

```

36 (path p10 street1)
37 (path p12 street1)
38 (path p12 street2)
39 )
40 (:global-goal (and
41 (= (at driver1) street1)
42 (= (pos truck1) street1)
43 (= (in package1) street0)
44 (= (in package2) street0)
45 ))

```

Listing A.3: Driverlog JaCaMo project file.

```

1 MAS driverlog {
2   infrastructure: JaCaMo
3   agents:
4     init;
5     driver1;
6     driver2;
7   aslSourcePath: "src/asl";
8 }

```

Listing A.4: Driverlog Moise organisation.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <organisational-specification
3   id="driverlog"
4   os-version="0.8"
5
6   xmlns='http://moise.sourceforge.net/os'
7   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
8   xsi:schemaLocation='http://moise.sourceforge.net/os
9     http://moise.sourceforge.net/xml/os.xsd' >
10 <structural-specification>
11 <role-definitions>
12 <role id="driver" />
13 <role id="driver1" > <extends role="driver"/> </role>
14 <role id="driver2" > <extends role="driver"/> </role>
15 </role-definitions>
16 <group-specification id="driverlog">
17 <roles>
18 <role id="driver1" min="1" max="1"/>

```

```
19 <role id="driver2" min="1" max="1"/>
20 </roles>
21 </group-specification>
22 </structural-specification>
23 <functional-specification>
24 <scheme id="driverlog_sch">
25 <goal id="goal">
26 <plan operator="sequence">
27 <goal id="par1">
28 <plan operator="parallel">
29 <goal id="walk1" />
30 <goal id="walk2" />
31 </plan>
32 </goal>
33 <goal id="par2">
34 <plan operator="parallel">
35 <goal id="walk3" />
36 <goal id="walk4" />
37 </plan>
38 </goal>
39 <goal id="walk5" />
40 <goal id="walk6" />
41 <goal id="board1" />
42 <goal id="drive1" />
43 </plan>
44 </goal>
45 <mission id="mdriver1" min="1" max="1">
46 <goal id="walk2" />
47 <goal id="walk4" />
48 </mission>
49 <mission id="mdriver2" min="1" max="1">
50 <goal id="walk1"/>
51 <goal id="walk3" />
52 <goal id="walk5" />
53 <goal id="walk6" />
54 <goal id="board1" />
55 <goal id="drive1" />
56 </mission>
57 </scheme>
58 </functional-specification>
59 <normative-specification>
```

```

60 <norm id="n1" type="obligation" role="driver1" mission="mdriver1" />
61 <norm id="n2" type="obligation" role="driver2" mission="mdriver2" />
62 </normative-specification>
63 </organisational-specification>

```

Listing A.5: Driverlog location artefact.

```

1 package tools;
2
3 import cartago.Artifact;
4 import cartago.OPERATION;
5 import cartago.ObsProperty;
6
7 public class location extends Artifact {
8     @OPERATION public void init(String newLink, String newPath) {
9         defineObsProperty("link", newLink);
10        defineObsProperty("path", newPath);
11    }
12    @OPERATION public void updateLink(String newLink) {
13        ObsProperty opLink = getObsProperty("link");
14        opLink.updateValue("[ "+newLink+" ]");
15    }
16    @OPERATION public void updatePath(String newPath) {
17        ObsProperty opPath = getObsProperty("path");
18        opPath.updateValue("[ "+newPath+" ]");
19    }
20 }

```

Listing A.6: Driverlog obj artefact.

```

1 package tools;
2
3 import cartago.Artifact;
4 import cartago.OPERATION;
5 import cartago.ObsProperty;
6
7 public class obj extends Artifact {
8     @OPERATION public void init(String newIn) {
9         defineObsProperty("in", newIn);
10    }
11
12    @OPERATION public void updateIn(String newIn) {

```

```

13     ObsProperty opIn = getObsProperty("in");
14         opIn.updateValue("[+newIn+]");
15     }
16 }

```

Listing A.7: Driverlog truck artefact.

```

1  package tools;
2
3  import cartago.Artifact;
4  import cartago.OPERATION;
5  import cartago.ObsProperty;
6
7  public class truck extends Artifact {
8      @OPERATION public void init(Boolean newEmpty, String newPos) {
9          defineObsProperty("empty", newEmpty);
10         defineObsProperty("pos", newPos);
11     }
12     @OPERATION public void updateEmpty(Boolean newEmpty) {
13         ObsProperty opEmpty = getObsProperty("empty");
14         opEmpty.updateValue(newEmpty);
15     }
16     @OPERATION public void updatePos(String newPos) {
17         ObsProperty opPos = getObsProperty("pos");
18         opPos.updateValue("[+newPos+]");
19     }
20
21 }

```

Listing A.8: Driverlog init agent.

```

1  !start.
2
3  +!start <- !create_truck_art("truck1", true, [street0]);
4             !create_truck_art("truck2", true, [street0]);
5             !create_obj_art("package1", [street0]);
6             !create_obj_art("package2", [street0]);
7             !create_location_art("street0", [street1,street2], [p10]);
8             !create_location_art("street1", [street0,street2], [p10,p12]);
9             !create_location_art("street2", [street0,street1], [p12]);
10            !create_location_art("p10", [], [street0,street1]);
11            !create_location_art("p12", [], [street1,street2]);

```



```

12         createWorkspace("ora4mas");
13     joinWorkspace("ora4mas",_);
14         makeArtifact("driverlog","ora4mas.nopl.GroupBoard",
15             ["src/driverlog-os.xml", driverlog, false, true],GrArtId);
16     focus(GrArtId);
17         makeArtifact("driverlog_sch", "ora4mas.nopl.SchemeBoard",
18             ["src/driverlog-os.xml", driverlog_sch, false, true], SchArtId);
19         .broadcast(achieve, join("driverlog"));
20     focus(SchArtId);
21     ?formationStatus(ok)[artifact_id(GrArtId)];
22     addScheme("driverlog_sch")[artifact_id(GrArtId)].
23
24 +!create_truck_art(ArtName,Empty,Pos)
25     <- .term2string(Pos,PosS);
26     makeArtifact(ArtName, "tools.truck", [Empty, PosS], ArtId);
27     .broadcast(achieve, discover_art(ArtName)).
28 -!create_truck_art(ArtName,Empty,Pos)[error_code(Code)]
29     <- .print("Error creating artifact ", Code).
30
31 +!create_obj_art(ArtName,In)
32     <- .term2string(In,InS);
33     makeArtifact(ArtName, "tools.obj", [InS], ArtId);
34     .broadcast(achieve, discover_art(ArtName)).
35 -!create_obj_art(ArtName,In)[error_code(Code)]
36     <- .print("Error creating artifact ", Code).
37
38 +!create_location_art(ArtName,Link,Path)
39     <- .term2string(Link,LinkS);
40     .term2string(Path,PathS);
41     makeArtifact(ArtName, "tools.location", [LinkS, PathS], ArtId);
42     .broadcast(achieve, discover_art(ArtName)).
43 -!create_location_art(ArtName,Link,Path)[error_code(Code)]
44     <- .print("Error creating artifact ", Code).
45
46 +?formationStatus(ok)[artifact_id(G)]
47     <- .wait({+formationStatus(ok)[artifact_id(G)]}).

```

Listing A.9: Driverlog common code.

```

1 processList(L,R) :- .term2string(T1,R) & .term2string(T2,L) & .member(T1,T2).
2
3 // try to find a particular artifact and then focus on it

```

```

4  +!discover_art(ToolName)
5      <- lookupArtifact(ToolName,ToolId);
6          +id(ToolName,ToolId);
7          focus(ToolId).
8  // keep trying until the artifact is found
9  -!discover_art(ToolName)
10     <- .wait(100);
11         !discover_art(ToolName).
12
13  /* Organisational Plans Required by all agents */
14
15  +!in_ora4mas : in_ora4mas.
16  +!in_ora4mas : .intend(in_ora4mas)
17     <- .wait({+in_ora4mas},100,_);
18         !in_ora4mas.
19  @lin[atomic]
20  +!in_ora4mas
21     <- joinWorkspace("ora4mas",_);
22         +in_ora4mas.
23
24  // plans to handle obligations
25  // obligation to commit to a mission
26  +obligation(Ag,Norm,committed(Ag,Mission,Scheme),Deadline)
27     : .my_name(Ag)
28     <- println("I am obliged to commit to ",Mission," on ",Scheme,"... doing so");
29         commitMission(Mission)[artifact_name(Scheme)].
30  // obligation to achieve a goal
31  +obligation(Ag,Norm,achieved(Scheme,Goal,Ag),Deadline)
32     : .my_name(Ag)
33     <- //println("I am obliged to achieve goal ",Goal);
34         println(" ---> working to achieve ",Goal);
35         !Goal[scheme(Scheme)];
36         println(" <--- done");
37         goalAchieved(Goal)[artifact_name(Scheme)].
38  // an unknown type of obligation was received
39  +obligation(Ag,Norm,What,DeadLine)
40     : .my_name(Ag)
41     <- println("I am obliged to ",What,", but I don't know what to do!").

```

Listing A.10: Driverlog driver1 agent.

```

1 { include("common.asl") }

```

```

2
3 at("street2").
4
5 +!join(GroupName)
6   <- !in_ora4mas;
7     lookupArtifact(GroupName, GroupId);
8     adoptRole(driver1)[artifact_id(GroupId)];
9     focus(GroupId);
10      !discover_art("driverlog_sch").
11
12 +!walk2 : V1 = "street2" & V2 = "p12" & id(V1,Id1) & id(V2,Id2) &
13          at(V1) &
14            path(L)[artifact_id(Id1)] & processList(L,V2)
15   <- -at(V1);
16     +at(V2).
17 -!walk2 <- .print("Plan walk2 failed, check solution plan.").
18
19 +!walk4 : V1 = "p12" & V2 = "street1" & id(V1,Id1) & id(V2,Id2) &
20          at(V1) &
21            path(L)[artifact_id(Id1)] & processList(L,V2)
22   <- -at(V1);
23     +at(V2).
24 -!walk4 <- .print("Plan walk4 failed, check solution plan.").

```

Listing A.11: Driverlog driver2 agent.

```

1 { include("common.asl") }
2
3 at("street2").
4
5 +!join(GroupName)
6   <- !in_ora4mas;
7     lookupArtifact(GroupName, GroupId);
8     adoptRole(driver2)[artifact_id(GroupId)];
9     focus(GroupId);
10      !discover_art("driverlog_sch").
11
12
13 +!walk1 : V1 = "street2" & V2 = "p12" & id(V1,Id1) & id(V2,Id2) &
14          at(V1) &
15            path(L)[artifact_id(Id1)] & processList(L,V2)
16   <- -at(V1);

```

```

17     +at(V2).
18 -!walk1 <- .print("Plan walk1 failed, check solution plan.").
19
20 +!walk3 : V1 = "p12" & V2 = "street1" & id(V1,Id1) & id(V2,Id2) &
21     at(V1) &
22     path(L)[artifact_id(Id1)] & processList(L,V2)
23     <- -at(V1);
24     +at(V2).
25 -!walk3 <- .print("Plan walk3 failed, check solution plan.").
26
27 +!walk5 : V1 = "street1" & V2 = "p10" & id(V1,Id1) & id(V2,Id2) &
28     at(V1) &
29     path(L)[artifact_id(Id1)] & processList(L,V2)
30     <- -at(V1);
31     +at(V2).
32 -!walk5 <- .print("Plan walk5 failed, check solution plan.").
33
34 +!walk6 : V1 = "p10" & V2 = "street0" & id(V1,Id1) & id(V2,Id2) &
35     at(V1) &
36     path(L)[artifact_id(Id1)] & processList(L,V2)
37     <- -at(V1);
38     +at(V2).
39 -!walk6 <- .print("Plan walk6 failed, check solution plan.").
40
41 +!board1 : V1 = "truck1" & V2 = "street0" & id(V1,Id1) & id(V2,Id2) &
42     at(V2) &
43     pos(L)[artifact_id(Id1)] & processList(L,V2) &
44     empty(E)[artifact_id(Id1)] & E
45     <- -at(V2);
46     +at(V1);
47     updateEmpty(false)[artifact_id(Id1)].
48 -!board1 <- .print("Plan board1 failed, check solution plan.").
49
50 +!drive1 : V1 = "truck1" & V2 = "street0" & V3 = "street1"& id(V1,Id1) &
51     id(V2,Id2) & id(V3,Id3) &
52     at(V1) &
53     pos(P)[artifact_id(Id1)] & processList(P,V2) &
54     link(L)[artifact_id(Id2)] & processList(L,V3)
55     <- updatePos(V3)[artifact_id(Id1)].
56 -!drive1 <- .print("Plan drive1 failed, check solution plan.").

```

B. Depots domain codes

Listing B.1: Depots MAP-POP PDDL domain for location agent.

```

1 (define (domain depot)
2 (:requirements :typing :equality :fluents)
3 (:types place hoist surface - object
4     depot distributor - (either place agent)
5     truck - agent
6     crate pallet - surface)
7 (:predicates
8   (myAgent ?a - place)
9   (clear ?x - (either surface hoist)))
10 (:functions
11   (located ?h - hoist) - place
12   (at ?t - truck) - place
13   (placed ?p - pallet) - place
14   (pos ?c - crate) - (either place truck)
15   (on ?c - crate) - (either surface hoist truck))
16 (:action LiftP
17   :parameters (?h - hoist ?c - crate ?z - pallet ?p - place)
18   :precondition (and (myAgent ?p) (= (located ?h) ?p)
19                   (= (placed ?z) ?p) (clear ?h) (= (pos ?c) ?p)
20                   (= (on ?c) ?z) (clear ?c))
21   :effect (and (assign (on ?c) ?h) (not (clear ?c)) (not (clear ?h))
22              (clear ?z)))
23 (:action LiftC
24   :parameters (?h - hoist ?c - crate ?z - crate ?p - place)
25   :precondition (and (myAgent ?p) (= (located ?h) ?p)
26                   (= (pos ?z) ?p) (clear ?h)
27                   (= (pos ?c) ?p) (= (on ?c) ?z) (clear ?c))
28   :effect (and (assign (on ?c) ?h) (not (clear ?c)) (not (clear ?h))
29              (clear ?z)))
30 (:action DropP
31   :parameters (?h - hoist ?c - crate ?z - pallet ?p - place)
32   :precondition (and (myAgent ?p) (= (located ?h) ?p)
33                   (= (placed ?z) ?p) (clear ?z) (= (on ?c) ?h)
34                   (not (clear ?c)) (not (clear ?h)))
35   :effect (and (clear ?h) (clear ?c) (not (clear ?z))
36              (assign (on ?c) ?z)))

```

```

37 (:action DropC
38   :parameters (?h - hoist ?c - crate ?z - crate ?p - place)
39   :precondition (and (myAgent ?p) (= (located ?h) ?p)
40                     (= (pos ?z) ?p) (clear ?z)
41                     (= (on ?c) ?h) (not (clear ?c))
42                     (not (clear ?h)))
43   :effect (and (clear ?h) (clear ?c) (not (clear ?z))
44              (assign (on ?c) ?z)))
45 )

```

Listing B.2: Depots MAP-POP PDDL domain for truck agent.

```

1  (define (domain depot)
2  (:requirements :typing :equality :fluents)
3  (:types place hoist surface - object
4        depot distributor - (either place agent)
5        truck - agent
6        crate pallet - surface)
7  (:predicates
8    (myAgent ?a - truck)
9    (clear ?x - (either surface hoist)))
10 (:functions
11   (located ?h - hoist) - place
12   (at ?t - truck) - place
13   (placed ?p - pallet) - place
14   (pos ?c - crate) - (either place truck)
15   (on ?c - crate) - (either surface hoist truck))
16 (:action Drive
17   :parameters (?t - truck ?x ?y - place)
18   :precondition (and (myAgent ?t) (= (at ?t) ?x))
19   :effect (and (assign (at ?t) ?y)))
20 (:action Load
21   :parameters (?h - hoist ?c - crate ?t - truck ?p - place)
22   :precondition (and (myAgent ?t) (= (at ?t) ?p) (= (pos ?c) ?p)
23                 (not (clear ?c)) (not (clear ?h))
24                 (= (on ?c) ?h) (= (located ?h) ?p))
25   :effect (and (clear ?h) (clear ?c) (assign (pos ?c) ?t)
26              (assign (on ?c) ?t)))
27 (:action Unload
28   :parameters (?h - hoist ?c - crate ?t - truck ?p - place)
29   :precondition (and (myAgent ?t) (= (located ?h) ?p) (= (at ?t) ?p)
30                 (= (pos ?c) ?t) (= (on ?c) ?t) (clear ?h)

```

```

31         (clear ?c))
32   :effect (and (assign (pos ?c) ?p) (assign (on ?c) ?h)
33             (not (clear ?c)) (not (clear ?h))))
34 )

```

Listing B.3: Depots MAP-POP PDDL problem.

```

1 (define (problem depotprob1818)
2   (:domain depot)
3   (:objects
4     depot0 - depot
5     distributor0 distributor1 - distributor
6     truck0 truck1 - truck
7     crate0 crate1 - crate
8     pallet0 pallet1 pallet2 - pallet
9     hoist0 hoist1 hoist2 - hoist
10  )
11  (:shared-data
12    (clear ?x - (either surface hoist))
13    ((at ?t - truck) - place)
14    ((pos ?c - crate) - (either place truck))
15    ((on ?c - crate) - (either surface hoist truck)) -
16    (either distributor0 distributor1 truck0 truck1)
17  )
18  (:init
19    (myAgent depot0)
20    (= (pos crate0) distributor0)
21    (clear crate0)
22    (= (on crate0) pallet1)
23    (= (pos crate1) depot0)
24    (clear crate1)
25    (= (on crate1) pallet0)
26    (= (at truck0) distributor1)
27    (= (at truck1) depot0)
28    (= (located hoist0) depot0)
29    (clear hoist0)
30    (= (located hoist1) distributor0)
31    (clear hoist1)
32    (= (located hoist2) distributor1)
33    (clear hoist2)
34    (= (placed pallet0) depot0)
35    (not (clear pallet0))

```

```

36 (= (placed pallet1) distributor0)
37 (not (clear pallet1))
38 (= (placed pallet2) distributor1)
39 (clear pallet2)
40 )
41 (:global-goal (and
42 (= (on crate0) pallet2)
43 (= (on crate1) pallet1)
44 ))

```

Listing B.4: Depots JaCaMo project file.

```

1 MAS depots {
2   infrastructure: JaCaMo
3   agents:
4     init;
5     depot0;
6       distributor0;
7       distributor1;
8       truck1;
9       truck2;
10  aslSourcePath: "src/asl";
11 }

```

Listing B.5: Depots Moise organisation.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <organisational-specification
3   id="depots"
4   os-version="0.8"
5
6   xmlns='http://moise.sourceforge.net/os'
7   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
8   xsi:schemaLocation='http://moise.sourceforge.net/os
9     http://moise.sourceforge.net/xml/os.xsd' >
10 <structural-specification>
11 <role-definitions>
12 <role id="depot" />
13 <role id="depot0" > <extends role="depot"/> </role>
14 <role id="distributor" />
15 <role id="distributor0" > <extends role="distributor"/> </role>
16 <role id="distributor1" > <extends role="distributor"/> </role>

```



```

17 <role id="truck" />
18   <role id="truck1" > <extends role="truck"/> </role>
19   <role id="truck2" > <extends role="truck"/> </role>
20 </role-definitions>
21 <group-specification id="depots">
22   <roles>
23     <role id="depot0"   min="1" max="1"/>
24     <role id="distributor0"   min="1" max="1"/>
25     <role id="distributor1"   min="1" max="1"/>
26     <role id="truck1"   min="1" max="1"/>
27     <role id="truck2"   min="1" max="1"/>
28   </roles>
29 </group-specification>
30 </structural-specification>
31 <functional-specification>
32   <scheme id="depots_sch">
33     <goal id="goal">
34       <plan operator="sequence">
35         <goal id="par1">
36           <plan operator="parallel">
37             <goal id="lift1" />
38             <goal id="lift2" />
39             <goal id="drive1" />
40           </plan>
41         </goal>
42         <goal id="par2">
43           <plan operator="parallel">
44             <goal id="load1" />
45             <goal id="load2" />
46           </plan>
47         </goal>
48         <goal id="par3">
49           <plan operator="parallel">
50             <goal id="drive2" />
51             <goal id="drive3" />
52           </plan>
53         </goal>
54         <goal id="par4">
55           <plan operator="parallel">
56             <goal id="unload1" />
57             <goal id="unload2" />

```

```

58     </plan>
59 </goal>
60 <goal id="par5">
61     <plan operator="parallel">
62         <goal id="drop1" />
63         <goal id="drop2" />
64     </plan>
65 </goal>
66 </plan>
67 </goal>
68 <mission id="mdepot0" min="1" max="1">
69     <goal id="lift2" />
70 </mission>
71 <mission id="mdistributor0" min="1" max="1">
72     <goal id="lift1"/>
73     <goal id="drop2" />
74 </mission>
75 <mission id="mdistributor1" min="1" max="1">
76     <goal id="drop1"/>
77 </mission>
78 <mission id="mtruck1" min="1" max="1">
79     <goal id="load2"/>
80     <goal id="drive3" />
81     <goal id="unload2" />
82 </mission>
83 <mission id="mtruck2" min="1" max="1">
84     <goal id="drive1"/>
85     <goal id="load1" />
86     <goal id="drive2" />
87     <goal id="unload1" />
88 </mission>
89 </scheme>
90 </functional-specification>
91 <normative-specification>
92 <norm id="n1" type="obligation" role="depot0" mission="mdepot0" />
93 <norm id="n2" type="obligation" role="distributor0" mission="mdistributor0" />
94 <norm id="n3" type="obligation" role="distributor1" mission="mdistributor1" />
95 <norm id="n4" type="obligation" role="truck1" mission="mtruck1" />
96 <norm id="n5" type="obligation" role="truck2" mission="mtruck2" />
97 </normative-specification>
98 </organisational-specification>

```

Listing B.6: Depots hoist artefact.

```

1 package tools;
2
3 import cartago.Artifact;
4 import cartago.OPERATION;
5 import cartago.ObsProperty;
6
7 public class hoist extends Artifact {
8     @OPERATION public void init(Boolean newClear, String newLocated) {
9         defineObsProperty("clear", newClear);
10        defineObsProperty("located", newLocated);
11    }
12    @OPERATION public void updateClear(Boolean newClear) {
13        ObsProperty opClear = getObsProperty("clear");
14        opClear.updateValue(newClear);
15    }
16    @OPERATION public void updateLocated(String newLocated) {
17        ObsProperty opLocated = getObsProperty("located");
18        opLocated.updateValue("[ "+newLocated+" ]");
19    }
20
21 }

```

Listing B.7: Depots surface artefact.

```

1 package tools;
2
3 import cartago.Artifact;
4 import cartago.OPERATION;
5 import cartago.ObsProperty;
6
7 public class surface extends Artifact {
8     @OPERATION public void init(String newPos, String newOn, String newPlaced,
9         Boolean newClear) {
10        defineObsProperty("pos", newPos);
11        defineObsProperty("on", newOn);
12        defineObsProperty("placed", newPlaced);
13        defineObsProperty("clear", newClear);
14    }
15
16    @OPERATION public void updatePos(String newPos) {
17        ObsProperty opPos = getObsProperty("pos");

```

```

18         opPos.updateValue("[+newPos+");
19     }
20     @OPERATION public void updateOn(String newOn) {
21         ObsProperty opOn = getObsProperty("on");
22         opOn.updateValue("[+newOn+");
23     }
24     @OPERATION public void updatePlaced(String newPlaced) {
25         ObsProperty opPlaced = getObsProperty("placed");
26         opPlaced.updateValue("[+newPlaced+");
27     }
28     @OPERATION public void updateClear(Boolean newClear) {
29         ObsProperty opClear = getObsProperty("clear");
30         opClear.updateValue(newClear);
31     }
32 }

```

Listing B.8: Depots init agent.

```

1  !start.
2
3  +!start <- !create_surface_art("crate0", [distributor0], [pallet1], [], true);
4             !create_surface_art("crate1", [depot0], [pallet0], [], true);
5             !create_surface_art("pallet0", [], [], [depot0], false);
6             !create_surface_art("pallet1", [], [], [distributor0], false);
7             !create_surface_art("pallet2", [], [], [distributor1], true);
8             !create_hoist_art("hoist0", true, [depot0]);
9             !create_hoist_art("hoist1", true, [distributor0]);
10            !create_hoist_art("hoist2", true, [distributor1]);
11            createWorkspace("ora4mas");
12            joinWorkspace("ora4mas",_);
13            makeArtifact("depots","ora4mas.nopl.GroupBoard",
14                ["src/depots-os.xml", depots, false, true],GrArtId);
15            focus(GrArtId);
16            makeArtifact("depots_sch", "ora4mas.nopl.SchemeBoard",
17                ["src/depots-os.xml", depots_sch, false, true], SchArtId);
18            .broadcast(achieve, join("depots"));
19            focus(SchArtId);
20            ?formationStatus(ok)[artifact_id(GrArtId)];
21            addScheme("depots_sch")[artifact_id(GrArtId)].
22
23  +!create_surface_art(ArtName,Pos,On,Placed,Clear)
24  <- .term2string(Pos,PosS);

```

```

25     .term2string(On,OnS);
26     .term2string(Placed,PlacedS);
27     makeArtifact(ArtName, "tools.surface", [PosS,OnS,PlacedS,Clear], ArtId);
28     .broadcast(achieve, discover_art(ArtName)).
29 -!create_surface_art(ArtName,Pos,On,Placed,Clear)[error_code(Code)]
30     <- .print("Error creating artifact ", Code).
31
32 +!create_hoist_art(ArtName,Clear,Located)
33     <- .term2string(Located,LocatedS);
34     makeArtifact(ArtName, "tools.hoist", [Clear, LocatedS], ArtId);
35     .broadcast(achieve, discover_art(ArtName)).
36 -!create_hoist_art(ArtName,Clear,Located)[error_code(Code)]
37     <- .print("Error creating artifact ", Code).
38
39 +?formationStatus(ok)[artifact_id(G)]
40     <- .wait({+formationStatus(ok)[artifact_id(G)]}).

```

Listing B.9: Depots common code.

```

1 processList(L,R) :- .term2string(T1,R) & .term2string(T2,L) & .member(T1,T2).
2
3 // try to find a particular artifact and then focus on it
4 +!discover_art(ToolName)
5     <- lookupArtifact(ToolName,ToolId);
6         +id(ToolName,ToolId);
7         focus(ToolId).
8 // keep trying until the artifact is found
9 -!discover_art(ToolName)
10    <- .wait(100);
11        !discover_art(ToolName).
12
13 /* Organisational Plans Required by all agents */
14
15 +!in_ora4mas : in_ora4mas.
16 +!in_ora4mas : .intend(in_ora4mas)
17     <- .wait({+in_ora4mas},100,_);
18     !in_ora4mas.
19 @lin[atomic]
20 +!in_ora4mas
21     <- joinWorkspace("ora4mas",_);
22     +in_ora4mas.
23

```

```

24 // plans to handle obligations
25 // obligation to commit to a mission
26 +obligation(Ag, Norm, committed(Ag, Mission, Scheme), Deadline)
27     : .my_name(Ag)
28     <- println("I am obliged to commit to ", Mission, " on ", Scheme, "... doing so");
29         commitMission(Mission)[artifact_name(Scheme)].
30 // obligation to achieve a goal
31 +obligation(Ag, Norm, achieved(Scheme, Goal, Ag), Deadline)
32     : .my_name(Ag)
33     <- //println("I am obliged to achieve goal ", Goal);
34         println(" ---> working to achieve ", Goal);
35         !Goal[scheme(Scheme)];
36         println(" <--- done");
37         goalAchieved(Goal)[artifact_name(Scheme)].
38 // an unknown type of obligation was received
39 +obligation(Ag, Norm, What, DeadLine)
40     : .my_name(Ag)
41     <- println("I am obliged to ", What, ", but I don't know what to do!").

```

Listing B.10: Depots truck1 agent.

```

1 { include("common.asl") }
2
3 at("depot0").
4
5 +!join(GroupName)
6     <- !in_ora4mas;
7         lookupArtifact(GroupName, GroupId);
8         adoptRole(truck1)[artifact_id(GroupId)];
9         focus(GroupId);
10            !discover_art("depots_sch").
11
12
13 +!load2 : V1 = "hoist0" & V2 = "crate1" & V3 = "truck1" & V4 = "depot0" &
14         id(V1, Id1) & id(V2, Id2) &
15         at(V4) &
16         pos(Pos1)[artifact_id(Id2)] & processList(Pos1, V4) &
17         clear(Clear1)[artifact_id(Id2)] & (not Clear1) &
18         clear(Clear2)[artifact_id(Id1)] & (not Clear2) &
19         on(On1)[artifact_id(Id2)] & processList(On1, V1)
20     <- updateClear(true)[artifact_id(Id1)];
21         updateClear(true)[artifact_id(Id2)];

```

```

22     updatePos(V3)[artifact_id(Id2)];
23     updateOn(V3)[artifact_id(Id2)].
24 -!load2 <- .print("Plan load2 failed, check solution plan.").
25
26 +!drive3 : V1 = "truck1" & V2 = "depot0" & V3 = "distributor0" &
27     at(V2)
28     <- -at(V2);
29     +at(V3).
30 -!drive3 <- .print("Plan drive3 failed, check solution plan.").
31
32 +!unload2 : V1 = "hoist1" & V2 = "crate1" & V3 = "truck1" & V4 = "distributor0" &
33     id(V1,Id1) & id(V2,Id2) &
34     at(V4) &
35     pos(Pos1)[artifact_id(Id2)] & processList(Pos1,V3) &
36     on(On1)[artifact_id(Id2)] & processList(On1,V3) &
37     clear(Clear1)[artifact_id(Id1)] & Clear1 &
38     clear(Clear2)[artifact_id(Id2)] & Clear2
39     <- updatePos(V4)[artifact_id(Id2)];
40     updateOn(V1)[artifact_id(Id2)];
41     updateClear(false)[artifact_id(Id2)];
42     updateClear(false)[artifact_id(Id1)].
43 -!unload2 <- .print("Plan unload2 failed, check solution plan.").

```

Listing B.11: Depots truck2 agent.

```

1 { include("common.asl" ) }
2
3 at("distributor1").
4
5 +!join(GroupName)
6     <- !in_ora4mas;
7     lookupArtifact(GroupName, GroupId);
8     adoptRole(truck2)[artifact_id(GroupId)];
9     focus(GroupId);
10     !discover_art("depots_sch").
11
12 +!drive1 : V1 = "truck2" & V2 = "distributor1" & V3 = "distributor0" &
13     at(V2)
14     <- -at(V2);
15     +at(V3).
16 -!drive1 <- .print("Plan drive1 failed, check solution plan.").
17

```

```

18 +!load1 : V1 = "hoist1" & V2 = "crate0" & V3 = "truck2" & V4 = "distributor0" &
19           id(V1,Id1) & id(V2,Id2) &
20           at(V4) &
21           pos(Pos1)[artifact_id(Id2)] & processList(Pos1,V4) &
22           clear(Clear1)[artifact_id(Id2)] & (not Clear1) &
23           clear(Clear2)[artifact_id(Id1)] & (not Clear2) &
24           on(On1)[artifact_id(Id2)] & processList(On1,V1)
25 <- updateClear(true)[artifact_id(Id1)];
26           updateClear(true)[artifact_id(Id2)];
27           updatePos(V3)[artifact_id(Id2)];
28           updateOn(V3)[artifact_id(Id2)].
29 -!load1 <- .print("Plan load1 failed, check solution plan.").
30
31 +!drive2 : V1 = "truck2" & V2 = "distributor0" & V3 = "distributor1" &
32           at(V2)
33 <- -at(V2);
34           +at(V3).
35 -!drive2 <- .print("Plan drive2 failed, check solution plan.").
36
37 +!unload1 : V1 = "hoist2" & V2 = "crate0" & V3 = "truck2" & V4 = "distributor1" &
38           id(V1,Id1) & id(V2,Id2) &
39           at(V4) &
40           pos(Pos1)[artifact_id(Id2)] & processList(Pos1,V3) &
41           on(On1)[artifact_id(Id2)] & processList(On1,V3) &
42           clear(Clear1)[artifact_id(Id1)] & Clear1 &
43           clear(Clear2)[artifact_id(Id2)] & Clear2
44 <- updatePos(V4)[artifact_id(Id2)];
45           updateOn(V1)[artifact_id(Id2)];
46           updateClear(false)[artifact_id(Id2)];
47           updateClear(false)[artifact_id(Id1)].
48 -!unload1 <- .print("Plan unload1 failed, check solution plan.").

```

Listing B.12: Depots depot0 agent.

```

1 { include("common.asl") }
2
3 +!join(GroupName)
4 <- !in_ora4mas;
5     lookupArtifact(GroupName, GroupId);
6     adoptRole(depot0)[artifact_id(GroupId)];
7     focus(GroupId);
8     !discover_art("depots_sch").

```



```

9
10 +!lift2 : V1 = "hoist0" & V2 = "crate1" & V3 = "pallet0" & V4 = "depot0" &
11         id(V1,Id1) & id(V2,Id2) & id(V3,Id3) &
12         clear(Clear1)[artifact_id(Id1)] & Clear1 &
13         pos(Pos1)[artifact_id(Id2)] & processList(Pos1,V4) &
14         on(On1)[artifact_id(Id2)] & processList(On1,V3) &
15         clear(Clear2)[artifact_id(Id2)] & Clear2
16 <- updateOn(V1)[artifact_id(Id2)];
17     updateClear(false)[artifact_id(Id2)];
18         updateClear(false)[artifact_id(Id1)];
19         updateClear(true)[artifact_id(Id3)].
20 -!lift2 <- .print("Plan lift2 failed, check solution plan.").

```

Listing B.13: Depots distributor0 agent.

```

1 { include("common.asl" ) }
2
3 +!join(GroupName)
4     <- !in_ora4mas;
5         lookupArtifact(GroupName, GroupId);
6         adoptRole(distributor0)[artifact_id(GroupId)];
7         focus(GroupId);
8         !discover_art("depots_sch").
9
10 +!lift1 : V1 = "hoist1" & V2 = "crate0" & V3 = "pallet1" & V4 = "distributor0" &
11         id(V1,Id1) & id(V2,Id2) & id(V3,Id3) &
12         clear(Clear1)[artifact_id(Id1)] & Clear1 &
13         pos(Pos1)[artifact_id(Id2)] & processList(Pos1,V4) &
14         on(On1)[artifact_id(Id2)] & processList(On1,V3) &
15         clear(Clear2)[artifact_id(Id2)] & Clear2
16 <- updateOn(V1)[artifact_id(Id2)];
17     updateClear(false)[artifact_id(Id2)];
18         updateClear(false)[artifact_id(Id1)];
19         updateClear(true)[artifact_id(Id3)].
20 -!lift1 <- .print("Plan lift1 failed, check solution plan.").
21
22 +!drop2 : V1 = "hoist1" & V2 = "crate1" & V3 = "pallet1" & V4 = "distributor0" &
23         id(V1,Id1) & id(V2,Id2) & id(V3,Id3) &
24         clear(Clear1)[artifact_id(Id3)] & Clear1 &
25         on(On1)[artifact_id(Id2)] & processList(On1,V1) &
26         clear(Clear2)[artifact_id(Id2)] & (not Clear2) &
27         clear(Clear3)[artifact_id(Id1)] & (not Clear3)

```

```

28     <- updateClear(true)[artifact_id(Id1)];
29         updateClear(true)[artifact_id(Id2)];
30         updateClear(false)[artifact_id(Id3)];
31         updateOn(V3)[artifact_id(Id2)].
32 -!drop2 <- .print("Plan drop2 failed, check solution plan.").

```

Listing B.14: Depots distributor1 agent.

```

1 { include("common.asl" )
2
3 +!join(GroupName)
4     <- !in_ora4mas;
5         lookupArtifact(GroupName, GroupId);
6         adoptRole(distributor1)[artifact_id(GroupId)];
7         focus(GroupId);
8         !discover_art("depots_sch").
9
10 +!drop1 : V1 = "hoist2" & V2 = "crate0" & V3 = "pallet2" & V4 = "distributor1" &
11         id(V1,Id1) & id(V2,Id2) & id(V3,Id3) &
12         clear(Clear1)[artifact_id(Id3)] & Clear1 &
13         on(On1)[artifact_id(Id2)] & processList(On1,V1) &
14         clear(Clear2)[artifact_id(Id2)] & (not Clear2) &
15         clear(Clear3)[artifact_id(Id1)] & (not Clear3)
16     <- updateClear(true)[artifact_id(Id1)];
17         updateClear(true)[artifact_id(Id2)];
18         updateClear(false)[artifact_id(Id3)];
19         updateOn(V3)[artifact_id(Id2)].
20 -!drop1 <- .print("Plan drop1 failed, check solution plan.").

```