PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

MODEL-DRIVEN ENGINEERING OF MULTI-AGENT SYSTEMS BASED ON ONTOLOGY

ARTUR LUIZ SILVA DA CUNHA FREITAS

Dissertation submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Renata Vieira
Co-Advisor: Prof. Rafael H. Bordini

Porto Alegre
2017

REPLACE THIS PAGE WITH
THE LIBRARY CATALOG
PAGE

REPLACE THIS PAGE WITH
THE COMMITTEE FORMS

ACKNOWLEDGMENTS

This journey would not have been possible without the support of my family, professors, and friends. To my family, thank you for encouraging me in my pursuits, inspiring me to follow my dreams and wanting the best for me. To my professors, thank you for showing me what it means to be a dedicated professional, pushing my limits and believing in me. To my advisor and co-advisor, receiving your guidances and working together to achieve such amazing results that we made together were a pleasure and an honor for me. To my friends, thank you for listening, offering me advice, and supporting me through this process. Lastly but not least, I am grateful for my girlfriend for all the special moments together.

Several achievements made me very proud during the course of my doctorate degree. I had the opportunity to participate in a research project sponsored by a multi-national company, which resulted in a registered patent and several papers. Then, I had the opportunity of presenting my work in international conferences and joining (summer and workshop) schools in the main areas of my doctorate degree. I am glad for integrating the PUCRS team in the Multi-Agent Programming Contest that was awarded with the first place in the competition. Also, I had the excellent opportunity of acting as teacher assistant during one year in the course of “Logic for Computing” for undergraduate students. All of these amazing experiences were of great learning for me.

The following are the acknowledgements to those who funded or were in any way important at least in part for the accomplishment of this work:

- We are grateful for the support given by CNPq and CAPES.
- Part of the results were obtained through research on a project titled “Semantic and Multi-Agent Technologies for Group Interaction”, sponsored by Samsung Eletrônica da Amazônia Ltda. under the terms of Brazilian federal law No. 8.248/91.
- Several open source solutions are acknowledged, such as JaCaMo, Eclipse, OWL API, Protégé, Pellet, Java, etc.

ENGENHARIA DIRIGIDA A MODELOS DE SISTEMAS MULTIAGENTES BASEADOS EM ONTOLOGIA

RESUMO

A engenharia orientada a modelos fornece abstrações e notações para melhorar a compreensão e para apoiar a modelagem, codificação e verificação de aplicações em domínios específicos. As ontologias, por outro lado, fornecem definições formais e explícitas de conceitualizações compartilhadas e permitem o uso de raciocínio semântico. Embora essas áreas tenham sido desenvolvidas por diferentes comunidades, sinergias importantes podem ser alcançadas quando ambas são combinadas. Essas vantagens podem ser exploradas no desenvolvimento de sistemas multi-agentes, dada a sua complexidade e a necessidade de integrar vários componentes que são frequentemente abordados de diferentes ângulos. Este trabalho investiga como aplicar ontologias para engenharia de software orientada a agentes. Inicialmente, apresentamos uma nova abordagem de modelagem onde os sistemas multi-agentes são projetados usando a ontologia OntoMAS proposta. Então, descrevemos técnicas, implementadas em uma ferramenta, para ajudar os programadores a trazer seus conceitos em código e também gerar código automaticamente a partir de modelos instanciados da ontologia. Várias vantagens podem ser obtidas a partir dessas novas abordagens para modelar e codificar sistemas multi-agentes, como o raciocínio semântico para realizar inferências e mecanismos de verificação. Mas a principal vantagem é a linguagem de especificação unificada de alto nível (conhecimento) que permite modelar as três dimensões que estão unidas em JaCaMo para que as especificações dos sistemas possam ser melhor comunicadas entre equipes em desenvolvimento. As avaliações dessas propostas indicam que elas contribuem com os diferentes aspectos da engenharia de software orientada a agentes, como a especificação, verificação e programação desses sistemas.

Palavras-Chave: Ontologia, Sistema Multiagente, Engenharia Dirigida a Modelos, Engenharia de Software Orientada a Agentes, JaCaMo, OntoMAS, OntoJaCaMo.

MODEL-DRIVEN ENGINEERING OF MULTI-AGENT SYSTEMS BASED ON ONTOLOGY

ABSTRACT

Model-driven engineering provides abstractions and notations for improving the understanding and for supporting the modelling, coding, and verification of applications for specific domains. Ontologies, on the other hand, provide formal and explicit definitions of shared conceptualisations and enable the use of semantic reasoning. Although these areas have been developed by different communities, important synergies can be achieved when both are combined. These advantages can be explored in the development of multi-agent systems, given their complexity and the need for integrating several components that are often addressed from different angles. This work investigates how to apply ontologies for agent-oriented software engineering. Initially, we present a new modelling approach where multi-agent systems are designed using the proposed OntoMAS ontology. Then, we describe techniques, implemented in a tool, to help programmers bring their concepts into code and also generate code automatically from instantiated ontology models. Several advantages can be obtained from these new approaches to model and code multi-agent systems, such as semantic reasoning to carry out inferences and verification mechanisms. But the main advantage is the unified high (knowledge) level specification language that allows modelling the three dimensions that are united in the JaCaMo framework so that systems specifications can be better communicated across developing teams. The evaluations of these proposals indicate that they contribute with the different aspects of agent-oriented software engineering, such as the specification, verification, and programming of these systems.

Keywords: Ontology, Multi-Agent System, Model-Driven Engineering, Agent-Oriented Software Engineering, JaCaMo, OntoMAS, OntoJaCaMo.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AOP – Agent-Oriented Programming
AOSE – Agent-Oriented Software Engineering
API – Application Programming Interface
BDI – Belief-Desire-Intention
DL – Description Logics
EOP – Environment-Oriented Programming
FIPA – Foundation for Intelligent Physical Agents
AI – Artificial Intelligence
IDE – Integrated Development Environment
MAOP – Multi-Agent Oriented Programming
MAPC – Multi-Agent Programming Contest
MAS – Multi-Agent System
MDE – Model-Driven Engineering
OOP – Organisation-Oriented Programming
OWL – Web Ontology Language
PDT – Prometheus Design Tool
SE – Software Engineering
SWRL – Semantic Web Rule Language
UML – Unified Modeling Language
XML – eXtensible Markup Language

CONTENTS

1. INTRODUCTION

> *"A journey of a thousand miles begins with a single step."*
>
> — Lao Tzu (604 BC - 531 BC)

Model-Driven Engineering (MDE) employs *models* as the cornerstone of software development processes [GNFC12] in order to improve productivity, portability, interoperability, maintenance, and so on. MDE is a research area that provides abstractions and notations to improve the understanding and to support the modelling of applications for specific domains. These advantages can be employed in the development of Multi-Agent Systems (MAS) given their complexity and the need for integrating several components that are often addressed from different angles [FHM$^+$15]. For example, the JaCaMo [BBH$^+$13] framework for MAS programming combines three separate technologies: Jason [BHW07] for coding the dimension of autonomous agents in AgentSpeak, CArtAgO [RVO06] for programming the environment as artifacts in Java, and Moise [HBKR10] for specifying MAS organisations in XML. JaCaMo is claimed to be the first successful combination of Agent-Oriented Programming (AOP), Organisation-Oriented Programming (OOP), and Environment-Oriented Programming (EOP) in a specific programming platform [BBH$^+$13]. In this context, programmers have three distinct starting points to code a MAS, making appropriate to have a single, unified and comprehensive meta-model combining all these dimensions [FBV17]. Besides filling the gaps between design and development, this modelling framework can be the basis of features such as code generation, support during programming, and reasoning to analyse a given system implementation.

In this context, the use of models is present in most agent methodologies [GNFC12] and MDE techniques for Agent-Oriented Software Engineering (AOSE) emerges naturally. A typical example of MDE for MAS is Prometheus [PW03]; however, differently from our work, the authors do so without exploring any use of formal ontologies as part of such models. In fact, there are already modelling frameworks for more than one dimension of MAS [PW03, GNFC12, UH14], but without using formal ontologies, semantic reasoning, or employing the model during the programming step. In literature, when an ontology is used to model a MAS, only a part is modelled, such as the environment [OVBdRC06] or the organisation [Zar12]. Currently, the use of ontologies to model integrated frameworks that consider the co-specification of different MAS dimensions is still an open issue [FSP$^+$15]. We observe two additional points: *(i)* MDE and ontologies share a number of principles and goals; and *(ii)* there is consistent work in combining ontologies and MAS. These synergies led us to propose and investigate the use of ontology for MDE of MAS, which has resulted in a model-based approach that is pioneer in simultaneously covers all these issues. This idea was derived when considering research comparing ontologies with

MDE [AGK06, KKK$^+$06, SWGP10], and also observing that when ontologies are used in MAS it is usually for purposes others than modelling [MVBH05, KB08, MAB$^+$14]. Moreover, we propose a tool to load an ontological instantiation in order to generate code for the different MAS platforms that are part of JaCaMo [BBH$^+$13]. As an analogy, the ontology proposed in this work acts as a meta-model, and its instantiation represents a model, which in our case is applied to specify a MAS implementation project for JaCaMo.

Ontologies can offer significant benefits for MAS, such as for example in terms of interoperability, reusability, support for MAS development activities (e.g., system analysis and agent knowledge modelling), new features for MAS operation (e.g., agent communication and reasoning), and so on [TL08]. Also, ontologies may be used at different stages within AOSE [HWDC09], so as to: enable decomposition of the overall problem, support the process of information retrieval and reuse, support the process of analysing and manipulating information, and enable communication between cooperatively working agents. However, we are not aware of any previous exploitation of ontologies and their benefits in the global modelling of MAS where such model is used also in a tool during the programming phase.

We started our investigation by systematically surveying the literature for current approaches and advantages that ontologies can provide for MAS. Then, we explored uses and benefits of applying ontology in the global modelling of MAS where such framework includes techniques to offer support also for the programming phase. A particular feature of our approach is that we are proposing the use of ontology as the basis for modelling, development, and verification of MAS. To the best of our knowledge, there is no previous work that has achieved all that. Therefore, the main contributions of this thesis are threefold. First, we are arguing that the use of ontologies improves the modelling, programming and verification of MAS, which emerges from our analysis of the state of the art research directions on the integration of MDE, ontologies, and MAS [FSP$^+$15, FBMV15, FCVB16]. Second, we are proposing and investigating an ontological modelling approach [FBV17] that considers the global characteristics of MAS, as it incorporates the agent, the environment, and the organisation dimensions. This approach covers MAS design and development as a whole in an integrated formalism. Also, using this ontology naturally enables the executions of semantic reasoners which provide inference mechanisms that can be used for verification and validation on the designed models. Third, we describe our techniques that are implemented in a tool [FHM$^+$15] for using MDE to support MAS programming based on a MAS designed with our proposed ontology modelling approach. This new platform provides features such as drag-and-drop and auto-complete from ontologies to MAS code in a plug-in for the Eclipse [Bud04] Integrated Development Environment (IDE).

Some of the key issues in developing MAS are: *(i)* techniques for integrating design and code; *(ii)* extension of agent-oriented programming languages to cover certain aspects that are currently weak or missing (e.g., social concepts, and modelling the environment); and *(iii)* development of debugging and verification techniques, with a particular focus on using model checking in testing and debugging, and applying model checking to design artifacts [BDW06]. Our research directly addresses two of these problems. First, it contributes with the integration of design and code as we demonstrate in our approach for generating code from models represented as ontological specifications. Second, our approach enables features of semantic reasoning and verification techniques on top of instantiated ontology-based models. Mechanisms for solving the aforementioned issues are claimed to be crucial for the practical adoption and deployment of agent technology [BDW06]. Thus, the investigation of software development methodologies should provide interesting answers, solutions and improvements for problems in AOSE. This thesis contributes with this point by means of proposing and evaluating the use of ontology-based techniques to support MAS modelling and programming, in which we refer to as OntoMAS and OntoJaCaMo.

1.1 Motivation

The development of MAS, like developing any software system, encompasses activities traditionally classified into three broad areas: software engineering (e.g., requirements elicitation, analysis, design), implementation (using some suitable programming language), and verification/validation [BDW06]. In current practice, the way in which a MAS is typically built is that the developer designs the agent organisation and the individual agents, then takes the detailed design and manually codes the agents in some programming language. The issue with developing the implementation manually from the design is that this creates the possibility for divergences between design and implementation, which makes the design less useful for further work in maintenance and comprehension [BDW06]. Thus, it would be optimal to have code and design being seen as different views on what is really a single conceptual activity [FBV17]. The key action to be done with respect to this is developing techniques and tools that allow for design and code to be strongly integrated with consistency checking and change propagation. Among the possible approaches for mitigating this “gap” between code and design there are: *(i)* the generation of code from design; and *(ii)* the extraction of changes in design/code to apply in the other [BDW06]. To consider useful the generation of code from design, techniques are required to ensure the continued consistency of design and code when one or the other is changed. However, such techniques must be developed to link one particular design notation with one target programming language [BDW06]. This thesis focuses on the proposed OntoMAS ontology for playing the role of design notation, and JaCaMo [BBH$^+$13] as the target programming platform for our code generation techniques that are implemented in the OntoJaCaMo tool.

As design notation, UML (Unified Modeling Language), in its original form, provides insufficient support for modelling MAS [dSdL03], such as, for example, the ones created with JaCaMo [BBH$^+$13]. The development of MAS in JaCaMo comprises three distinct dimensions, namely: agent, environment, and organisation. However, these dimensions are not uniformly integrated into a single formalism: agents are programmed in Jason [BHW07] using the AgentSpeak language; environments are coded in Java using the CArtAgO API [RVO06]; and organisations are specified in Moise [HBKR10] using XML. Some disadvantages of this current approach are that: *(i)* programmers have three distinct starting points to code their MAS; *(ii)* it is difficult to keep track of issues because errors in one level can affect the other levels; *(iii)* it becomes cumbersome to explore interconnections between the different layers; and *(iv)* it requires programmers to have knowledge about different paradigms [FSP$^+$15]. Agent-based systems require adequate techniques that explore their benefits and their peculiar characteristics [dSdL03]. To address these issues, this thesis is proposing a unified semantic model which covers these three MAS programming dimensions and integrates their formalisms. An integrated ontological model that represents these MAS dimensions also enables semantic reasoning and can be used as a common vocabulary in agent-oriented programming. In our thesis these dimensions can interconnect with each other, and reuse relevant concepts from the other MAS dimensions. Each dimension details different aspects, and these interconnections when combined result in an integrated knowledge model with a clear correspondence to an integrated programming platform, such as JaCaMo [BBH$^+$13].

Some aspects of MAS, such as the organisational properties addressed by Moise [HBKR10], are already related to a programming framework. This allows to convert from ontological specifications to a programming level [Zar12], which provides more flexibility for modelling and developing the organisation dimension of MAS. This OWL semantic description of agent organisations also helps agents in becoming aware, querying, and reasoning about their social and organisational context in a uniform way [Zar12]. We have found also an environment ontology [OVBdRC06] based on MAS environment aspects of agent programming technologies. This modelling approach can be used to specify environments and derive a project-level, complete, and executable definition of multi-agent environments. Semantic representations of MAS environments also improve the way agents reason about the objects with which they interact and the overall environment where they are situated [OVBdRC06]. This is important because most agent-oriented programming languages are weak in allowing the developer to model the environment within which the agents will execute [BDW06]. Such modelling frameworks are desirable for all dimensions at the same time, but these levels have to be aligned for that to work as a common specification. This will make possible to model, to reuse, and to extend the MAS in one dimension while maintaining the others, which enables the designer to work without going into specifics of the programming languages that define each dimension. In this context, agent system's designs are more easily expressed and communicated, and the resulting models can be more easily verified and converted to code in agent-oriented programming languages.

We claim that ontologies have an important role for all MAS dimensions and in the whole system development, rather than exclusively in the programming phase [FBV17]. This is in line with CArtAgO development directions [RVO06] in considering ontologies to represent the artifacts, with Moise's recent research which proposes a semantic description of multi-agent organisations [Zar12], and with ontologies of MAS environments [OVBdRC06]. These are however all separate initiatives, whereas in the development of MAS an ontology should interconnect the various specification levels [FBV17]. This allows for an unified view of systems engineering, and should co-exist with integrated agent platforms, such as JaCaMo [BBH$^+$13]. As result, developers obtain a new paradigm for developing complex software systems with a semantic infrastructure that applies principles both from software engineering and knowledge engineering. Unified MAS programming platforms, such as JaCaMo [BBH$^+$13], are being developed with the purpose of helping developers to build these complex solutions, however, such unification must happen during the system design and at the modelling and knowledge level [FBV17]. Thus, this thesis investigates the integration of agent programming platforms by applying ontologies to streamline model-based development MAS in JaCaMo.

1.2 Research Goals

This research focuses on improving MAS modelling and development by exploring the use of ontology and MDE approaches. We formalise this point in one main research goal, and goals that are means of achieving that, as follows:

- **Main research goal:** To elaborate an ontology-based approach in the domain of MAS modelling to support MAS developers and investigate its uses. Sections 4 and 6 show our research in this direction. To address this goal, the following questions must be considered:

  - How to specify/model the design of a MAS using an ontology?

  - How elements from ontologies can be explored to support MAS development?

  - Which tools/benefits/reasoning may be obtained from such ontology representation?

The means of achieving and verifying the achievement of this goal are:

- **Goal 1)** To investigate ontologies and model-based development approaches in AOSE that address questions such as MAS modelling, knowledge representation, and reasoning. Sections 2 and 3 are specially dedicated to achieve this goal, which includes:

  - To review the current literature in the areas of MDE, ontology, and MAS modelling;

  - To analyse and to compare existing solutions in order to point out possible limitations.

- **Goal 2)** To explore and to evaluate the impact of the ontology model and derived techniques and tools in MAS modelling, development, and verification. Sections 5 and 7 are specially dedicated to this goal, which includes:

  - To plan, to execute, and to analyse tests to explore the use of our ontology to support AOSE;

  - To perform experiments to investigate the resulting ontology-based tools;

  - To compare our new approaches in practice against the usual ones in MAS modelling, development, and verification.

Our work aims to achieve a model-based approach using ontology to covers all required dimensions and abstractions of MAS. This thesis differs from other approaches where: *(i)* ontologies are not used; *(ii)* only a part of such systems is modelled in ontologies; or *(iii)* the modelling approach is not integrated with coding or verification mechanisms. In our thesis, an ontology is used for modelling and verification of MAS, and a model-based tool supports the programming of these systems in each of their main dimensions. Examples of advantages derived from such research are techniques for: *(i)* integrating design and code; *(ii)* supporting MAS programming through model-based development; and *(iii)* performing verification with focus on the use of semantic reasoning and model checking. To the best of our knowledge, this work is the first exploitation of ontologies and their benefits in the global modelling and verification of MAS where such model is used also in a tool during the programming phases.

1.3 Thesis Structure

This thesis is structured as follows:

- **Section 2** introduces the background knowledge to support our work, which addresses the areas of MDE, ontologies, and MAS.
- **Section 3** explains related work through the interconnection of MDE, ontologies, and MAS. This literature review gives special emphasis on previous ontologies related to MAS aspects: agents, organisations, and environments.
- **Section 4** presents the definitions and exemplifies the use of our ontology-based MDE approach for MAS modelling through OntoMAS.
- **Section 5** shows our evaluations on this proposed ontology for MAS modelling.
- **Section 6** describes techniques for converting ontological models designed in OntoMAS to programming code and the OntoJaCaMo tool that materialises such techniques.
- **Section 7** illustrates our empirical evaluations of the techniques and tool for ontology-based MAS development.
- **Section 8** presents our final remarks.

2. THEORETICAL BACKGROUND

> *“All things are difficult before they are easy."*
>
> *Thomas Fuller* — (1608 - 1661)

The design of complex systems, such as MAS, should consider models that are clear to communicate, provide support during programming, and allow reuse and reasoning over the specification [FBV17]. Therefore, we propose and investigate the use of ontologies to achieve such goals, as they can also offer code generation features and help in organising the many concepts involved in the modelling, development, and verification of MAS. Since ontology will be playing the role of meta-model for MAS, this Section briefly explains the main topics in the areas of MDE, ontology, and MAS that have led us to this thesis.

2.1 Model-Driven Engineering

Models help us to understand a complex problem and its potential solutions through abstraction [Sel03]. Therefore, it seems obvious that software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modelling techniques. MDE is related to the design and specification of modelling languages [AK03, SWGP10]. In short, MDE provides abstractions and notations for better understanding and easier modelling applications of a specific domain. It is usually applied to: *(i)* produce high-quality results quickly; *(ii)* reuse solutions effectively; *(iii)* specify complex structured information concisely; *(iv)* design rich textual and graphical notations; and *(v)* implement powerful runtime solutions. One of the basic principles of MDE is to consider models as first class entities and any software artifact as a model or as a model element [Béz06]. MDE employs *models* as the cornerstone of software development processes [GNFC12] in order to improve productivity, portability, interoperability, maintenance, and so on. Also, it is possible to gradually evolve an abstract software model into the final product through a process of incremental refinement, without requiring a change in skills, methods, concepts, or tools [Sel03]. However, the models must be formally connected to the actual software to ensure that programmers are following the design decisions captured in a model during implementation [Sel03].

Models are used to reason about a problem domain, design a solution in the solution domain, and they are considered effective if: *(i)* they can serve as a basis for implementing systems; and *(ii)* they make sense from the point of view of a user that is familiar with the domain [Roe12]. To be useful and effective, an engineering model must possess, to a sufficient degree, the following five key characteristics: abstraction, understandability, accuracy, predictiveness, and inexpensiveness [Sel03]. *Abstraction* is almost the only available means of coping with the complexity of the demand for ever-more sophisticated functionality from our software systems. Since a model is always a reduced rendering of the system that it represents, the essence can be more easily understood when details considered irrelevant for a given viewpoint are removed or hidden. *Understandability* is a direct function of the expressiveness of the modelling form used (expressiveness is the capacity to convey a complex idea with little direct information). A model with *accuracy* must provide a true-to-life representation of the modelled system's features of interest. A *predictiveness* model should be able to correctly predict the modelled system's interesting but nonobvious properties, either through experimentation (such as by executing a model on a computer) or through some type of formal analysis. Finally, a model is considered *inexpensive* when it is significantly cheaper to construct and analyse than the modelled system.

One of the most relevant MDE concept is the idea of meta-models, which are models to describe models. Meta-models define general concepts of a given problem domain and their relationships [GNFC12], and their advantage in the development process is the higher abstraction level to work with. This role of meta-models is also played by ontologies, as we highlight next in subsection 2.2. Software models capture relevant characteristics of a software artifact to be developed, yet, most often these software models have no formal semantics, or the underlying (often graphical) software language varies from case to case in a way that makes it hard if not impossible to fix its semantics [SWGP10]. Also, it is not always clear how the concepts used to express the models are mapped to the underlying implementation technologies such as programming language constructs, operating system functions, and so forth [Sel03]. This semantic gap is exacerbated if the modelling language is not precisely defined, leaving room for misinterpretation. Since ontology languages are described by meta-models and allow for describing structural and behavioural models, they provide the capability to be applied as software modelling languages [SWGP10].

This subsection provided several definitions for the most relevant terms backed up by different authors from the MDE community. We also began to introduce concepts from the area of ontology, which is now addressed in the next subsection.

2.2 Ontology

Ontology is defined as an *explicit specification of a conceptualisation* [Gru93], where a *conceptualisation* is an abstract, simplified view of the world that we wish to represent for some purpose. Some essential properties of ontologies are: *(i)* ontologies describe a specific domain[1]; *(ii)* ontology users agree to use the terms consistently; *(iii)* ontology concepts and relations are unambiguously defined in a formal language by axioms and definitions; *(iv)* relationships between ontology concepts determine the ontology structure (e.g., hierarchical or non-hierarchical[2]); and *(v)* ontologies can be understood and processed by computers [HWDC09].

Ontology has different meanings within different contexts, e.g., in Philosophy and in Metaphysics ontology encompasses nature and existence, beings and relation between beings [HWDC09]. The resulting knowledge is explicit, shared, and understood by humans. In Computer Science and in Artificial Intelligence, ontology is used to formally and explicit represent shared domain knowledge through definitions, axioms of concepts, and relationships between concepts [HWDC09]. This work refers to ontology in the context of Computer Science and Artificial Intelligence, in which the main difference is that ontologies are designed to be machine-understandable.

Ontologies are knowledge representation structures, usually based on Description Logics [BHS04], composed of concepts, properties, individuals, relationships, and axioms. A concept (or class) is a collection of objects that share specific restrictions, similarities, or common properties. A property expresses relationships between concepts. An individual (instance, object, or fact) is an element of a concept. A relationship instantiates a property to relate two individuals. Finally, an axiom (or rule) imposes constraints on values of concepts or individuals normally using logic languages (that can be used to check ontological consistency or infer new knowledge).

Both a meta-model and a modelling language are means of describing how a domain could be instantiated. Thus, a model is an instantiation or a specific case that follows an explicit meta-model or modelling language. Ontologies are formalised in a language as well, and they can be created from scratch, or include and extend some others as starting point. An ontology can be seen as a conceptualisation, usually referred as terminological components (TBox). Then, an ontology can be populated/instantiated with facts or assertions, usually called assertion component (ABox), which is associated with a terminological vocabulary. Shortly, TBox statements describe a system in terms of controlled vocabularies, for example, a set of classes and properties; and ABox are TBox-compliant statements about that vocabulary.

Ontology empowers the execution of semantic reasoners that provide functionalities such as *consistency checking*, *concept satisfiability*, *classification*, and *realisation*. In other words, reasoners are able to automatically infer logical consequences from a set of axioms. Pellet [SPG$^+$07] is one example of a semantic reasoner implementation over OWL

---

[1] This definition that was cited claims that an ontology describes a specific domain, which means a part of knowledge in some degree. It does not claim that all ontologies are domain specific, since there are the so called upper level (or foundational) ontologies.

[2] The generalisation/specialisation relationship (i.e., "is-a" relationship) between two concepts is an example of a hierarchical relationship between concepts.

ontologies. OWL (Web Ontology Language) is a language for processing web information and a semantic web standard formalism to explicitly represent the meaning and relationships of terms [BvHH$^+$04].

We consider that these essential properties have a role to play in the process of modelling complex systems, and models based on these underlying properties may be explored in many different ways. More specifically, this thesis is interested in investigating the use of ontologies as an integrated global model for designing and programming MAS. Therefore, we explain the research area of MAS in sequence.

2.3 Multi-Agent Systems Development Platforms

Agents are reactive systems that can independently determine how to best achieve their goals and perform their tasks [BHW07] while demonstrating properties such as autonomy, reactivity, proactiveness, and social ability. Agents are situated in an environment, where they can perceive and modify it, and they should be able to exchange information, cooperate, and coordinate activities. MAS development integrates aspects from different dimensions (e.g., agent, environment, and organisation) that are addressed by different technologies. For example, MAS programming in JaCaMo [BBH$^+$13] requires the development of code in Jason, CArtAgO, and Moise. These three distinct formalisms and starting points for MAS development make desirable a single model combining all MAS's dimensions which can also offer an abstraction to such programming platforms. However, current AOSE models and methodologies (such as Prometheus [PW03]) are deficient in at least one of the following areas of MAS development: agent internal design (design of agent mental constructs such as beliefs, goals, plans, and actions), interaction design (design of interaction protocols and exchanged messages), or organisation modelling (design of acquaintances and authority relationships amongst agents or agents' roles) [TL08]. One problem in this scenario is that such characteristics can appear only hard-coded in MAS programming platforms without being fully and explicit contemplated in high-level models of MAS [FBV17].

The interest in using ontology with agent systems is not recent. For example, the InfoSleuth project in the late 90's used ontologies to model agents and included explicit ontology agents [BBB$^+$97]. Ontologies in InfoSleuth were used to specify both the infrastructure underlying the agent-based architecture and to characterise the information content in the underlying data repositories. According to its authors, their motivations for using ontologies were two-fold: *(i)* capturing and reasoning about information content (e.g., database schema, conceptual models); and *(ii)* specification of the agent infrastructure (e.g., agent configurations, workflow specifications). The ontology agents managed the semantics of the domain/environment in which they operate, and the ontology models for agents describe their knowledge and attributes. One of our main differences is that the ontology models in that paper had been used for interoperability, but not for supporting implementation like the MDE approach in our work. Also, that work used technologies from that time, for example, their agents were implemented using Java, which, currently, is not considered an agent-oriented programming language.

In this work, we focus on the JaCaMo [BBH$^+$13] platform for MAS programming, which integrates: autonomous *agents* coded in Jason [BHW07], shared distributed artifact-based *environments* developed in CArtAgO [RVO06], and *organisations* of agents defined in Moise [HBKR10]. Jason [BHW07] is an AgentSpeak language implementation that focuses on agent actions and mental concepts. Jason is an open source interpreter that offers features such as speech-act based agent communication, plans annotation, architecture customisation, distributed execution, extensibility through internal actions, among other features. On the environment side of agent systems, CArtAgO [RVO06] is a platform to support the artifact notion in MAS. Artifacts are function-oriented computational devices which provide services that agents can exploit to support their individual and social activities [RVO06]. Lastly, the specification of agents at the organisation level can be achieved using an organisation modelling language, such as Moise [HBKR10]. Moise explicitly decomposes the specification of an organisation into its structural, functional, and normative dimensions.

Currently, we have separate approaches for addressing the modelling and programming of MAS, resulting in gaps and conceptual divergences in AOSE [FBV17]. While JaCaMo [BBH$^+$13] is a programming platform that uses three different formalisms for coding MAS, Prometheus [PW03] is an agent modelling approach but without applying or exploring any formal ontologies as part of its technique. Therefore, in this work we investigate an ontology-based MDE approach as an integrated global model of MAS's main characteristics. We also explore ways of using such model to support MAS verification and programming. Although the advantages of ontologies for agents are clear, few MAS platforms currently integrate ontology techniques [TL08, FBV17]. Limited ontological support is provided by the AOSE methodologies since they do not incorporate ontologies throughout the entire systems development life cycle nor consider ways in which ontologies can be used to account for interoperability and verification during design [TL08].

An important contribution of agent-oriented programming as a new paradigm was to provide ways to help programmers to develop autonomous systems [BBH$^+$13]. For example, agent programming languages typically have high-level programming constructs which facilitate (compared to traditional programming languages) the development of systems that are continuously running and reacting to changes in the dynamic environments where such autonomous systems usually operate [BBH$^+$13]. Agent-oriented paradigms are normally used to develop very complex systems, where not only are many autonomous entities present in a shared environment, but also they need to interact in complex ways and need to have social structures and norms to regulate the overall social behaviour that is expected of them [BBH$^+$13].

In this context, JaCaMo was the first approach aimed at explicitly investigating the integration of all the dimensions of MAS from a design and programming point of view: previous existing approaches consider either only the agent–organisation dimensions, or the agent–environment dimensions [BBH$^+$13]. The combination of all these dimensions of MAS into a single programming paradigm with a concrete working platform has had a major impact on the ability to program complex distributed systems [BBH$^+$13]. The authors of JaCaMo pointed out, as future work, the desire for an integrated development environment to facilitate the process of design, development, and execution of JaCaMo applications, potentially reusing and integrating existing Jason, CArtAgO, and Moise tools and technologies [BBH$^+$13]. Thus, recognising the importance achieved by JaCaMo, this research direction is one of the motivations in this thesis. JaCaMo is the unique platform with the three dimensions of MAS completely integrated and functional at the programming level, as far as we known, despite being the first one to achieve all of these [BBH$^+$13]. Thus, our proposed techniques for modelling and code generation address the design of MAS with an eye on implementations using JaCaMo as the target programming platform. For completeness, other frameworks for MAS development are commented next. These other agent programming languages provide some support for environments, or some organisational notions such as roles, but without including a fully fledged organisational model and first-class environment abstractions that are provided by JaCaMo.

The “Organization Oriented Programming Language” (2OPL) [DGMT09] is a rule-based language that allows the programming of multi-agent organisations in terms of norms and which is meant to be exploited in synergy with the agent programming language 2APL. The combination of 2APL with 2OPL covers the dimensions of agent and organisation, however it lacks a clear description of how the organisation integrates with the agent level from a practical programming point of view. Moreover, important abstractions for the environment dimension are missing, since no effective integration between the organisation and environment dimensions is considered.

The Golem [BS08] agent platform allows the programming of both cognitive agents and computational environments that are structured as non-cognitive objects organised into “containers”. In other words, this platform covers the dimensions of agents and environments, but it lacks a clear description of organisations. It also do not explore how the organisations would be integrated from a practical programming point of view. Recent work on that approach is beginning to present the initial steps for extending Golem with norms to realise norm-governed MAS [UBSA10].

JACK [Win05] is a cross-platform environment for building and running MAS. It is built based on the BDI architecture, which is sound logical foundation, intuitive, and powerful abstraction. JACK includes an agent-oriented programming language; a platform for executing agents with infrastructure such as message marshalling and a name server; and development tools including a design tool, a graphical plan editor and a number of debugging views. JACK is entirely written in Java, which makes it portable and capable of running on any device. According with [KB15], JACK was the first platform with support for capabilities and hierarchical team structures. Although some elements from the organisation dimension are presented in JACK, not all of them are represented. Also, the dimensions of agents, organisations, and environments are not completely integrated.

JADE [BCG07], which stands for Java Agent Development Framework, is, as its name indicates, a framework for programming agents fully implemented in Java. It simplifies the implementation of MAS through a middleware that complies with the FIPA (Foundation for Intelligent Physical Agents) specifications. The JADE system can be described from two different points of view: (i) JADE is a runtime system for FIPA-compliant MAS, supporting application agents whenever they need to exploit some feature covered by the FIPA standard specification (message passing, agent life-cycle management, etc.); and (ii) JADE is a Java framework for developing FIPA-compliant agent applications, making FIPA standard assets available to programmers through object-oriented abstractions. In short, JADE is an open source project that does address abstractions of agents, but important concepts of MAS, such as from the dimensions of environments and organisations, are missing.

Surveys on agent-oriented development tools and agent platforms [PB09, KB15] point out that most of these frameworks suffer from the strong heterogeneity of the MAS field. This heterogeneity often leads to very specific approaches suitable only for one specific agent approach, e.g., a design tool for BDI agents only. From a Software Engineering (SE) perspective, a methodology (including concepts, notations, a process, and techniques) that guides practitioners in designing their systems is desired in agent-oriented approaches. Such methodology must comply with the concrete agent platform that is used to create the agent systems.

In practice, we observe that most AOSE approaches lack a standard and explicit methodology, as highlights Figure 2.1, which was obtained from one survey [PB09] on agent-oriented development tools. The authors Pokahr and Braubach [PB09] have defined that a *methodology* is what gives the conceptual foundations for the *modelling tools*, and, similarly, the *architecture* is what forms the conceptual foundation of programming *languages*. So far, the only AO methodology that this survey [PB09] have mentioned is Prometheus, which is tied with JAL only, the language of JACK [Win05]. We have updated this figure to include JaCaMo [BBH$^+$13] and the approach being proposed in this thesis on top of it. In order to situate the readers both on Figure 2.1 as well as on this thesis, we observe that: (i) the OntoMAS methodology is fully detailed in Section 4; and (ii) the OntoJaCaMo, a development environment integrated with Eclipse, is addressed in Section 6. We have indicated Protégé [Mus15] as the modelling tool for our methodology, however any other ontology editor could be an alternative to it for interacting with the OntoMAS ontology. The term MAOP (Multi-Agent Oriented Programming) was obtained in the paper from the authors of JaCaMo [BBH$^+$13], so we used it as its architecture, and, based on that, we have coined the term MAO (Multi-Agent Oriented) as the new paradigm of SE approach for covering all these points.

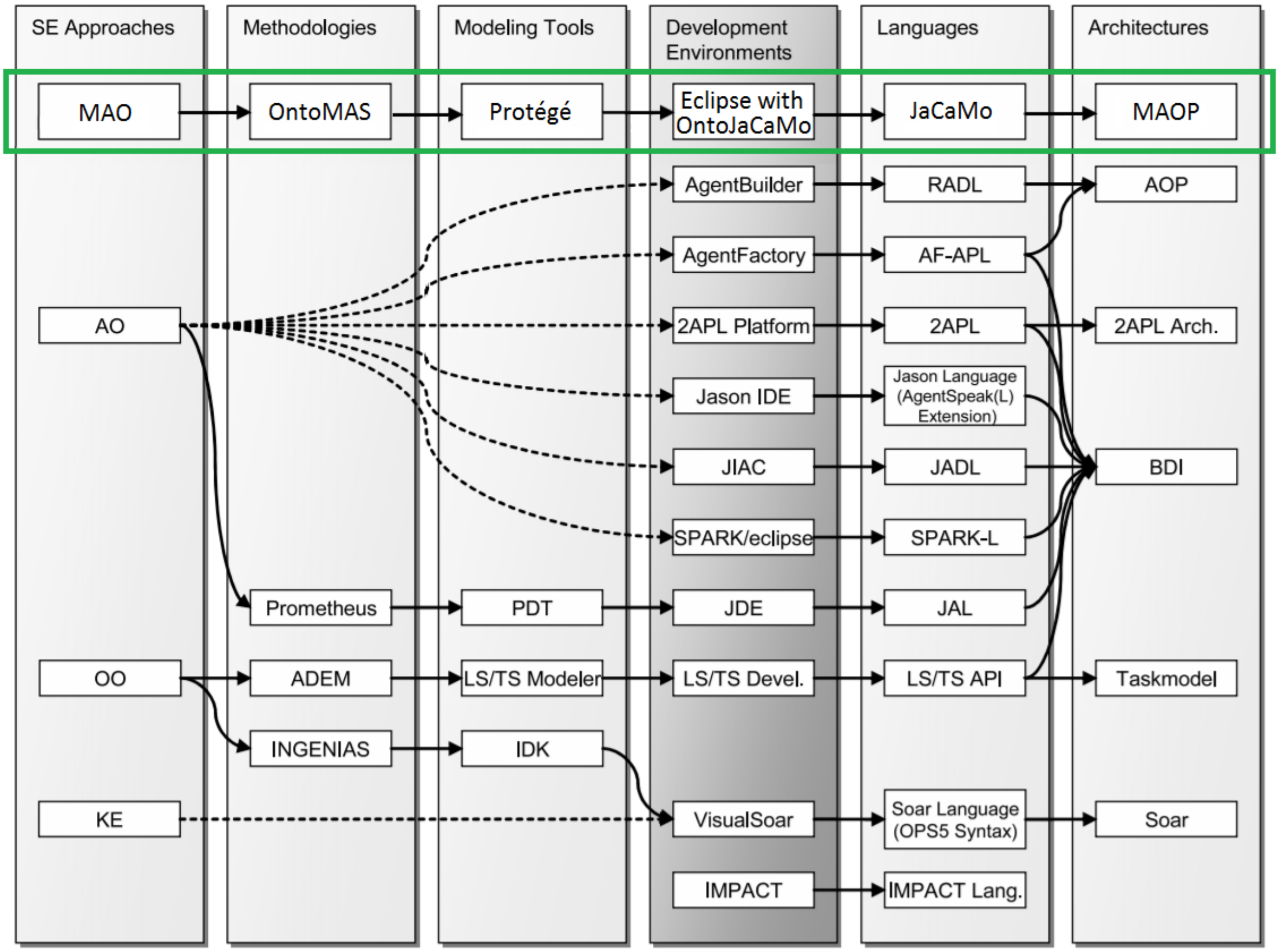


Figure 2.1 – Agent-oriented IDEs (obtained and adapted from [PB09]).

In this Section we explained the background knowledge on the topics of MDE, ontology, and MAS. Also, as we observed, other platforms for MAS programming do not cover the important features that JaCaMo provides. We already started to link one research area with each other, and the interconnections among them are explored with more detail in next Section.

3. RELATED WORK

> *"The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them."*
>
> *Sir William Bragg* — (1862 - 1942)

This Section shows the connections among MDE, MAS, and ontologies from three viewpoints. First, we show the importance and advantages of models for MAS (subsection 3.1). Second, relations of MDE with ontologies are given (subsection 3.2). Third, uses of ontologies in MAS are shown (subsection 3.3). This leads to the integration of all these three topics, in which we analyse the use of ontology for modelling agent systems (subsection 3.4). Figure 3.1 illustrates how such combinations of topics are addressed in each subsection in this chapter. Moreover, each of these topics in isolation was previously addressed in the background (Section 2) of this thesis: MDE in subsection 2.1, ontologies in subsection 2.2, and MAS in subsection 2.3. For a summarised comparison of related work, we refer to Table 3.1 at the end of the current Section, in which we also situate the research in this thesis.

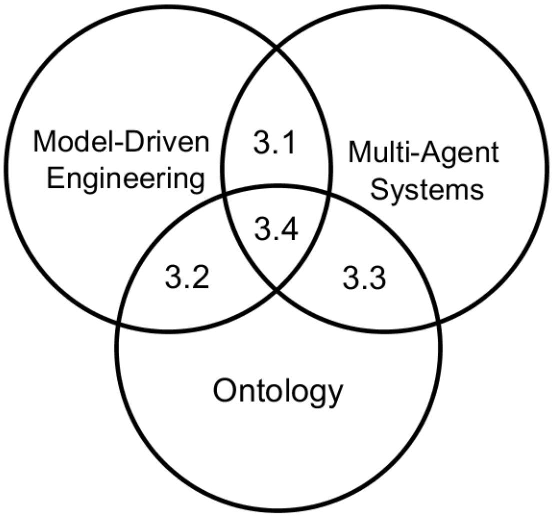


Figure 3.1 – Structure of subtopics in this Section of related work.

3.1 Model-Driven Engineering and Multi-Agent Systems

Several models and methodologies can be found in literature to formalise and define the processes of MAS design and implementation. For example, Prometheus [PW03] is one of the most well-known MAS modelling methodology for developing intelligent agent systems. It defines a development process with associated deliverables proven to be effective in assisting developers to design, document, and build agent systems based on concepts such as goals, beliefs, plans, and events. Prometheus [PW03] contains three phases: *system specification*, *architectural design*, and *detailed design*. It starts with the *system specification* phase that focuses on identifying the basic system functionalities, along with inputs (percepts), outputs (actions), and any important shared data sources. Then, the *architectural design* phase uses the outputs from the previous step to determine which agents the system will contain and how they will interact. Lastly, the *detailed design* phase looks at the internals of each agent and how it will accomplish its tasks within the overall system. Among future work for Prometheus [PW03] there is the introduction of social concepts to improve its current models, however one of our recent papers [FCVB16] has indicated that these improvements are not available yet in the latest official version of the Prometheus Design Tool (PDT). Therefore, some aspects of MAS are not covered by the models of Prometheus, which also does not explore the use of formal or explicit ontologies as part of its approach.

Prometheus is usually used as reference when MAS are combined with MDE in literature. For example, the Prometheus AEOlus [UH14] allows the integrated development of the three MAS dimensions (agent, environment, and organisation). It contributes with: *(i)* a new meta-model that combines the meta-models of Prometheus and JaCaMo; *(ii)* a new interactive incremental process based on the Prometheus process; and *(iii)* a code generation approach for JaCaMo based on this new meta-model. Prometheus AEOlus [UH14] improves modelling, code generation, and reduces the conceptual gap between the analysis and implementation phases. On one side, it extends Prometheus meta-models by including concepts to consider the environment and organisation dimension of JaCaMo. On the other side, it applies JaCaMo concepts to improve Prometheus development process to ensure that concepts used during the design and analysis stages will be used in the implementation stage. However, the proposed meta-models are not integrated with semantic technologies, reasoners, ontologies, and neither used during MAS programming. We observe that the code generation in Prometheus AEOlus [UH14] requires the refinement of entities in the model to generate code (for JaCaMo components, i.e., Jason, Moise, and CArtAgO). Thus, models designed in this approach must be refined to include platform-specific information, and once the first version of the designed MAS code is generated, the models are no longer used during the programming step to complete the MAS development.

Research in the direction of building tools for developing MAS through exploiting MDE techniques have led to a new proposal [GNFC12] of using Ecore with Prometheus. Ecore is used by the Eclipse Meta-modelling Framework to define meta-models, and that work have applied it to develop the meta-model concepts specific to Prometheus. More specifically, it have addressed the generation of MAS graphical editors based on the models and how agent code generators can be developed from such visual models. In the end, MAS programming code can be automatically generated from the models, ranging from code skeletons to completely deployable products [GNFC12]. To demonstrate this claim, templates have been created to automatically generate code in JACK language [GNFC12], which uses the BDI model to represent the internal structure of its agents. Once the model is converted to code, the developer must continue the MAS programming phase without using the model. Similarly as we see in other related work, this approach does not explore ontologies as part of such models and the MDE proposal does not present techniques for being used during MAS coding.

New aspects of MAS for programming platforms are also created and proposed as models. For example, a model specifying the *interaction* as a first-class abstraction to define MAS with respect to agents, environments, interactions, and organisations [ZH14]. The interaction allows the definition of the desired sequence of steps to achieve the organisational goals (while the organisational goals provide information about what the agents need to do, the interaction protocols provide a more detailed description about how to behave to achieve them). More specifically, that work presents a conceptual model for the interaction component, a programming language to specify the interaction, and how the proposed approach may be integrated in the JaCaMo platform. Such contributions allow developers to model the interaction in a separate component. Thus, the interaction does not need to be hard-coded inside the program instructions of agents or other components [ZH14].

MAS-ML is a multi-agent system modelling language [dSdL03] that extends the UML based on TAO (Taming Agents and Objects). The TAO meta-model defines the static and dynamic aspects of MAS. New diagrams - Organisation and Role diagrams - have been created due to the set of different elements and relationships defined in the TAO meta-model that have been incorporated in the UML meta-model [dSdL03]. Also, UML diagrams that already exist - Class and Sequence diagrams - have been adapted. Explanations about these diagrams can be found as follows [dSdL03]:

- The **Sequence diagram** represents the dynamic interaction between the elements that compose a MAS - i.e., between objects, agents, organisations, and environments.
- The extended **Class diagram** also represents agents and organisations together with the relationships between them and classes as defined in TAO.
- The **Organisation diagram** models the system organisations identifying their habitats, roles, and other elements - objects, agents, and sub-organisations.
- The **Role diagram** is responsible for clarifying the relationships between the agent roles and object roles.

MAS-ML has as similarity with this thesis the fact of defining agent, organisation, and environments as first order abstractions. However, our approach proposes to include the use of an ontology as meta-model for the MAS. Also, MAS-ML claims that it would be easier to use a programming language that considers these elements as first order abstractions to implement MAS-ML models [dSdL03]. While MAS-ML states as contribution “the mapping of the design elements in the agent level of abstraction to a programming language” but without determining a specific programming language, the research in this thesis emphasises the use of JaCaMo for MAS development.

MDE with agent-based models can facilitate the implementation of methods and tools for the development of MAS [PGSF06]. This subsection establishes links among MDE with MAS and it shows that models are commonly used to generate code automatically, but without making use of ontologies, or offering any type of model-based support during the programming and verification steps. We believe that such models can play a role not only in model transformation approaches that generate a first or skeleton version of MAS code, but also being employed until the end of MAS development. However, moving from agent models to implementation is, currently, not fully addressed by most agent-oriented methodologies in a systematic way [PGSF06], which leaves a gap between design and implementation. Also, we argue that while many modelling methodologies were proposed for agent development platforms in the past, they are not sufficient for the new and emerging techniques in agent programming, such as dealing with the multiple abstraction levels and not focusing only on the agents as individuals [FCVB16]. The approaches presented in this Section so far do not relate the models with ontologies in the areas of AOSE. In further subsections, we present the current research in this direction, but first the next subsection discusses relations of MDE with ontologies from a general viewpoint (i.e., without focusing in the specific context of MAS).

3.2 Model-Driven Engineering and Ontologies

None of the approaches in MDE we see in previous subsection explores ontologies for improving MAS modelling. Now we present an analysis of the relations of MDE with ontologies. Often, models are specified by instantiating meta-models. The definitions we found [AGK06] claim that “all ontologies are models, but not all models are ontologies”, however “there is no widely accepted definition of what distinguishes models from ontologies”. These two areas are, superficially, very similar, and in fact are sometimes visualised using the same language (e.g., UML). To better characterise their differences, we observe that models tend to use the close world assumption and focus on realisation issues, while ontologies usually rely on the open world assumption and focus on capturing abstract domain concepts and their relationships [AGK06].

Also, ontologies and meta-models are often designed with different goals in mind [KKK$^+$06]. For example, meta-models prove to be more implementation-oriented as they often bear design decisions that allow producing sound, object-oriented implementations.

Due to this, language concepts can be hidden in a meta-model, but they have to be made explicit in an ontology [KKK$^+$06]. In fact, there are proposals to create ontologies from meta-models, such as the lifting procedure [KKK$^+$06], which was designed to achieve semantic integration in modelling languages.

Other research directions rely on exploring these areas in interconnected ways, which is, for example, the application of MDE with ontology technologies [SWGP10]. Since OWL 2 has not been designed to act as a meta-model for defining modelling languages, Staab et al. [SWGP10] show how to build such languages in an integrated manner by bridging pure language meta-models and OWL in order to benefit from both approaches.

However, MDE and ontologies differ in some other points [AGK06] such as: *(i)* ontologies are generally used for run-time knowledge exploitation while models are not intended to contain instance data or be accessible at run-time; *(ii)* ontologies usually support “reasoning” while models cannot (or do not); and *(iii)* ontologies are expected to be represented with well-defined semantics in a language like OWL while models in a less precise language like UML. Atkinson et al. [AGK06] do not claim that models do not contain instance data, but that models are not intended to contain it. Our goal in mentioning this is to expose several different authors’ viewpoints in these areas since it is acknowledged that some definitions remain ambiguous and confusing.

Although the research areas of MDE and ontologies have been developed by two different communities, important synergies can be achieved by combining them [FBV17]. However, there are open research challenges for ontological approaches to model engineering, e.g., in which tasks ontologies and software models can be optimally used together and how ontologies should be integrated into MDE. Such investigations will lead to ways in which they can be made compatible and linked so as to benefit both communities.

This subsection discussed relations of MDE with ontologies from a general viewpoint, i.e., without focusing in the specific context of MAS. Next, in subsection 3.3, we show approaches where ontologies are used in and for MAS, but without addressing modelling issues. We highlight these two different roles played by ontologies in MAS: situations where ontologies are used without addressing modelling issues, and situations where ontologies are considered for AOSE. This topic, which has most relation with our thesis, namely the uses of ontologies for MAS modelling, is addressed in subsection 3.4.

3.3 Ontologies and Multi-Agent Systems

One of the first approaches in literature to consider ontologies to enhance an agent-oriented programming language was AgentSpeak-DL [MVBH05]. However, AgentSpeak-DL focuses on using ontologies during agent reasoning, instead of modelling aspects of MAS in ontologies. AgentSpeak-DL extends agents’ belief base with Description Logic (DL), in which the belief base includes: (*i*) one immutable TBox (terminological box, or conceptualisation) that characterises the domain concepts and properties; and (*ii*) one ABox (assertion box, or instantiation) with dynamic factual knowledge that changes according to the results of environment perception, plan execution, and agent communication. AgentSpeak-DL approach enriches the agent belief base with the definition of complex concepts that can go beyond factual knowledge [MVBH05]. The advantages pointed out of integrating agents and ontologies are [MVBH05]: (*i*) more expressive queries in the belief base, since results can be inferred from the ontology and thus are not limited to explicit knowledge; (*ii*) refined belief update given that ontological consistency of a belief addition can be checked; (*iii*) the search for a plan to deal with an event is more flexible because it is not limited to unification[1], i.e., it is possible to consider subsumption relationships between concepts; and (*iv*) agents can share knowledge using ontological languages such as OWL. Although such advantages enable new reasoning mechanisms for MAS by means of ontologies, our work investigates a different research direction in which ontologies are used as part of AOSE methodologies to aid modelling and implementation of agent systems.

JASDL [KB08] also merges agent belief base and ontological reasoning since it implements AgentSpeak-DL to provide Jason agents with ontology manipulation capabilities using the OWL API. Agent programmers benefit from features such as plan trigger generalisation based on ontological knowledge and the use of such knowledge in belief base querying [KB08]. Some Jason modules were altered to implement JASDL such as: the belief base was extended to partly resides within an ontology ABox, which, combined with a DL reasoner, facilitates the reuse of available knowledge in ontologies (to increase the inferences that an agent can make based on its beliefs and assure knowledge consistency); the plan library to enable enhanced plan searching; and the agent architecture to augment it with message processing to obtain semantically-enriched inter-agent communication.

CooL-AgentSpeak [MAB$^+$14] is an extension of AgentSpeak-DL with plan exchange and ontology services. It implements a CArtAgO artifact functioning as an ontology repository tool which stores a possibly dynamic set of ontologies and offers related ontology matching/alignment features. It searches for ontological relevant plans not only in the agent's local plan library, but in the other agents' libraries too, according to a cooperation strategy not based solely on unification and subsumption relations between concepts, but also on ontology matching. In short, CooL-AgentSpeak [MAB$^+$14] performs cross ontological unification for agents that do not disclose their ontologies to each other (that cooperate while preserving their privacy).

One of our own work is related with this thesis. We have implemented a mechanism for agents to interact with ontologies [FPH$^+$17][2] that is coded in a CArtAgO artifact, so any

---

[1] The traditional plans' unification relies on pattern matching mechanisms based only on syntax and lexical approaches for comparing plans. Thus, semantics is not considered to infer that a plan could be attempted in a given situation.

[2] [FPH$^+$17] is a journal extended version of the paper [FPH$^+$15].

agent platform that supports CArtAgO is able to make use of it. Such artifact allows agents the ability of interacting with OWL ontologies, so that ontologies may be used in the development and execution of MAS. The knowledge of agents is then increased from its standard knowledge representation mechanism (Jason, for example, uses a belief base) to any accessible OWL ontologies. Besides the gains on expressiveness, we have demonstrated experimental results in which the new approach for agents to represent their knowledge bases brings (given some scenarios and configurations) advantages in terms of better execution time and memory allocation. Others of our papers describe situations in which this artifact for ontologies was used, for example to interact with an ontology that represents tasks in the context of collaborative groups of agents [SPF$^+$16], and when information from this ontology was employed by agents to communicate and argue about task reallocations [PFS$^+$15].

This subsection presented approaches for incorporating ontological reasoning in agents. Although the advantages of using ontologies for agents are clear, few agent-oriented platforms are currently integrated with ontology techniques [FBV17]. Next subsection focuses on examples of ontologies proposed for modelling agent systems.

3.4 Ontologies for Modelling Multi-Agent Systems

As far as we know, the use of ontology to support modelling, development, and verification of MAS in the context of MDE is a new idea currently not fully explored in literature [FBV17]. We have found ontologies to represent only partial aspects of MAS, such as an environment ontology [OVBdRC06], and an ontology for organisations of agents [Zar12]. Such ontology models are desirable for all dimensions of MAS at the same time, but these levels have to be aligned so that they work as a common specification [FBV17]. This will make possible to model, to reuse, and to extend a MAS in one dimension while maintaining the others, which enables the designer to work without going into specifics of the programming languages that define each dimension. In this context, a MAS can be better designed, expressed, and communicated, and a specific modelled project can be more easily verified and converted to code or to a formal verification system.

Environments play an essential role in MAS, and their semantic representation improves the way agents reason about the objects with which they interact and the overall environment where they are situated [OVBdRC06]. This is important because most agent-oriented programming languages are weak in allowing the developer to model the environment within which the agents will execute [BDW06]. The use of an environment ontology adds three important features to existing multi-agent approaches [OVBdRC06]: *(i)* ontologies provide a common vocabulary to enable environment specification by agent developers (since it explicitly represents the environment and agent essential properties, defining environments in ontologies facilitates and improves the development of multi-agent simulations);

(ii) an environment ontology is useful for agents acting in the environment because it provides a common vocabulary for communication within and about the environment (it allows interoperability of heterogeneous systems); and *(iii)* environment ontologies can be defined in ontology editors with graphical user interfaces, making easier for those unfamiliar with programming to understand and design such ontologies.

In [OVBdRC06] an environment ontology is proposed based on environment aspects of agent programming technologies that is integrated into a platform for developing cognitive multi-agent simulations. Thus, it can be used to specify environments and derive a project-level, complete, and executable definition of multi-agent environments. An environment description is a specification of its properties and behaviour, which includes concepts such as: *objects* (i.e., resources of the environment); *agents* (i.e., their "physical" representation in the environment that is visible to other agents); *actions* that each type of agent can perform in the environment; *reactions* of the environment and objects when an agent's actions affect them; *perception types* available to each type of agent; and *observable properties*, that is, the information about the simulation to which observers (e.g., the agents) have access.

In [OVBdRC06], the relationship between the environment and other MAS dimensions was already foreseen, since they mention the intention of looking at higher-level aspects of environments, i.e., social environment aspects of agents, such as the specification of social norms and organisations in agent societies. In fact, on the MAS organisation dimension, there is a semantic description of MAS organisations [Zar12] formalised in OWL to specify an ontology for organisational characteristics of the Moise meta-model (structural, functional, and normative levels). This approach helps agents in becoming aware, querying, and reasoning about their social and organisational context in a uniform way. Also, this work makes possible to convert between the ontology and the Moise specification, providing more flexibility for modelling and developing agent organisations. This semantic description of Moise [Zar12] provides agent-side reasoning, querying features, and benefits such as increased modularisation, knowledge enriching with meta-data, reuse of specifications, and easier integration. With the semantic web effort aiming to represent the information in semantic formats, the MAS community can take advantage of these new technologies in MAS development tasks such as to integrate organisational models, to monitor organisations, and to analyse agent societies [Zar12].

3.4.1 Summary

A comparison among such related work and the research in this thesis is depicted in Table 3.1. It shows that there are already models and MDE approaches for more than one MAS dimension [PW03, dSdL03, GNFC12, UH14], but without using ontologies, semantic reasoning, or employing the model during the programming step. We also compared ontologies with MDE [AGK06, KKK^{+}06, SWGP10], and we show that ontologies are applied to extend agents' capabilities, but with other goals than the modelling of MAS [MVBH05, KB08, MAB^{+}14, FPH^{+}17]. When an ontology is used for MAS modelling, only a part is modelled, such as the environment [OVBdRC06], or the organisation [Zar12].

Table 3.1 – Comparing related work in the areas of MDE, MAS, and ontologies (adaptation from Table published in [FBV17]).

Research	**Overview of the work**	**Ontologies included**	**MAS dimensions modelled**	**MAS platforms used**
Model-Driven Engineering and Multi-Agent Systems				
Prometheus [PW03]	MAS modelling and development methodology	No	Agent, environment, and organisation	JACK
Prometheus AEOlus [UH14]	Approach for MAS modelling and programming	No	Agent, environment, and organisation	JaCaMo
MDE for MAS development [GNFC12]	Ecore meta-model of Prometheus for MAS development in Eclipse	No	Agent, environment, and organisation	JACK
Interaction component [ZH14]	Conceptual model and programming language for interaction aspects	No	Interaction	JaCaMo
MAS-ML [dSdL03]	MAS modelling language that extends UML based on TAO	No	Agent, environment, and organisation	Unspecified
This thesis - OntoMAS and OntoJaCaMo	An ontology for MAS modelling and a tool to aid in MAS programming	An ontology plays a key role in the proposed approach for MDE of MAS	Agent, environment, and organisation	JaCaMo
Ontologies and Multi-Agent Systems				
AgentSpeak-DL [MVBH05]	An approach for using ontologies during agent reasoning	A way for agents to represent knowledge and interact with ontologies	Ontologies extend agents' belief base with DL	AgentSpeak
JASDL [KB08]	Jason implementation of AgentSpeak-DL	Jason agents can represent knowledge and interact with ontologies	Ontologies extend agents' belief base with DL	Jason
CooL-AgentSpeak [MAB^{+}14]	AgentSpeak-DL's extension with plan exchange and ontology services	Each agent has its private ontologies	Ontologies extend agents' belief base with DL	CArtAgO and Jason
Ontology CArtAgO artifact [FPH^{+}17]	An artifact for agents to interact directly with OWL ontologies	MAS platforms aligned with CArtAgO have access to any ontology	Agents may use ontologies as their knowledge sources	CArtAgO and Jason
This thesis - OntoMAS and OntoJaCaMo	Methodology in which MAS are first modelled in the given ontology	An ontology is the basis of techniques for MAS modelling and coding	Agent, environment, and organisation	JaCaMo
Ontologies for Modelling Multi-Agent Systems				
MAS env. ontology [OVBdRC06]	Ontology to specify and derive definitions of MAS environments	An ontology for modelling characteristics of MAS environments	Environment	Unspecified
MAS org. ontology [Zar12]	OWL description of Moise organisations	An ontology for modelling concepts of organisations based on Moise	Organisation	Moise
This thesis - OntoMAS and OntoJaCaMo	An ontology and a tool for model-based development of MAS	An ontology for modelling elements of the three MAS dimensions	Agent, environment, and organisation	JaCaMo

Next Section explains the main point of our thesis which consists of an ontology and a methodology for MAS modelling that we called OntoMAS. Later Sections present our techniques and tool for model-based MAS development according with the proposed modelling approach. Our research combines MDE with an ontology perspective for building MAS, and we integrate concepts from different agent dimensions in a single framework, so agent-oriented software engineers benefit from receiving a robust methodology and tool to support the development of their systems with a comprehensive approach. In this context, our research pioneers in covering all these issues of MAS modelling, programming, and verification at the same time, while providing interesting benefits to agent developers, notably the users of JaCaMo.

4. AN ONTOLOGY FOR MODELLING MAS: ONTOMAS

> *“We should be taught not to wait for inspiration to start a thing. Action always generates inspiration. Inspiration seldom generates action."*
>
> *Frank Tibolt* — (1897 - 1989)

An ontology for defining the main abstractions of MAS, namely the concepts from agents, environments, and organisations is proposed in this thesis. The underlying idea is that the conception of a MAS project should start by its modelling in such ontology, which we refer to as OntoMAS. This can be done by extending the ontology top-level concepts, and adding new classes, instances and relationships in order to specify the corresponding desired project to be implemented in terms of agent-oriented concepts. Our approach advocates that a MAS should be first modelled based on this upper ontology of agent systems, which uses a single formalism to encompass the global characteristics of AOSE platforms. In these terms, OntoMAS can be seen as a language, a meta-model, a high-level conceptualisation, or as a domain-independent model of varied agent systems in which agent developers would use it to model/extend/instantiate their specific agent project [FBV17]. As result from using the proposed OntoMAS methodology for defining a specific MAS project, designers obtain an extended and instantiated ontology that may correspond to a project in JaCaMo [BBH$^+$13].

OntoMAS represents different MAS concerns while allowing to relate them, therefore offering advantages such as increased maintainability, usability, and extensibility for MAS modellers and developers. The OntoMAS modelling methodology consists in creating subclasses, instances, and relationships based on the concepts and properties provided by the ontology, which can be done with any ontology editor. When using OntoMAS, a particular MAS begins to be modelled by *extending* the proposed ontology, which is done by creating new subclasses to its top-level concepts. Then, individuals are created in the process of *instantiating* the extended ontology. From an instantiated model, it is possible to perform *reasoning* and obtain an inferred specification. Then, a model specified using OntoMAS may be used in our techniques for supporting MAS *programming*, which are embodied in the OntoJaCaMo tool. Such approach also allows to gradually refine from high-level abstract views to elements directly available in concrete technical MAS programming platforms. Figure 4.1 illustrates how OntoMAS and OntoJaCaMo fit in the phases of AOSE. Currently, any ontology editor tool, such as Protégé [Mus15], can be used to interact with OntoMAS during the MAS modelling. For the MAS development, OntoJaCaMo was implemented as a plug-in for Eclipse [Bud04], the standard IDE for JaCaMo [BBH$^+$13].

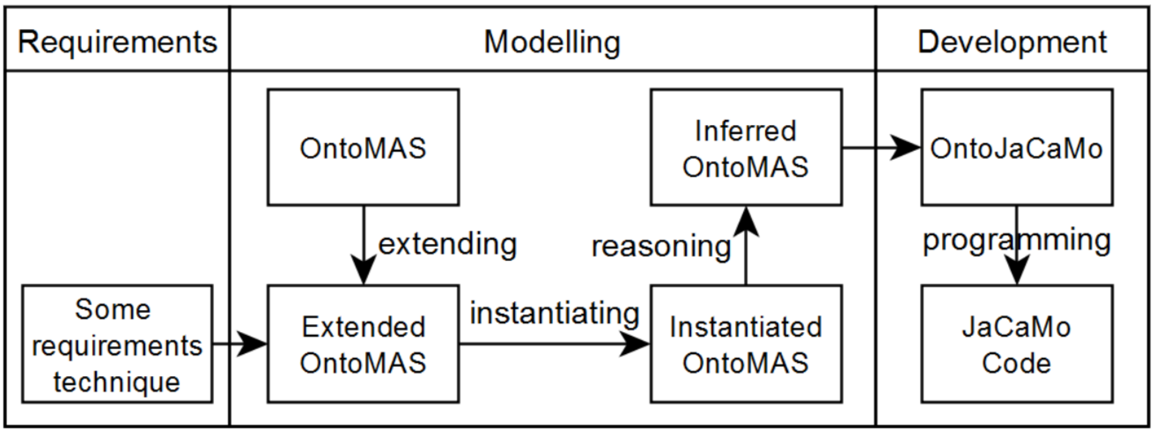


Figure 4.1 – AOSE methodology using OntoMAS and OntoJaCaMo.

Our investigation is also related with the exploration of approaches and tools that can apply models obtained from ontologies during the MAS coding step to offer support for programming and the use of ontology reasoning to perform inferences and verifications in a specified project of MAS. Section 6 of this thesis is specially dedicated to demonstrate the techniques implemented in OntoJaCaMo of how instantiated models of OntoMAS can be applied to support the generation and development of MAS code. Our techniques emphasises the transformation of elements from ontology models into code for JaCaMo through operations such as an initial conversion of ontology elements to code [FSP$^+$15], or enabling the drag-and-drop content from model to code [FBV17]. Advantages derived from such approach are techniques for: *(i)* integrating design and code; *(ii)* supporting MAS programming with automatic code generation through model-based development; and *(iii)* performing verification with focus on the use of semantic reasoning and model checking.

In each of the following subsections we describe in detail the meanings of concepts and properties represented in OntoMAS for each of its MAS dimensions. The conceptualisation formalised in OntoMAS was composed from the analysis and observation of several AOSE approaches and platforms, however it is highly oriented to and integrated with JaCaMo [BBH$^+$13]. The concepts and properties in OntoMAS are grouped according to the three MAS dimension previously discussed: agent, environment, and organisation. Through this Section, we present the general guidelines for project conception using our approach, and a comprehensive example specified in it that includes demonstrations of inferences obtained from semantic reasoning. We refer to Appendix A for an extended, complementary, and more detailed explanation on the guidelines for project conception using OntoMAS.

4.1 Agent Dimension of Multi-Agent Systems

From the agent dimension, we are not interested in defining any possible and generic characteristics of any kind of agent, such as physical agents. Instead, we are interested in specifying only the concepts of virtual agents that make sense in the context of programming for this dimension. Thus, the OntoMAS ontology contains the following 6 top-level concepts to represent the agent dimension: **Agent**, **Plan**, **Action**, **Goal**, **Belief**, and **Message**. Next, we detail the meaning of these concepts, the meaning of making subclasses to them, and the meaning of creating individuals for such concepts. The relation of these elements with the code in JaCaMo is also provided, as well as simple examples for contextualisation. Figure 4.2 summarises the main concepts and properties in the agent dimension of OntoMAS.

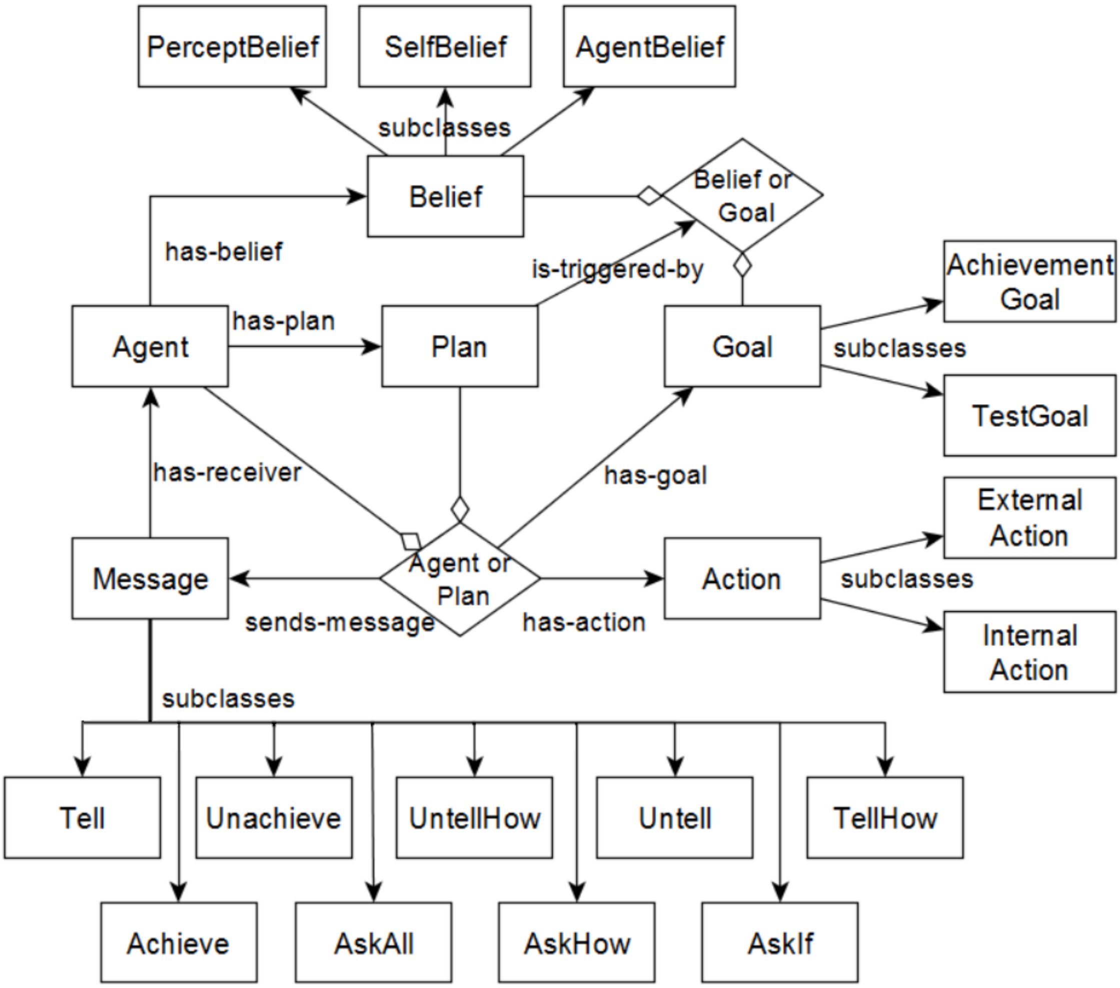


Figure 4.2 – Concepts and properties in the agent dimension of OntoMAS.

A subclass of **Agent** represents a type of agent, such as for example, $Player$. When defining a given concept as a subclass of **Agent**, this concept represents all individual agents of that kind. Subclasses of **Agent** are usually found in JaCaMo as the .asl files. An instance of a subclass of **Agent** represents an individual agent of that corresponding type, such as for example $playerJohn$. There are classes in this dimension that can be applied just by creating instances, which we argue that is the most simple way. However, the modeller is allowed to create subclasses to achieve an additional layer of expressiveness.

A **Plan** is a procedure composed of actions and triggered inside agents. The definition of each plan should be represented as an instance of the **Plan** concept. Thus, instances of plans represents the specification of a plan, such as for example $chooseMovement$. The specification of a plan is found in JaCaMo inside the .asl code of the type of agent that contains such plan. From this modelling perspective adopted in OntoMAS, the designer does not need to create subclasses of **Plan**, but this possibility is allowed.

There are two kinds of **Actions** represented in OntoMAS: **ExternalAction** and **InternalAction**. An **ExternalAction** is what the agent does that affects the environment, such as the act of opening a door. An **InternalAction** is how an agent act to manipulate its mental state, for example, forgetting some belief. While internal actions may be defined by local actions in the agent's state, external actions may refer to performing operations of artifacts that are situated in an environment. The definition of an action is represented by creating an instance of **Action**, such as for example $openDoor$. Actions are usually found in JaCaMo in the body of agents' plans. Similarly with plans, the designer does not need to create further subclasses of **Action**, but this possibility is allowed. For example, the subclass $openDoor$ could have two different instances according to different door handles, $openDoor - barhandle$ and $openDoor - knobshandle$.

A **AgentGoal** represents some agent individual desire to be achieved. Goals can be in one of the two following types. An **AchievementGoal** represents a state of the world (objective) that an agent can have intention to attain, such as having the door opened. A **TestGoal** is a check on the agent's beliefs in order to verify if a given belief holds, for example, querying the belief about the door being closed. Both achievement and test goals may fail, but for the plans which use them in order to continue their execution and finish with success, their goals must be completed. The definition of a type of goal that agents may pursue is represented by creating an instance of **AgentGoal**, such as for example to achieve $doorOpened$. Goals are usually found in JaCaMo inside code of agents (.asl files).

The **Belief** encodes the knowledge of agents, which can be one of the three types, as follows. **PerceptBeliefs** are obtained from environment perception, for example, the belief *stoveLit* to represent the state perceived from a device. **AgentBeliefs** are beliefs obtained from some other agent, for example, when an agent is told by other about something. **SelfBeliefs** are obtained by internal agent reasoning, for example, when an agent believes in something but not because it was perceived from the environment nor it was told by other agent. The definition of a type of belief is specified by creating an instance of **Belief**, such as for example $preferredMove$, which can be a **SelfBelief**. Beliefs are usually found in JaCaMo inside the code of agents (.asl files).

A **Message** is a communication that goes from one agent to another. The definition of a type of message is represented by creating an instance of **Message**, such as for example $informLocation$. Sending a **Message** may be a part of a plan in agents. The message types correspond to which performative is part of the $sender$ agent's intention, for example, if it is delegating a goal (**AchieveMessage**), informing a belief (**TellMessage**), requesting a plan (**AskHowMessage**), etc. There are 9 different types that a message can assume, each representing its illocutionary force. This is represented by the following concepts that are subclasses of **Message**:

- **AchieveMessage**: $sender$ intends $receiver$ to try and achieve $content$.
- **AskAllMessage**: $sender$ wants all of $receiver$'s answers to a question.
- **AskHowMessage**: $sender$ wants all of $receiver$'s plans that are relevant for the triggering event $content$.
- **AskIfMessage**: $sender$ wants to know if $content$ is true for $receiver$.
- **TellHowMessage**: $sender$ informs $receiver$ of a plan.
- **TellMessage**: $sender$ intends $receiver$ to believe $content$ to be true.
- **UnachieveMessage**: $sender$ intends $receiver$ to drop the goal $content$.
- **UntellHowMessage**: $sender$ requests that $receiver$ discard a certain plan.
- **UntellMessage**: $sender$ intends $receiver$ not to believe $content$ to be true.

Table 4.1 summarises the object properties related with the agent dimension in OntoMAS. We explain first the properties that involve the concept of **Plan**, which are simpler to explain since this concept does not require subclasses. After that, we explain properties involving subclasses and instances of the **Agent** concept.

Table 4.1 – Ontological object properties in the agent dimension.

Domain	**Property**	**Range**
Plan or Agent	has-action	Action
Plan or Agent	has-goal	AgentGoal
Plan	is-triggered-by	Belief or AgentGoal
Agent	has-plan	Plan
Agent	has-belief	Belief
Plan or Agent	sends-message	Message
Message	has-receiver	Agent

Plans may contain actions, which means that when a given plan is being executed, its corresponding actions may be performed. This is represented connecting instances of these concepts using the $has\text{-}action$ property, for example, $chooseMovement\ has\text{-}action\ openDoor$. The same is true for plans that may start the pursue of goals, defined through the property $has\text{-}goal$, as exemplified by $chooseMovement\ has\text{-}goal\ doorOpened$. Also about plans, they may be triggered by an event involving a belief or a goal, which is given by the property $is\text{-}triggered\text{-}by$ to connect instances of these concepts. To indicate that a given plan sends a specific message, the $sends$-$message$ property may be used. There is no need to specify for agents the has-$action$ and $sends$-$message$ properties if they were all specified for plans, a general rule can make inferences to check if an agent contain plans that have actions and send messages, the agent will also present these properties too. We refer to subsection 4.6 for more information about rules and reasoning over OntoMAS models.

Some properties work with the concept of Agent as its domain or range. We have explained that the Agent concept may have both subclasses (e.g., Player) and instances (e.g., $playerJohn$). When it is desired to use a property to connect between instances, the semantic is the same as explained in the previous paragraph. For example, agents may have beliefs, as expressed by the has-$belief$ property. If $playerJohn$ has some belief, lets call $preferredMove$, then these instances have to be connected using the mentioned has-$belief$ property. However, if all agents of that type (Player) have such belief, then a “subclass of” restriction should be used in that concept. This is represented as: Player is a subclass of has-$belief$ $value$ $preferredMove$. The same principles are applied to: the has-$goal$ property, which indicates the goals of agents; the has-$plan$ for indicating the plans of agents; and the $sends$-$message$ property, which indicates which messages the agent sends.

To connect an instance of message with an instance of agent that should receive it, the property has-$receiver$ can be applied (e.g., $informLocation$ has-$receiver$ $playerJohn$). To represent that all agents of a type (exemplified as Player) are receivers of a given message, a rule may be used as follows:

Message($informLocation$), Player($?p$) $->$ has-$receiver$($informLocation$, $?p$).

This rule can be read as: the instance of Message called $informLocation$ has all instances of Player (represented by $?p$) as receiver of such message. We refer to subsection 4.6 for more information about rules and reasoning over OntoMAS models. Next, we explain the formalisation of the environment dimension.

4.2 Environment Dimension of Multi-Agent Systems

From the environment dimension, OntoMAS is not interested in defining any possible and generic characteristics of any kind of environment, such as physical environments. Instead, it focuses on specifying only the concepts of virtual environments that make sense in the context of programming for this dimension using JaCaMo, in other words the characteristics of CArtAgO implemented environments. From the environment perspective, the main concepts are the *Spaces*, *Artifacts*, *Operations*, and *Percepts*. This dimension contains properties such as *has-artifact*, *has-operation*, and *has-observable-property*. Figure 4.3 summarises the main concepts and properties in the environment dimension of OntoMAS.

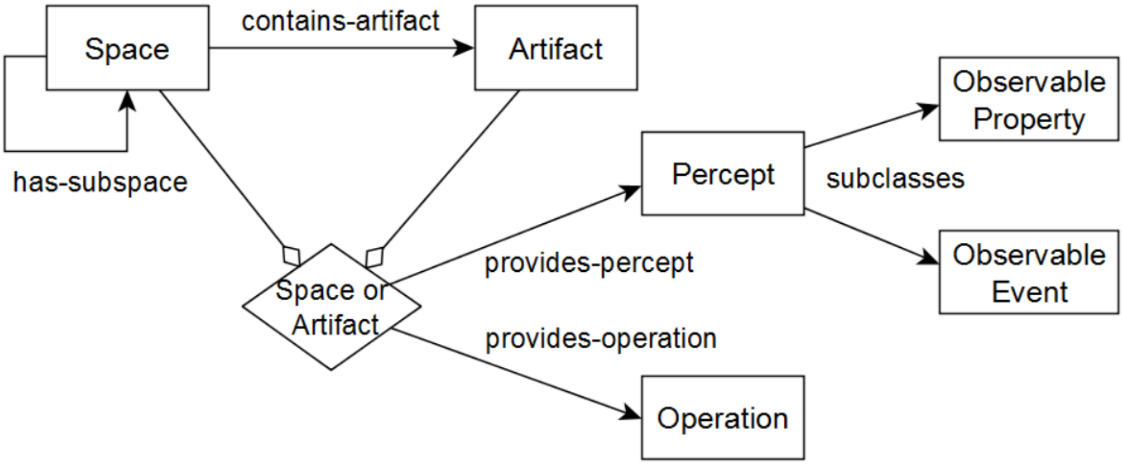


Figure 4.3 – Concepts and properties in the environment dimension of OntoMAS.

The OntoMAS ontology contains the following 4 top-level concepts to represent the environment dimension: **Artifact**, **Space**, **Operation**, and **Percept**. Next, we detail the meaning of these concepts, the meaning of making subclasses to them, and the meaning of creating individuals for such concepts. The relation of these elements with the code in JaCaMo is also provided, as well as simple examples for contextualisation.

Artifacts are resources that agents can interact with, thus, a subclass of **Artifact** represents a type of artifact. For example, $Computer$ may be a new concept that is an **Artifact** subclass in order to represent all artifacts of that kind. Each type of **Artifact** is found in JaCaMo as a Java class. Artifacts can be either the target (outcome) of agent activities, or the tools used by agents as means to support their activities (consequently, artifacts reduce the complexity of agents tasks' execution). This new concept of **Artifact** can be instantiated with individuals, such as $homeComputer$, so that it represents a concrete instance of that artifact's type. An instance of a subclass of **Artifact** represents a real object of that type, which may be found in JaCaMo in the .jcm file that describes the initial artifacts of a system, however other artifacts may be created after the initialisation of the MAS.

A **Space** represents the idea of workspaces where artifacts, agents, and sometimes even other spaces are situated. In simulations the environments are virtual, but they can be employed to make an abstraction of physical ones, where it is sometimes expected for the final deployment of agent systems. The concept of **Space** does not required to include subclasses on it, just instances, which represents a concrete space, such as for example $classRoom$. Spaces are initialised in JaCaMo at the .jcm file, but agents may make reference to spaces in their code (the .asl files).

An **Operation** represents a procedure that artifacts provide for agents to perform on them. For example, $turnOn$ may be an instance of **Operation**. Operations are found in JaCaMo as methods of the artifact that implements such procedures. Every instance of **ExternalAction** (from the agent dimension) that is invoked by an agent executes an operation of an artifact's instance, thus these concepts of **Operation** and **ExternalAction** are strongly related. There is no need to create subclasses for **Operation**.

A **Percept** is some information available to be observed from artifacts, which can be one of the two types: an **ObservableProperty**, which persists in the belief bases of agents; or an **ObservableEvent**, which does not persist in agents' belief bases. This differentiation exists because sometimes it is interesting for agents to store information obtained from environment perception, but in some cases this is not true. Thus, artifacts in CArtAgO have two methods to provide information for agents. This can be exemplified as: if the observation of temperature would be used only to trigger a plan, then $temperature$ may be an instance of **ObservableEvent**. However, if it is desired for agents that are observing the artifact to obtain and persist the value of the temperature, then it should be an instance of **ObservableProperty** instead. Instances of **ObservableProperty** are found in JaCaMo in the java code of artifacts through methods provided by the CArtAgO API (such as defineObsProperty, getObsProperty, and updateObsProperty) to manipulate them. An **ObservableEvent** is generated from another method provided by CArtAgO (called signal).

Table 4.2 summarises the object properties related with the environment dimension in OntoMAS. We explain first the properties that involve the subclasses and instances of the concept **Artifact**. After that, we explain properties involving instances of the **Space** concept.

Table 4.2 – Ontological object properties in the environment dimension.

Domain	**Property**	**Range**
Space	contains-artifact	Artifact
Artifact or Space	provides-operation	Operation
Artifact or Space	provides-percept	Percept
Space	has-subspace	Space

If a characteristic affects all artifacts of a type, then it should be defined as a class restriction, for example, to represent that all computers contain the operation to be turned on, this would be an ontology restriction on the $Computer$ concept that should be added using the $provides\text{-}operation\ value\ turnOn$, with $turnOn$ being an instance of **Operation** and $Computer$ a subclass of **Artifact**. The same principle is applied to the the $provides\text{-}percept$ property (e.g., $Computer$ is subclass of $provides\text{-}percept\ value\ temperature$, with $temperature$ an instance of **Percept**). Thus, a subclass of **Artifact** should define which operations the artifacts of this type provide, and which percepts can be obtained from them. A characteristic that affects the artifacts at the individual level is defined in the relations of individuals of **Space**, which we are shown next.

Examples of object properties that should be used to define characteristics of spaces are $contains\text{-}artifact$ and $has\text{-}subspace$. Spaces may contain artifacts, for example, suppose that there is a space called $home$, modelled as an instance of **Space**, and that there is an instance of the Artifact type Computer, lets call $homeComputer$, located in this space. Then, $home$ should have a relationship with $homeComputer$ using the property

$contains\text{-}artifact$. Also, an space may contain other, for facilitating the representation of the environment. This is represented by the $has\text{-}subspace$ property, such as for example $home$ $has\text{-}subspace$ $livingRoom$.

It can be inferred which operations and which percepts can be obtained from each space based on which artifacts are located in such spaces. A rule may be used as follows:

$$contains\text{-}artifact(?s, ?a), provides\text{-}percept(?a, ?p) -> provides\text{-}percept(?s, ?p).$$

This rule can be read as: if the space s contains an artifact a, and a provides a percept p, then s provides p. The same reasoning principle applies to operations of artifacts that are located in some space. Moreover, another general rule about environments is that the percepts and operations of subspaces are also provided by the spaces that contain them. Next, we explain the formalisation of the organisation dimension.

4.3 Organisation Dimension of Multi-Agent Systems

In literature it is possible to find considerable research on ontology for representing generic organisational characteristics [FBV17], however, it is inadequate to the needs and goals of OntoJaCaMo, which emphasises the concepts of organisations that make sense in the context of programming for this dimension using JaCaMo, in other words the characteristics of Moise implemented organisations. Thus, OntoMAS does not focus in defining any possible and generic characteristics of any kind of organisation. The definition of some concepts such as role, group, and so on, may differ from those work to ours, because of our implementation-oriented approach for this specific viewpoint in this dimension.

Thus, the OntoMAS ontology contains the following 5 top-level concepts to represent the organisation dimension: *Role*, *Group*, *OrganisationGoal*, *Mission*, and *Norm*. Exemplar properties in the organisation dimension are that *Group* $contains\text{-}role$ *Role*, *Mission* $has\text{-}goal$ *Goal*, and *Norm* $targets\text{-}role$ *Role*. Next, we detail the meaning of these concepts, the meaning of making subclasses to them, and the meaning of creating individuals for such concepts. The relation of these elements with the code in JaCaMo is also provided, as well as simple examples for contextualisation. Figure 4.4 summarises the main concepts and properties in the organisation dimension of OntoMAS.

An organisational **Role** is a function that agents can adopt. A *Role* definition states that agents playing that role are willing to accept the behavioural constraints related to it. Roles are used during the definition of groups, and agents that adopt roles should be compliant with some restrictions. Instances of **Role** should be create for each definition of a role in the organisation, such as for example, $buildingCompany$. The definitions of roles are found in JaCaMo in the XML file corresponding to Moise, and the code of agents in Jason can make reference to such roles. There is no need to create subclasses of **Role**.

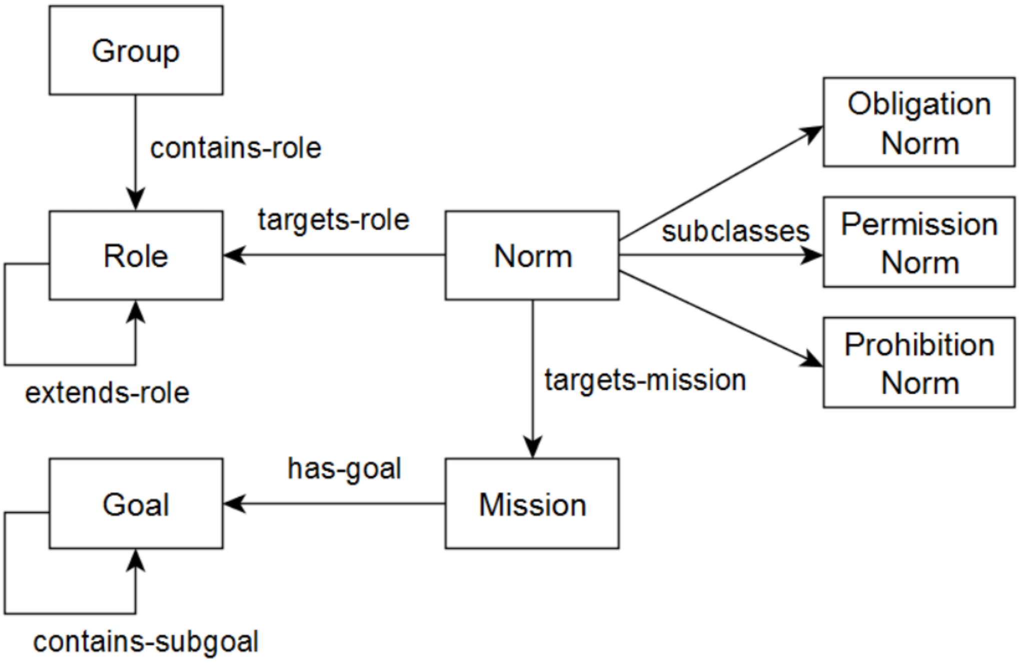


Figure 4.4 – Concepts and properties in the organisation dimension of OntoMAS.

A subclass of **Group** represents a type of group. For example, $HouseBuilderTeam$ may be a new concept that is a subclass of **Group**, which represents all groups of that kind. The definition of groups can be found in JaCaMo in the XML that specifies an organisation in Moise. This new concept can be instantiated, e.g. $johnsHouseBuilderTeam$, so that it represents instances of that groups's type, in other words, concrete groups. Instances of groups can be found in JaCaMo in the .jcm file, and the code of agents in Jason can make references to those groups.

An **OrganisationGoal** represents an organisational objective to be achieved. For each goal of this kind to be represented, an instance must be created for this class. Thus, there is no need to create subclasses in the **OrganisationGoal** concept. An instance of **OrganisationGoal** makes reference to a goal at the organisational level, such as for example $houseBuilt$. Instances of this type are found in JaCaMo in the file that specifies an organisation in Moise, and the code of agents in Jason can make references to organisation goals (for example, agents may have plans to act when a goal is assigned to them from the organisation).

A **Mission** is an structured mean composed of organisational goals. The organisation functional dimension specifies how global collective *Organisation Goals* should be achieved, i.e., how they are decomposed in global plans, grouped in coherent sets (missions) to be individually distributed to agents. An example of **Mission** can be the instance $managementHouseBuilding$, which is used for coordinating the execution of organisational goals. Missions are defined in the XML file of the Moise part of a JaCaMo project. There is no need to create subclasses in the **Mission** concept.

A **Norm** is a regulation over the organisational behaviour, which can be one of the three types: an **ObligationNorm** is a norm that must be done; a **PermissionNorm** is a norm that is allowed (not prohibited); or a **ProhibitionNorm** is a norm for something defined as forbidden. The normative dimension binds the structural level with the functional one to specify role's permissions, prohibitions, and obligations for missions. Each new norm should be an instance of one of these concepts. One example of norm may be the instance $buildingCompanyContractualObligation$, of the concept **ObligationNorm**, to define that some role in the system is obligated to fulfil a given mission. Instances of norm are defined in the Moise organisation files of a JaCaMo project, and there is no need to create subclasses in the **Norm** concept, or to any of its three already defined subclasses.

Table 4.3 summarises the object properties related with the organisation dimension in OntoMAS. If a characteristic affects all groups of a type, then it should be defined as a class restriction, thus representing that all groups of that kind have that specific characteristic. For example, all soccer teams have a minimum of 5 soccer players defined as an ontology restriction on the $SoccerTeam$ concept should be added as $contains\text{-}role\ min\ 5\ SoccerPlayer$, with $SoccerPlayer$ being an instance of **Role**, and $SoccerTeam$ a subclass of **Group**.

Table 4.3 – Ontological object properties in the organisation dimension.

Domain	**Property**	**Range**
Group	contains-role	Role
OrganisationGoal	contains-subgoal	OrganisationGoal
Role	extends-role	Role
Mission	has-goal	OrganisationGoal
Norm	targets-mission	Mission
Norm	targets-role	Role

A **Role** may be an specialisation of another one, thus, the property $extends\text{-}role$ may be used to define hierarchy among instances of Roles, where each instance defines a type of role. Following the example of an organisation to build a house, we can specify that $painterCompany\ extends\text{-}role\ buildingCompany$.

Instances of organisation goals can be related among themselves to define composition characteristics using the object property $contains\text{-}subgoal$, which specify that a goal is compose of some other (sub)goals. This is done to decomponse a bigger goal into other goals that are easier to coordinate or distribute in the MAS. For example, $houseBuilt\ contains\text{-}subgoal\ plumbingInstalled$, with both individuals as **OrganisationGoal** instances.

Several instances of goal may be part of a **Mission**, so, a Mission's instance that contain goals should use the $has\text{-}goal$ object property to specify its required goals. One example of this is that the mission $managementHouseBuilding\ has\text{-}goal\ houseBuilt$.

The **Role** that is being regulated by a **Norm** is specified through the $targets\text{-}role$ relationship, and its related **Mission** is defined with the property $targets\text{-}mission$. Following the example of the **ObligationNorm** instance called $buildingCompanyContractualObligation$, it may be defined with the following relationships: $targets$-$role$ $buildingCompany$, and $targets$-$mission$ $managementHouseBuilding$.

Next, we explain the integration of these three dimensions through the use of concepts, properties, and rules that connect elements from different dimensions.

4.4 Connecting Agents, Environments, and Organisations

The classes and properties in OntoMAS are modelled in three sub-ontologies, one for each dimension: agent, environment, and social organisation (subsections subsections 4.1, 4.2, and 4.3, respectively). The integration and connections among concepts in the dimensions of OntoMAS are encoded by means of concepts, object properties, and rules which determine how elements are allowed to relate among each other. Figure 4.5 summarises properties for connecting concepts from the three different dimensions of OntoMAS.

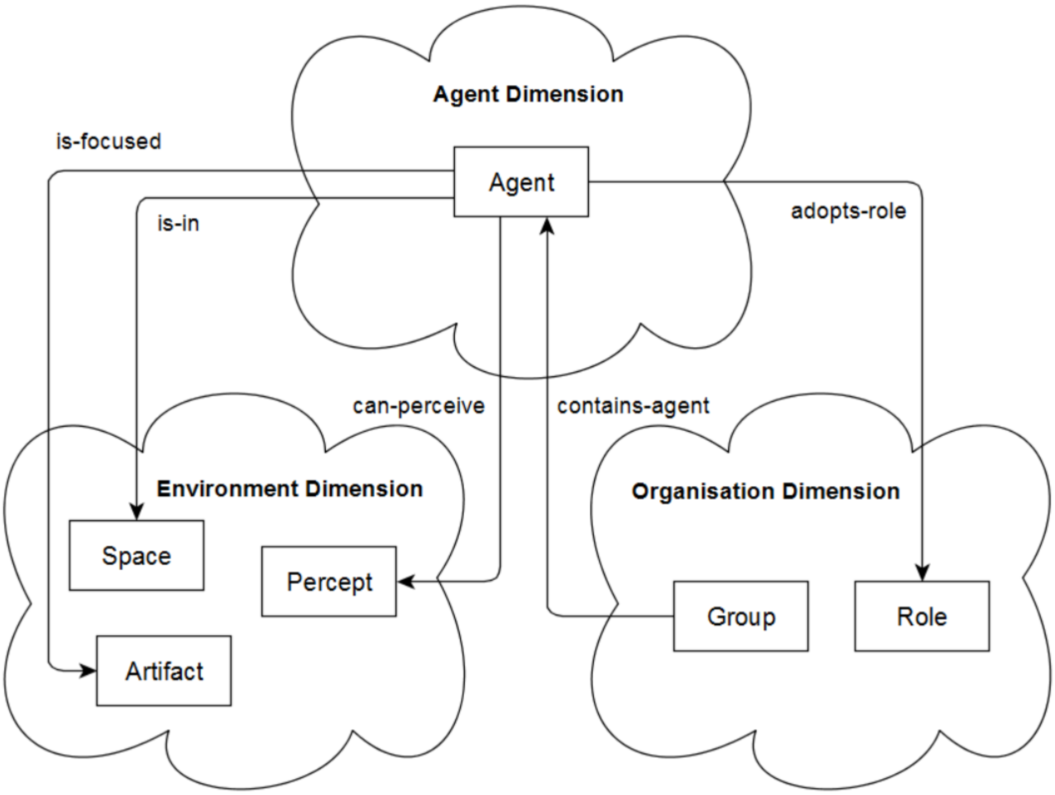


Figure 4.5 – Main properties for integrating concepts of OntoMAS dimensions.

Table 4.4 lists object properties for connecting concepts in from the different dimensions of OntoMAS. To specify the location of instances of agents in the spaces from the environment dimension, the property is-in may be used. For example $playerJohn$ is-in $classRoom$. The property is-$focused$ connects an agents with an instance of artifact in which that agent is focused, such as $playerJohn\ is\text{-}focused\ homeComputer$. Then, some properties may be obtained by inference over elements from different dimensions. If an agent ($?a$) is in a space ($?s$), and this space provides some percept ($?p$), then this agent can have such percept ($?a\ can\text{-}perceive\ ?p$). This is specified through the following rule:

$$is\text{-}in(?a, ?s), provides\text{-}percept(?s, ?p) \text{ −> } can\text{-}perceive(?a, ?p).$$

When relating concepts from the dimensions of agent and organisation, we may desire to specify that a given agent is adopting a role. This may be done with the property $adopts\text{-}role$. If a characteristic affects only some individuals of a group, then it should be defined as an object property in those affected instances. In this case, for example, if the $redSoccerTeam$ contains $playerJohn$ agent these instances should be related with the object property $contains\text{-}agent$.

Table 4.4 – Ontological object properties integrating concepts of different MAS dimensions.

Domain	Property	Range
Agent	is-in	Space
Agent	is-focused	Artifact
Agent	can-perceive	Percept
Agent	adopts-role	Role
Group	contains-agent	Agent

Next, we comment a different example of an OntoMAS modelling possibility for completeness.

4.5 An example of MAS modelled based on OntoMAS

This subsection provides a detailed example of how a given MAS can be modelled on the basis of the OntoMAS ontology. Although that subsections 4.1, 4.2, 4.3, and 4.4 already have introduced examples while explaining the concepts and properties in each dimension of the ontology, we found it interesting to mention this example that is more unified towards an unique scenario that was recently published in one of our papers [FBV17]. The scenario exemplified in this subsection, besides complementing our explanations, already introduces some examples on the topic of ontological reasoning over OntoMAS models, which is further addressed in subsection 4.6.

Consider a soccer game scenario to be designed and implemented from an agent-oriented point of view. In this scenario there are two different types of agents: players and coaches. Player agents can perform actions such as moving in the soccer field, or passing the ball. The coach can send messages to player agents in order to inform which roles they should adopt in each moment of the match. The environment is the soccer field, where all agents are situated. Agents in the soccer field environment can perceive things such as the ball position, and the match score. A team is an organisation, composed of a group of one coach and several players. The players can play different roles, such as defender, striker, and captain (or leader). For these roles, different missions may be assigned, such as defending or attacking. This brief specification addresses concepts of each one of the three MAS dimensions. Next, we illustrate how to formalise it on the basis of *extending* and *instantiating* OntoMAS.

Figures 4.6, 4.7, and 4.8 illustrate, respectively, examples from the agent, the environment, and the organisation dimensions. These examples contemplate how subclasses should be created and instances have to be added considering the soccer scenario specification. These are the steps have been previously referred as *extending* and *instantiating* the OntoMAS. The original elements from OntoMAS are represented by blue rectangles, while the subclass extensions are depicted by yellow rectangles, and the instances are illustrated by the purple diamonds. In Figure 4.6 we can see that agents have two types in this MAS specification, therefore two subclasses of *Agent* were created: *Player* and *Coach*. We included two instances of *Players* to exemplify how to make reference to individual agents (named *player1* and *player2*). Each instance of a subclass of the agent concept represents a concrete agent in the system, whereas its type is specified by its class. Moreover, two types of *Actions* are defined as subclasses of the *Action* concept: *Move* and *PassBall*.

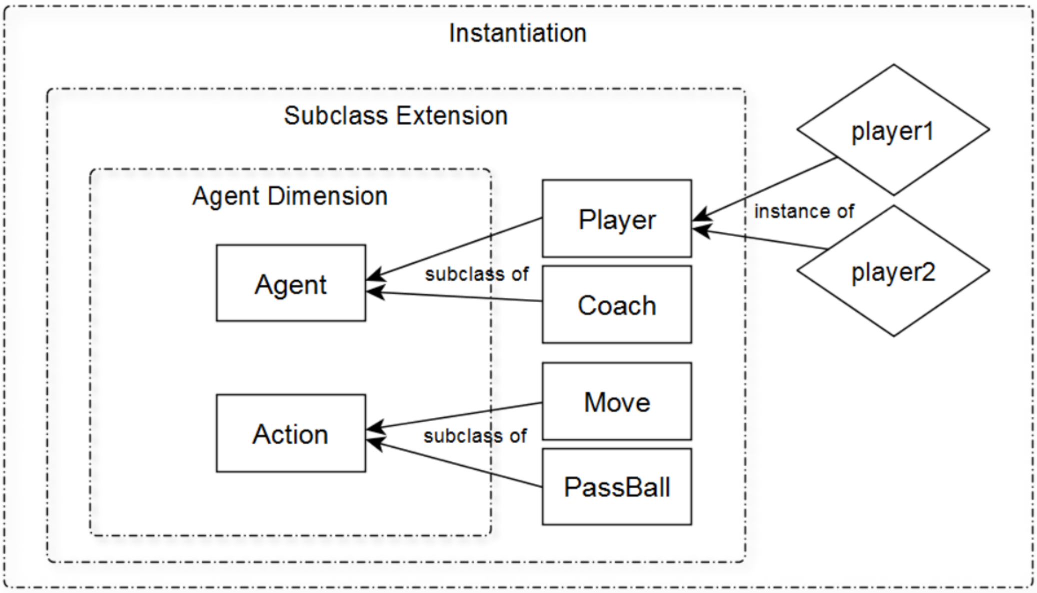


Figure 4.6 – Agent example (adaptation of image published in [FBV17]).

The example of environment characteristics in this MAS are depicted by the subclasses and instances illustrated in Figure 4.7. For example, the subclass *SoccerField* represents a type of *Space* in which our agents are situated. A concrete individual of this type is specified by the instance *soccerField1*. One type of *Artifact* (i.e., resource) that exists in the environment for agents to interact with is defined by the subclass *Ball*, whereas *ball1* is assigned as a valid instance of it. The subclasses *Score* and *BallPosition* illustrate types of

Observable Properties that resources may provide to agents, whereas an instantiation with values for these properties (for beginning the MAS simulation) is represented by *score1* and *ballPosition1*.

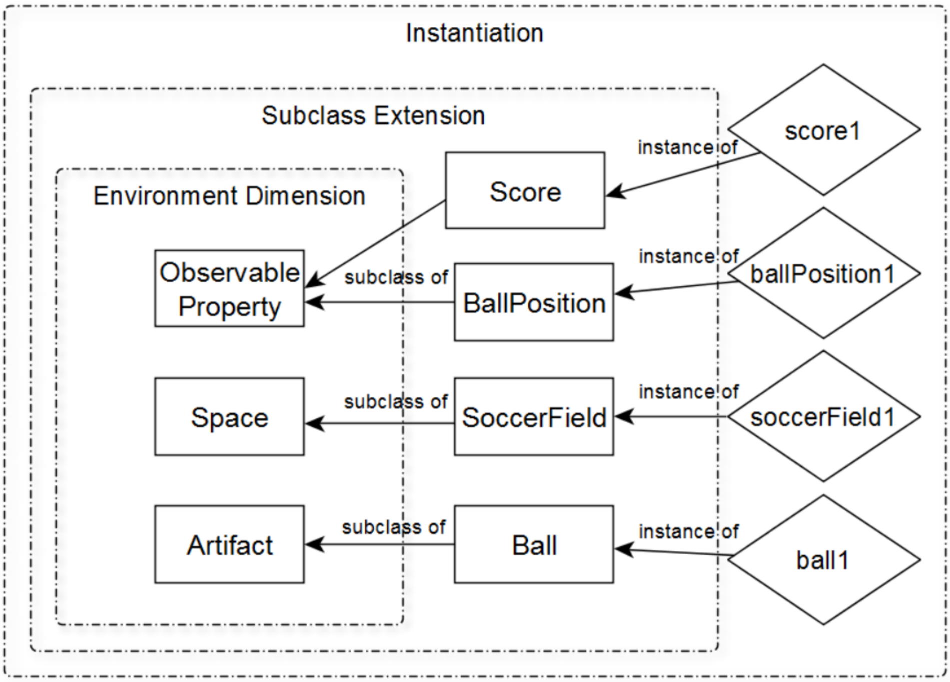


Figure 4.7 – Environment example (adaptation of image published in [FBV17]).

Figure 4.8 shows subclasses and instances to represent organisation characteristics of this MAS. Two subclasses specify types of *Missions*: *Defending* and *Attacking*. Instances of these subclasses, such as *attacking1* and *defending1*, define an assignment of that Mission type to an agent. The types of *Roles* are given by the following three subclasses: *Defender*, *Striker*, and *Captain*. Instances of these subclasses of *Role* (e.g., *defender1*, *defender2*, *striker1*, *captain1*) define which agents are playing the corresponding roles. This example shows how to encode part of one possible strategy for modelling organisational characteristics of agent systems. However, other strategies are possible and would result in different designs and implementations for this scenario. The modelling requires also the creation of relationships among the individuals, and the inclusion of some other axioms, which our example illustrates by using object properties and restrictions over the subclasses.

Figure 4.9 illustrates in more detail how such example can be specified using screenshots obtained directly with the Protégé [Mus15] ontology editor[1]. We have adopted Protégé since it has become the most widely used software for building and maintaining ontologies [Mus15]. We suggest the use of Protégé to interact with OntoMAS, however any other ontology editor could be an alternative to it. As already explained, the desired MAS is specified and modelled on top of the elements provided by the proposed OntoMAS ontology. This is done by, for example, adding new classes, refining concepts, creating instances,

---

[1] Available open source at http://protege.stanford.edu/

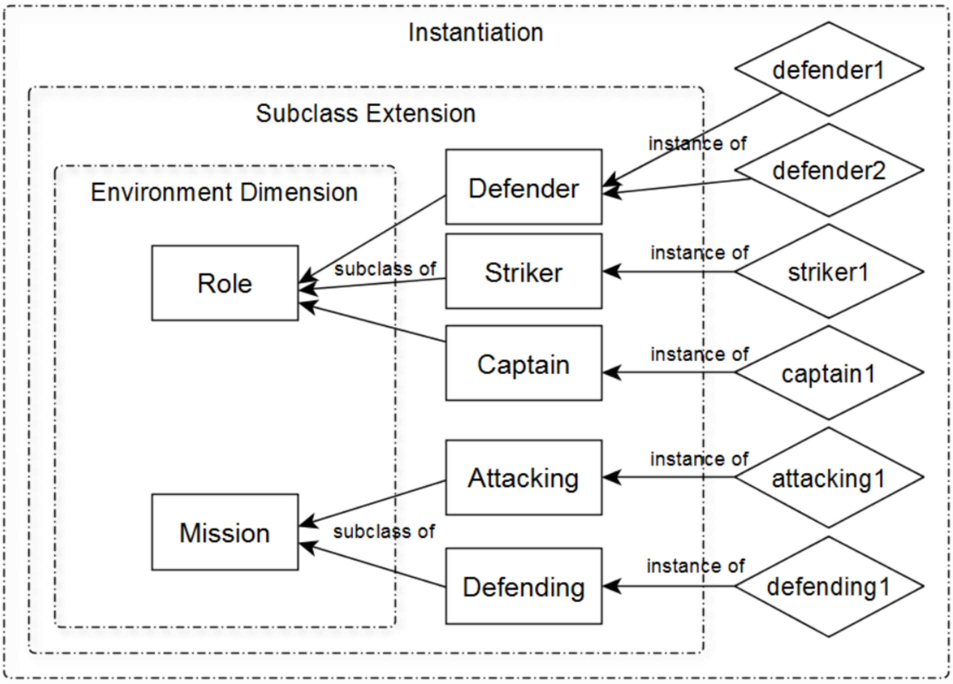
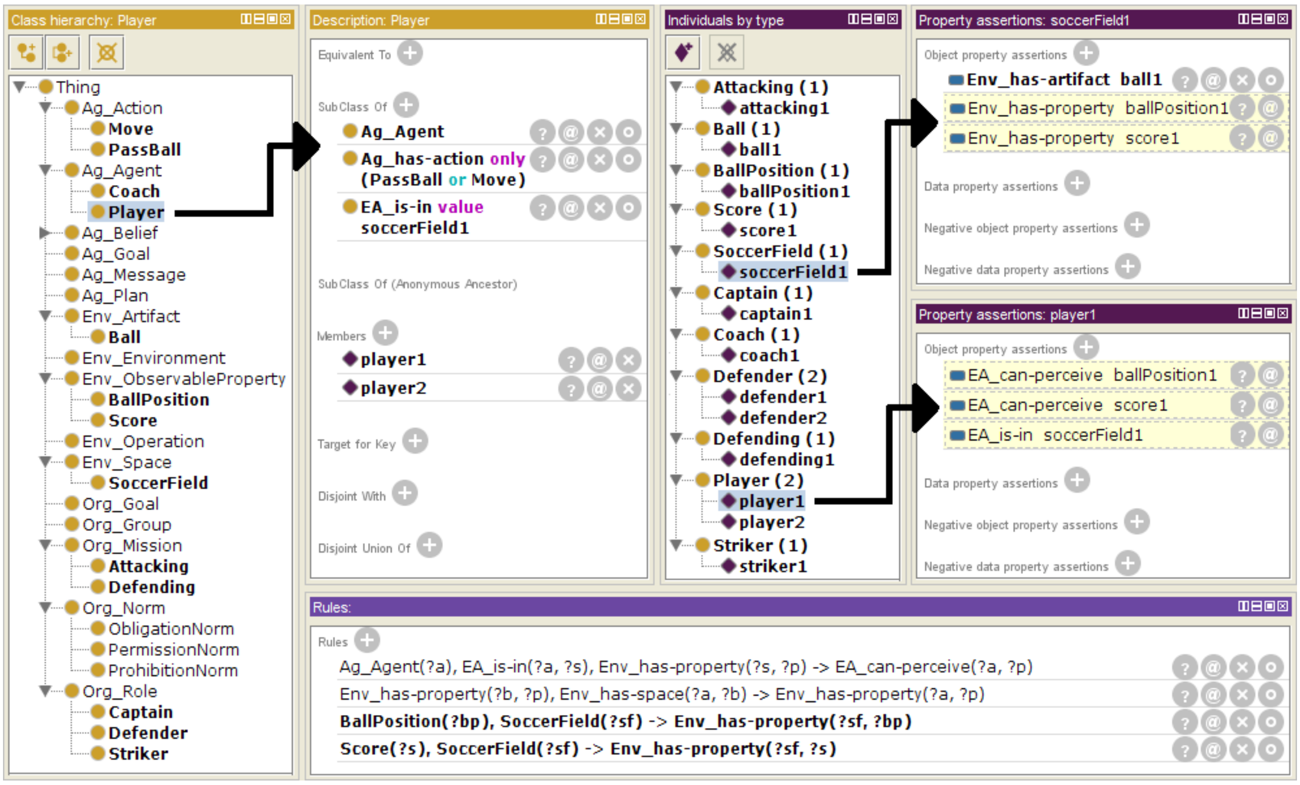


Figure 4.8 – Organisation example (adaptation of image published in [FBV17]).

asserting properties, and so on. After using any ontology editor for modelling the MAS, the resulting OWL file can be employed for several purposes, such as for example, in the OntoJaCaMo tool, detailed in Section 6, for supporting agent-oriented programming using a given ontology-based specification.

The left-hand side of Figure 4.9 illustrates the subclasses hierarchy, with emphasis on describing the *Player* concept. This part states the actions that players may execute, an environment where players are situated, and instances of this subclass. The classes *Coach* and *Player* represent types that individual agents can assume. Thus, as shown in Figure 4.9, they were created as subclasses of *Ag_Agent*. This figure also illustrates some characteristics for the *Player* concept such as players being able to execute the actions of passing the ball and moving (this part is defined using the “SubClass of” restrictions on the concept *Player*), and which individuals are of the type *Player* (e.g., the members *player1* and *player2*). The environment where the agents are situated is specified using a relation that connects the *Environment* and *Agent* concepts: the property *EA_is-in* from *Ag_Agent* to *Env_Space*. In this case, there is a restrictions for specifying that all players are situated inside the space defined as *soccerField1*.

The right-hand side of Figure 4.9 shows details about individuals and relationships that are asserted or inferred to some of these individuals. The inferences obtained by the execution of semantic reasoning over this example are shown in this figure inside dashed rectangles (while the asserted information is in bold). It is asserted that the *soccerField1* instance has an artifact called *ball1*, and the defined rules allow the inference that this space contains the observable properties of *ball position* and *score*. Since *player1* is explicitly defined as an individual of *Player*, reasoners can use class restrictions of Players to imply



```
Ag_Agent(?a), EA_is-in(?a, ?s), Env_has-property(?s, ?p) -> EA_can-perceive(?a, ?p)
Env_has-property(?b, ?p), Env_has-space(?a, ?b) -> Env_has-property(?a, ?p)
BallPosition(?bp), SoccerField(?sf) -> Env_has-property(?sf, ?bp)
Score(?s), SoccerField(?sf) -> Env_has-property(?sf, ?s)
```

Figure 4.9 – Subclasses of agent with some conditions, instances, and rules in the ontology with asserted and inferred properties (first published in [FBV17]).

its location (*soccerField1*). Also, rules support the inference of which observable properties this individual agent can perceive because of its location. These rules, which are shown at the bottom of Figure 4.9, use an extension for OWL called SWRL (Semantic Web Rule Language). Rules can be inherited from the base ontology, and new ones can be added particularly to a specific extension and instantiation when defining a desired MAS scenario (these are in bold). Next, in Subsection 4.6, we explain in more detail this part of using semantic reasoning and rules in SWRL to obtain inferences over OntoMAS models.

4.6 Ontological Reasoning over OntoMAS Models

Model verification refers to the processes and techniques that the model developer uses to assure that his or her model is correct and matches any agreed-upon specifications and assumptions [Car02]. OntoMAS can be explored with its available reasoning mechanisms to implement model verification in the context of MAS [FBV17]. The literature reports that most practical approaches for verification of MAS are done on code, and most of the work done on model checking within the MAS research area is quite theoretical [BDW06]. However, there are approaches that use existing model checkers, typically to check properties of particular aspects of a given MAS. While this has the advantage of proving properties of the system that will be actually deployed, it is also often useful to check properties during the system design [FBV17]. In fact, all the work on model checking for MAS is claimed to be still in early stages so not really suitable for use on large and realistic systems [BDW06].

Considering this context, semantic reasoners may provide, for example, consistency checking and inferences about the MAS specified in an ontology [FBV17]. The possibility to reason about the model fosters the implementation of model checking features in the context of MAS. For example, when considering only MAS organisations, it is possible to verify conflicts considering the existing norms, roles, and missions. When an instantiation of MAS organisation is combined with instantiated agents, it is possible to verify other kind of inconsistencies integrating information from more than one dimension, such as if the agents contain the required capabilities to achieve the existing organisation goals. Similarly, other kinds of verification are possible when focusing on the interactions of instantiated agents and environments, i.e., verifying if agents' actions are valid in a given environment configuration [FBV17].

As previously explained, when integrating and matching information from more than one dimension, it is possible to perform consistency checking through inferences obtained by reasoning with our ontology [FBV17]. One example is to identify if the actions from agent's dimension are available as operations provided by the environment dimension. If there is an action from an agent that does not exist in the environment, the invocation of such action in run-time will result in an action failure. Thus, the verification of characteristics in instantiated model at design time may prevent future errors to happen during the execution time of the corresponding JaCaMo specified project. Similarly, it is possible to verify if the given agents have the capability to achieve the goals specified in any specific organisation. Organisation goals are assigned to agents playing the organisation roles, and an agent playing a specific role may not have the required plans to achieve the goals that such organisation may assign to it. Reasoning can be applied also to verify consistency among the norms in the organisation. The combination of some norms can result in contradictions, for example, when a prohibition occurs simultaneously with an obligation or permission. These contradictions can appear when considering the missions of just one isolated role, or when combining the missions of two or more roles.

As commented in previous subsection, Figure 4.9 shows that *Rules*, coded in SWRL, can be inherited from the base OntoMAS ontology, and new *Rules* can be added particularly to a specific extension and instantiation of OntoMAS, when defining a desired MAS scenario (these are in bold). All elements in the ontology are taken into consideration when semantic reasoners are executed for making inferences such as the ones depicted in Figure 4.9. For example, one general rule is that if an agent A is in a space S, and this space S can provide an observable property P, then it can be inferred that the agent A is able to perceive P if it chooses to do so. This rule is coded as follows:

$$is\text{-}in(?a,\ ?s),\ provides\text{-}percept(?s,\ ?p)\ {-}{>}\ can\text{-}perceive(?a,\ ?p).$$

Rules created specifically for this scenario state that observations of ball position and score take place in spaces defined as soccer fields. As can be noted, in such reasoning mechanism it is allowed to relate elements from any dimension (e.g., agent) with elements from another (e.g., environment).

Lets suppose now a more complex example for inferences about a modelled MAS. We already commented that agents join in organisations by playing organisation defined roles, and it is expected that such agents have in their codes the required plans to handle the goals that the organisation may deliver to them. Organisation goals are assigned to agents, for example if there is an obligation norm on that role, and an agent that adopted it should have a plan for achieving that goal. Lets represent this with a new property to specify that $Agents\ should\text{-}have\text{-}plan\text{-}for\ Goals$. This can be inferred, for example, if there is an obligation norm N that targets a role R, and there is an agent A that adopts the role R, then, the conclusion is that the agent A should have a plan for the goal G, where G is a goal of the mission M, which is a mission of the obligation norm N. The following rule exemplifies how to make this inference:

$$\begin{aligned}&ObligationNorm(?n),\ targets\text{-}role(?n,\ ?r),\ adopts\text{-}role(?a,\ ?r),\\&targets\text{-}mission(?n,\ ?m),\ has\text{-}goal(?m,\ ?g)\ {-}{>}\ should\text{-}have\text{-}plan\text{-}for(?a,\ ?g).\end{aligned}$$

As we have exemplified using some rules in this subsection and in previous ones, each time even more complex information can be inferred from the basic conceptualisation proposed by OntoMAS. Also, it allows extensions to be made on top of it, by including for example new concepts, properties, and so on. Next Section evaluates the OntoMAS modelling methodology and establishes a comparison with Prometheus in the context of designing JaCaMo projects.

5. EVALUATING ONTOMAS IN THE MODELLING OF MAS

“There are no such things as applied sciences,
only applications of science."
Louis Pasteur — (1822 - 1895)

This Section presents our empirical results obtained when the proposed modelling approach based on OntoMAS was used in practice. Despite investigating the use of this approach, it is interesting to compared OntoMAS with the most closely related methodology for MAS modelling reported in literature: Prometheus [PW03, PTW08]. In order to obtain results on comparative experiments, the same scenario should be specified using both approaches. Thus, this Section explains the details on our empirical results when both OntoMAS and Prometheus were put in practice for modelling the same agent systems.

The round of experiment for evaluating OntoMAS and comparing it with Prometheus was conducted with a group of 5 Graduate Computer Science students engaged in a course named “Multi-Agent Systems” at PUCRS in 2016. They have declared their expertise and profile as follows:

- **Multi-Agent Systems and JaCaMo:** 3 participants only had their first contact with MAS and JaCaMo in that course, while 2 had already a previous practical experience with these two topics.

- **Ontology and Protégé:** 3 participants had previous knowledge in ontology and Protégé before this course, whereas 2 only had their first contact with ontologies during the experiment. Although 3 participants claimed to know ontologies, none of them had any previous knowledge about OntoMAS (until beginning the experiments).

- **Prometheus:** their first contact with Prometheus was only in the course, during the experiment.

The experiments were conducted as follows. Initially, all participants were presented with a explanation and a demonstration of the two approaches – OntoMAS and Prometheus. The tool for working with Prometheus models was the Eclipse plug-in called PDT, and the tool used for the OntoMAS approach was Protégé. They were guided through a learning and an experiment scenario. The learning scenario was an adapted specification of the JaCaMo Hello World[1], and it can be seen in Appendix B. The experiment scenario was an adapted specification of the Gold Miners simulation[2], and it can be seen in Appendix C. In the end, all participants have used both approaches for modelling a specific MAS, but

---

[1] Obtained from http://jacamo.sourceforge.net/tutorial/hello-world/
[2] Obtained from https://multiagentcontest.org/2007/

they were divided into two groups. Each group started with a different methodology to avoid bias, and then switched to use the other methodology in the same scenario (3 have started with Prometheus, and 2 have used first OntoMAS). Figure 5.1 summarises this sequence of activities conducted in the experiments for comparing the modelling approaches. Each of these steps took place into the weekly two hours encounters of activities in classroom.

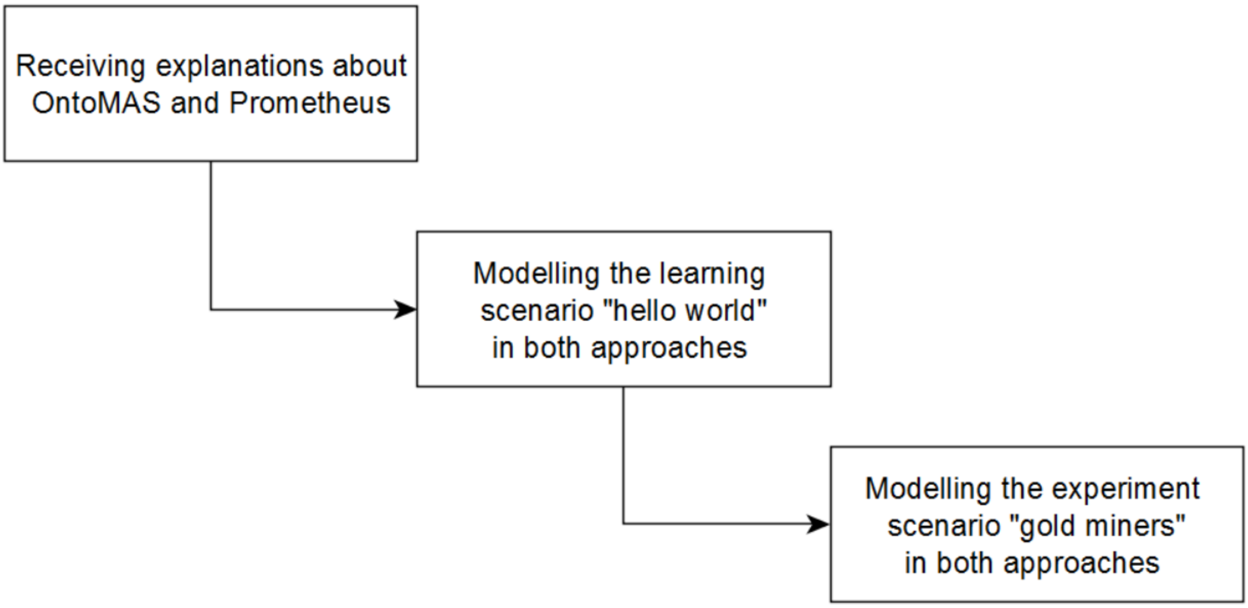

Receiving explanations about OntoMAS and Prometheus
Modelling the learning scenario "hello world" in both approaches
Modelling the experiment scenario "gold miners" in both approaches

Figure 5.1 – Sequence of activities conducted in the modelling experiments.

5.1 Evaluations on the use of OntoMAS X Prometheus

After modelling the same scenarios of MAS using the two different approaches, the participants were given a 5-point Likert scale [Lik32] for assessing the following assertions (with “X” being replaced by Prometheus or OntoMAS in each affirmation):

- **A1.** To implement a system in JaCaMo, I find it **easy to specify** models in X.
- **A2.** I could **understand** the elements provided by the X approach.
- **A3.** The characteristics of JaCaMo are **correctly represented** in the X approach.
- **A4.** The X approach is **complete** since it covers all the essential elements of JaCaMo systems.
- **A5.** I believe that it is **useful** to represent my JaCaMo system in the X approach.
- **A6.** The X approach provides **good support for MAS programming** in JaCaMo.

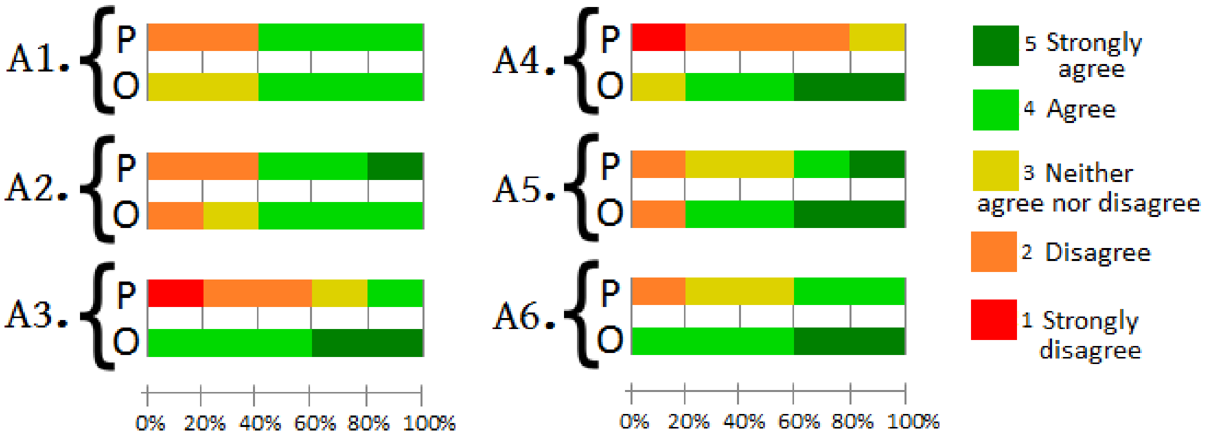


Figure 5.2 – Comparing Prometheus (P) and OntoMAS (O) (first shown in [FBV17]).

Our comparison of the two approaches under those criteria is summarised in Figure 5.2, in which, it is important to mention, all participants have answered to us anonymously through a web page. In general, we have observed better acceptance towards the ontology model, with the exception of assertion 2 (A2), regarding how easy it is to understand the elements, and this may also be due to the fact that OntoMAS was considered more complete (A4). OntoMAS was considered more correctly aligned with JaCaMo (A3) and, therefore, more useful for MAS modelling in this context (A5). The participants' opinions were more positive for OntoMAS than Prometheus in the issue of supporting programming as well (A6). Although, until this point, this group of participants did not work with the programming part yet, which took place after this part of evaluating the modelling aspects (experiments related with programming can be found in Section 7).

The questionnaires made with the participants have also asked for them if they wish to make comments, reviews, suggestions, and/or critics to the two approaches. For instance, we want to know if, according with their understanding, the meaning of something was confusing or wrong for representing JaCaMo properties, and if something was missing in order to make full designs of MAS as JaCaMo projects. We have obtained the following feedback:

- One of the participants have opined that Prometheus is more generic and does not cover some JaCaMo aspects, and that OntoMAS covers well the properties of JaCaMo, but the specification may be more laborious and sometimes less intuitive.

- A participant observed that OntoMAS serves best the project needs than Prometheus, since it is possible to better represent in OntoMAS what is intended to be implemented in JaCaMo. It was suggested that the experience of using ontology might be better with a complete and detailed manual describing every part of it, since many times the participant did not know which abstraction to use to solve some modelling problems that were faced.

- Another participant pointed out that the use of Prometheus is more natural because it is a visual tool, with easily defined flows, however it does not provide all the necessary functionality to define a complete MAS to be later developed through JaCaMo. OntoMAS was claimed to be more complete, but with a slower learning curve due to its various concepts and relations that can be modelled.

- According to another participant, the modelling in Prometheus is more intuitive because it has a graphical user interface with drag-and-drop. However, the approach that uses ontology through Protégé provides more options, so they were able to do a much more complete modelling. It was suggested that, after finishing the ontological modelling, it would be interesting to export it using some sort of visual notation such as in the form of diagrams.

- Another suggestion was that the Protégé does not block users from creating an inconsistency in their modelled system. It, however, finds the inconsistencies when a semantic reasoners is executed over the ontology. In some situations it would be interesting to have a way to interact with OntoMAS that prevents the user from creating relationships that do not make sense.

As can be observed from the evaluations and comments obtained with the participants until this part of the experiment, in terms of correct representing the characteristics of JaCaMo, the ontology approach seems to be playing a better role than Prometheus. Figure 5.3 focuses on a graph that compares the data about the assertion A3, previously depicted in Figure 5.2, to emphasise this point.

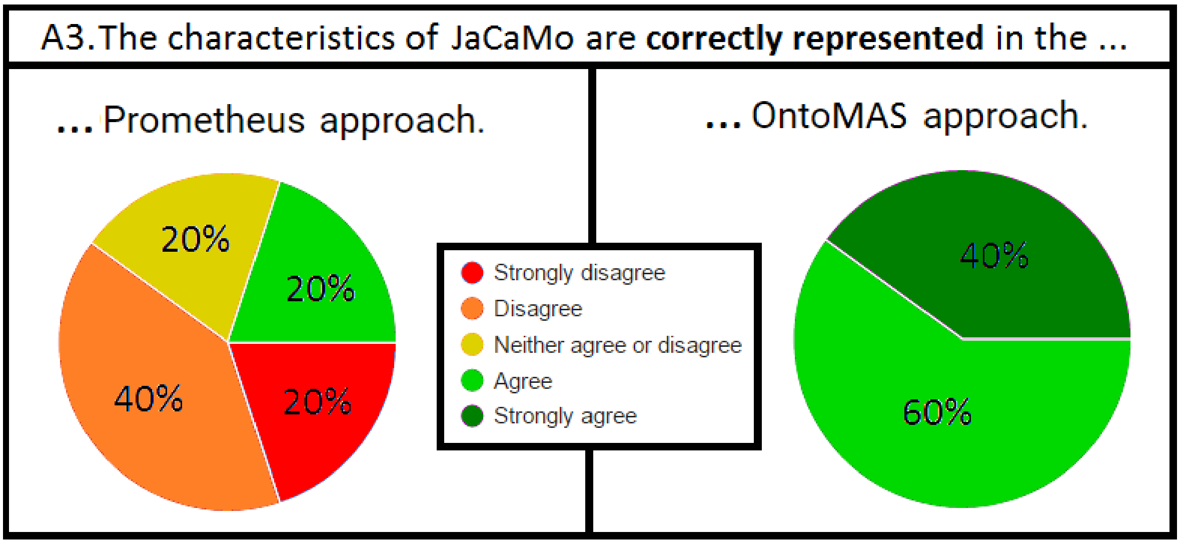


Figure 5.3 – Correctly representation of JaCaMo characteristics in each approach.

Thus, in summary, our results indicate that: the ontology seems to be considered as presenting a small advantage in terms of easiness of use (A1); it does not seem to have much difference among the two approaches in the criteria of the level of understandability regarding the elements provided in each approach (A2); modelling JaCaMo projects with our ontology was considered more complete than modelling in Prometheus (A4); on the topic of which approach is more useful to represent JaCaMo systems, the ontology got better votes than Prometheus (A5); and before the participants knowing how these modelling approaches would actually provide benefits for coding, their opinions were more inclined towards OntoMAS than Prometheus in the issue which modelling approach may be better for supporting programming of JaCaMo projects (A6).

Our initial evaluations of OntoMAS can be found in one of our journal paper [FBV17]. In that occasion, we have indicated some advantages for using ontology as proposed, such as *(i)* standardisation of structures and concepts; *(ii)* more expressiveness to represent further MAS concepts than with just the diagrams of Prometheus and Moise; *(iii)* just one specification to model a complete MAS; *(iv)* avoid inconsistencies when naming the MAS elements (especially in projects with several people); and *(v)* the relation between MAS components can become clearer, such as among agents and actions. Participants from that previous evaluation highlighted that the ontology is more flexible to embrace the needs of MAS, and it groups in one place what other approaches represent in several separate diagrams. Also, they found Prometheus and Moise diagrams more intuitive at first, but without covering as many details about the MAS environment as the proposed ontology does. Some drawbacks were identified in that occasion too, it was commented as disadvantages that: *(i)* specify an ontology is not a trivial task (previous knowledge of the concepts is required); *(ii)* it can be more confusing to specify a MAS in an ontology; and *(iii)* the Protégé can be complex to edit the ontology (however other software could be used to improve productivity).

Also, we have conducted a case study [FCVB16] in which one application was modelled in Prometheus considering its future implementation in JaCaMo, which led us to the identification of some gaps in the use of Prometheus in the context of JaCaMo projects. Next, we evaluate some characteristics obtained from the OntoMAS and Prometheus models developed by the participants in this part of our experiment.

5.2 Comparing the models created in Prometheus and OntoMAS

This subsection presents a comparative evaluation that focuses on analysing the models obtained from the use of the two different modelling approaches: OntoMAS and Prometheus [PTW05]. All participants of the experiment have developed their models for the same scenario of MAS (Gold Miners, described in Appendix C) using these two approaches and having in mind its implementation in the JaCaMo [BBH$^+$13] programming platform.

Our analysis of the models created by the participants aims at verifying the presence of some key elements that should be taken into consideration when designing a JaCaMo project. These elements are depicts in the first column of Table 5.2. Our first observation is that there are some MAS elements of JaCaMo that simply could not be represented in Prometheus, such as the idea of $Groups$ and $Norms$. Then, some elements can not be found directly in Prometheus, but there are some elements with “similar” semantics that could be used for that purpose. This is the case of $Data$ that can be considered as $Belief$, $Actors$ can be seen as the $Artifacts$, and $Actions$ may be used to refer to $Operations$. A third point on expressiveness and representativeness, is that in Prometheus we do not have these two layers offered by an ontology to represent types of elements (using subclasses) and the individuals (using instances). For example, it is not possible to use the only element provided in Prometheus that refers to $Agents$ to make a reference both to the types of agents as well as the individual agents.

With these observations in mind, some JaCaMo elements were took place in the same percentage of the models defined by the participants in both approaches: $Agents$, $Goals$, and $Messages$. The models produced in OntoMAS more frequently contained $Plans$, $Beliefs$, and $Artifacts$, while the models created with Prometheus considered more often $Percepts$, $Operations$, and $Roles$. As already mentioned, $Groups$ and $Norms$ can not be represented in Prometheus.

Table 5.1 – Comparing the presence of elements in Prometheus and OntoMAS models in some of the key elements of JaCaMo projects.

	Percentage of participants that defined the corresponding elements in ...	
MAS Elements	**... Prometheus**	**... OntoMAS**
Agents	100%	100%
Goals	100%	100%
Messages	20%	20%
Plans	0%	**40%**
Beliefs	40% (if considering Data=Belief)	**80%**
Artifacts	60% (if considering Actor=Artifact)	**100%**
Percepts	**100%**	80%
Operations	**100%** (if considering Action=Operation)	60%
Roles	**100%**	60%
Groups	Can not be represented.	80%
Norms	Can not be represented.	40%

As we analyse the modelling methodologies and the models produced with them with an eye on JaCaMo, we observe that JaCaMo developers lack a specialised approach for their projects. Existing approaches, such as Prometheus, are not fully compatible and suitable for the abstractions provided in JaCaMo. In this directions, agent-oriented software engineers would have the option of not using a modelling methodology, which would result in some limitations and problem. Another alternatives would be to adapt an existing methodology (e.g., Prometheus), or create a new approach highly oriented to and integrated with the desired coding platform (as done in this thesis).

Prometheus [PW03] demonstrates, by means of PDT, a level of integration and alignment with the JACK agent programming platform. PDT is a graphical tool that supports the Prometheus methodology in order to build the design of MAS [PTW08]. It started as a stand alone tool, but it is nowadays being developed as a plug-in for Eclipse. PDT offers, among several interesting features, the possibility of generating skeleton code for the JACK platform, when the detailed design is completed. However, as we investigated in one of our papers [FCVB16], PDT and Prometheus have some issues when used for developing MAS for JaCaMo. On top of that, we may also argue that JaCaMo has demonstrated important advantages for its programmers and users when compared with JACK.

This Section have analysed and discussed the results for the experiments comparing our approach with Prometheus for modelling MAS in the context of JaCaMo projects. An extended comparative analysis between the models built in each of the approaches, and comparisons of each approach for other stages of AOSE is planned as future work. Although the results reported here are limited to the given population and applications that took place in the experiments, they help to indicate some interesting advantages regarding the application of the approach proposed in this thesis for modelling MAS. Next Section presents our techniques for supporting the programming of such systems in JaCaMo using models specified in OntoMAS.

6. TECHNIQUES FOR PROGRAMMING USING ONTOMAS

> *“The art of programming is the art of organising complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."*
>
> *Edsger Dijkstra* — (1930 - 2002)

This thesis proposes two different techniques for code generation based on models specified using OntoMAS. One technique is the iterative drag-and-drop of elements from the ontology to transform them into the different parts of code that compose a JaCaMo project: Jason, Cartago, Moise, or the jcm file. This first part is described in the subsection 6.1, and its explanation is complemented by the Appendix D. The other technique is the automatic generation of the initial files and code of a JaCaMo project that matches the ontology-specified content. This second part is described in the subsection 6.2, and the explanation is complemented by the Appendix E. Our tool that implements both techniques, called OntoJaCaMo, is shown in subsection 6.3.

Initially, lets make a mapping of where elements from OntoMAS are usually found in a JaCaMo project. Each subclass of **Agent** is found in Jason as .asl file, while its instances are usually found as the individual agents defined by an agentID in the .jcm file. Instances of **Plan** are found inside the .asl code of the type of agent that contains such plan, and instances of **Action** are usually found in Jason in the body of agents’ plans. Instances of **Goal** and **Belief** are usually found inside the code of agents (.asl files). Also, sending a **Message** may be a part of a plan in agents.

Each subclass of **Artifact** is found in CArtAgO as a Java class, and instances of **Artifact** subclasses represents a real object of that type, which may be found in JaCaMo in the .jcm file that describes the initial artifacts of a system, however other artifacts may be created after the initialisation of the MAS. **Spaces** are initialised in JaCaMo at the .jcm file, but agents may make reference to spaces in their code (the .asl files). **Operations** are found in CArtAgO as methods of the artifact that implements such procedures. Instances of **Percept** (**ObservableProperty** or **ObservableEvent**) are found inside the java code of artifacts through methods provided by the CArtAgO API to manipulate them (such as defineObsProperty, getObsProperty, updateObsProperty, and signal).

Subclasses of **Group** can be found in the XML that specifies an organisation in Moise, and their instances can be found in the .jcm file of JaCaMo, as well as in the code of agents in Jason that can make references to groups (e.g. join_group). Instances of **Role** are found also in the Moise XML file, and the code of agents in Jason can make reference to such roles (e.g. adopt_role). Instances of **OrganisationGoal** are also found in the Moise file, and the code of agents in Jason can make references to those goals (for example, agents may have plans to act when a goal is assigned to them from the organisation). Lastly, instances of **Missions** and **Norms** are also defined in the Moise XML file of a JaCaMo project.

6.1 Drag-and-Drop Transformations from OntoMAS to JaCaMo

The idea of using an ontology in a tool for providing drag-and-drop operations from models to code in JaCaMo has been published initially in [FHM$^+$15], and most recently in [FBV17]. The elements of the ontology can be dragged to generate code for the different parts of JaCaMo, such as Jason, CArtAgO, Moise, or the JCM file that corresponds to the specification that initialises a JaCaMo MAS. How each element from OntoMAS models can be transformed through drag-and-drop for Jason code is depicted in Table 6.1, which means when a programmer is dropping an element from ontology in the .asl file that corresponds to the code of a Jason agent. Table 6.2 shows this type of transformations for CArtAgO, which is coded in a Java class. Table 6.3 specifies the drag-and-drop from OntoMAS to Moise specifications in XML, and Table 6.4 considers the conversions for the JCM file, which is responsible for the initialisation of MAS project in JaCaMo. Thus, these tables contemplate the technique of making drag-and-drop transformations from OntoMAS models to each part of programming in JaCaMo (Jason, CArtAgO, Moise, and JCM file).

To exemplify the drag-and-drop conversions, lets take a loot at how instances of the **ObservableProperty** concept may be employed in the code of each of the different parts of JaCaMo. Suppose an instance of **ObservableProperty** called $temperature$, this is characterised from the environment dimension, so if a programmer makes a drag-and-drop of $temperature$ in this dimension, a code suggestion may be to update the value of such observable property. Thus, the following code may be created:

$getObsProperty(temperature).updateValue(newValue);$

In Jason, making a drag-and-drop using this same instance of **ObservableProperty** may give origin to a plan triggered by the observation of such property:

$+temperature : true <\!- planBody.$

However, if dropped in the middle of a plan, then just the corresponding belief identified by $temperature$ may be generated. When a JaCaMo system is running, the observable properties provided by artifacts in the environment becomes beliefs to agents that are focused on these artifacts, and when becoming beliefs some plans may be triggered by the belief addition event. Instances of observable properties are not applicable for drag-and-drop code transformations in the case of Moise or JCM file.

We have condensate the information about the drag-and-drop operations for transforming OntoMAS to JaCaMo code in these 4 tables (6.1, 6.2, 6.3, and 6.4). Appendix D complements the explanation about this topic in a textual manner.

Table 6.1 – Drag-and-drop code generation for Jason from ontology elements.

Instance of the Ontology Element	**Drag-and-Drop Code for Jason**	**Explanation**
Agent Dimension		
ExternalAction	actionName();	an external action invocation inside a plan's body representing an agent acting in the environment.
InternalAction	.actionName();	an internal action invocation inside a plan's body representing an action that an agent performs mentally.
Agent	agentName	the identification of the individual agent in order to send messages, or perform some other tasks.
Belief	+beliefName[source(value)];	a belief addition event with source's value defined by the belief's subtype: self, percept, or other agent.
AchievementGoal	!achievementGoalName;	an initial goal for that agent; or a goal that has to be achieved during the execution a plan.
TestGoal	?testGoalName;	a goal that has to be tested during the execution of a plan.
Message	send(receiver, illocutionaryForce, propositionalContent);	the act of sending the corresponding instance of message
Plan	is_triggered_by : true <- actions; goals.	a plan with its triggering condition, context (that by default is true), and a body composed (mainly) of actions and goals.
Environment Dimension		
Space	joinWorkspace("workspaceName");	an action for that agent to join the corresponding workspace.
Artifact	focus(artifactName);	the action of focusing on that instance of artifact.
Operation	operationName();	the invocation of the corresponding operation in the body of a plan representing the execution of that operation by an agent.
ObservableProperty	+propertyName : true <- planBody.	a plan triggered by the observation of the corresponding property.
ObservableEvent	+eventName : true <- planBody.	a plan triggered by the observation of the corresponding event.
Organisation Dimension		
Group	join_group(groupName);	an action of joining in the given group.
Role	adopt_role(roleName,groupName);	the action to adopt the given role in the specified group.
Mission	commit_mission(missionName, schemeId);	an action to commit with the instance of mission in the given scheme.
Norm	+normName: true <- planBody.	a plan triggered by the perception of the given norm.
OrganisationGoal	+!goalName : true <- planBody.	a plan triggered by the addition event of the specified goal.

Table 6.2 – Drag-and-drop code generation for CArtAgO from ontology elements.

Instance of the Ontology Element	**Drag-and-Drop Code for CArtAgO**	**Explanation**
Agent Dimension		
ExternalAction	operationName();	an operation invocation within a method representing an artifact executing the given operation.
InternalAction	it is not applicable for drag-and-drop code transformations for CArtAgO.	
Agent	agentName	an individual agent so the environment may be programmed to react accordingly.
Belief	getObsProperty(perceptName). updateValue(newValue);	if it is a percept-type belief, then it updates the value of the corresponding observable property[a].
AchievementGoal	signal(!achievementGoalName);	an information that artifacts can send to agents in order to initiate the achievement of a goal.
TestGoal	signal(?testGoalName)	an information that artifacts can send to agents in order to initiate the test of a goal.
Message	signal(!send(receiver, act, value));	a signal where an artifact requests the sending of the corresponding message.
Plan	signal(triggeringCondition);	a signal with the triggering condition corresponding to that plan.
Environment Dimension		
Space	updatePosition(new AbstractWorkspacePoint());	an update to the position of this artifact.
Artifact	ClassName individualName = new ClassName();	the declaration and initialisation of a new artifact.
Operation	operationName();	the invocation of the given operation representing an artifact executing the related operation.
ObservableProperty	getObsProperty(propName). updateValue(newValue);	an update in the value of the observable property.
ObservableEvent	signal(eventName);	the generation of a signal of the observable event.
Organisation Dimension		
Group, Role, Mission, Norm, OrganisationGoal	it is not applicable for drag-and-drop code transformations for CArtAgO.	

---

[a] For beliefs in the types of self or other agent, there is no related code to generated regarding the environment.

Table 6.3 – Drag-and-drop code generation for Moise from ontology elements.

Instance of the Ontology Element	Drag-and-Drop Code for Moise	Explanation
Agent Dimension		

Plan	operator=“sequence”>
	id=“goalN/></plan>
		the description of how the plan is decomposed in its goals.

ExternalAction, InternalAction, Agent, Belief, AchievementGoal, TestGoal, Message	it is not applicable for drag-and-drop code transformations for Moise.	
Environment Dimension		
Space, Artifact, Operation, ObservableProperty, ObservableEvent	it is not applicable for drag-and-drop code transformations for Moise.	
Organisation Dimension		

Role	id=“roleName”>
	role=“extendsRole”/></role >
		a role definition which may extend some other role.

Mission	id=“missionName” min=1 max=1>
	id=goal1/>...id=goalN/></mission>
		the specification of the mission which contains a set of goals.

Norm	id=“normName” type=“normType”
	role=“targetRole” mission=“targetMission”/>
		the specification of a norm that targets the given role and mission.

OrganisationGoal	id=“goalName”>	the specification of a goal.
Group	it is not applicable for drag-and-drop code transformations for Moise.	

Table 6.4 – Drag-and-drop code generation for JCM file from ontology elements.

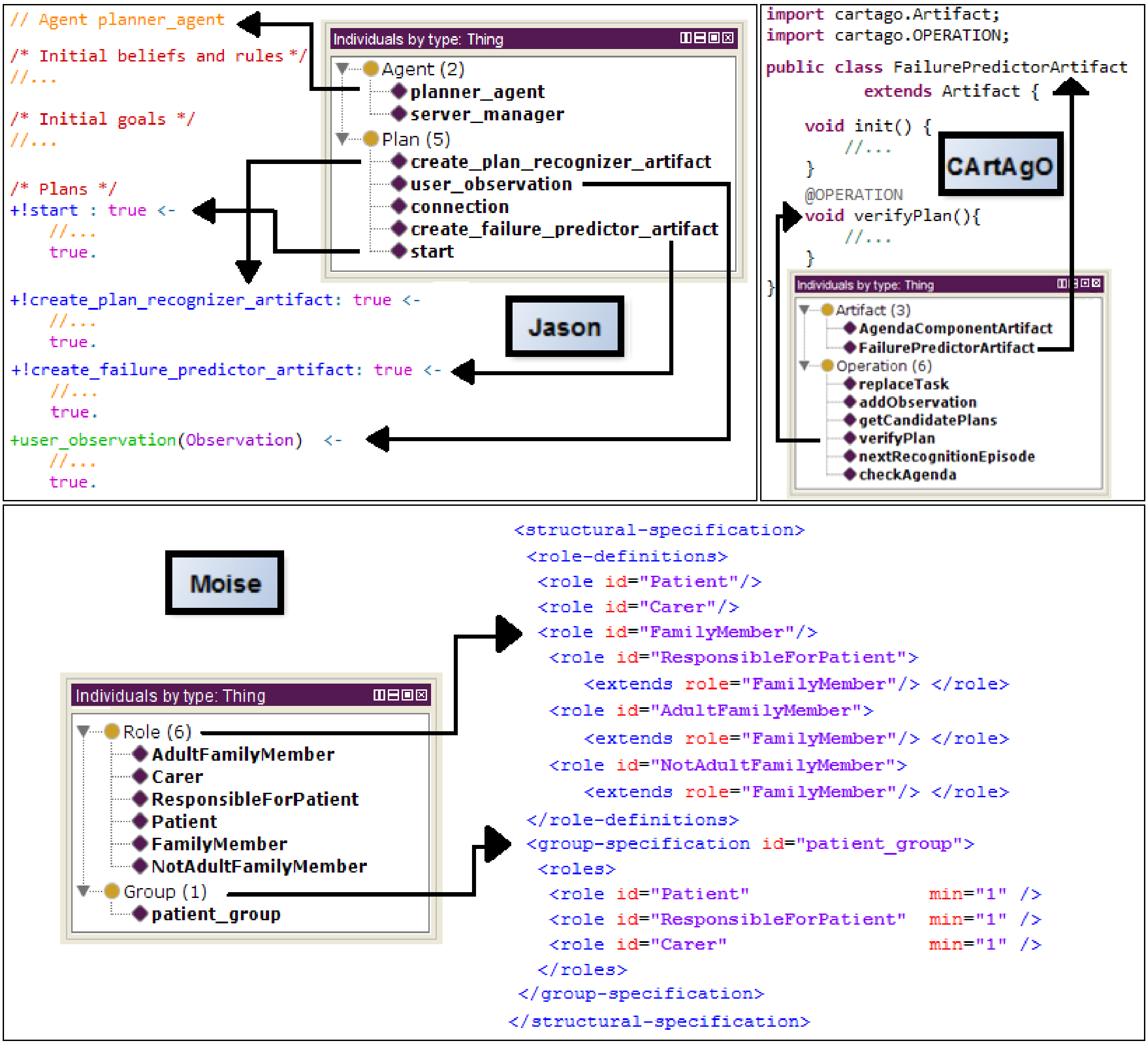
Instance of the Ontology Element	Drag-and-Drop Code for the JCM file	Explanation
Agent Dimension		
Agent	agent agentName : agentType.asl { parameters }	an instantiation of an individual agent of the given type
Belief	beliefs : beliefName	the definition of an initial belief, if it is a self-type belief[a].
AchievementGoal	goals : goalName	the definition of an initial goal.
ExternalAction, InternalAction, TestGoal, Message, Plan	it is not applicable for drag-and-drop code transformations.	
Environment Dimension		
Space	workspace spaceID { parameters }	the definition of a workspace.
Artifact	artifact artifactName : className()	an instance of artifact of the given type.
Operation, ObservableProperty, ObservableEvent	it is not applicable for drag-and-drop code transformations.	
Organisation Dimension		
Group	group groupName : groupType { parameters }	an instance of group of the given type.
Role	roles : roleName in groupName	the definition that an agent will be playing the given role in a group.
Mission, Norm, OrganisationGoal	it is not applicable for drag-and-drop code transformations.	

---

[a] For beliefs in the types of percept or other agent, there is no related code to generated regarding the initial configuration file.

6.2 Initial JaCaMo Project Generation from OntoMAS Models

The idea of using an ontology for the automatic generating a skeleton code for each of the JaCaMo languages has been published initially in [FSP$^+$15]. Figure 6.1 illustrates ontological elements and their resulting code counterpart in Jason, CArtAgO, and Moise. According to this image, instances[1] of some concepts of an ontology are being transformed into code for JaCaMo.



```
// Agent planner_agent

/* Initial beliefs and rules */
//...

/* Initial goals */
//...

/* Plans */
+!start : true <-
    //...
    true.

+!create_plan_recognizer_artifact: true <-
    //...
    true.
+!create_failure_predictor_artifact: true <-
    //...
    true.
+user_observation(Observation)  <-
    //...
    true.
```

```
import cartago.Artifact;
import cartago.OPERATION;

public class FailurePredictorArtifact
              extends Artifact {

    void init() {
       //...
    }
    @OPERATION
    void verifyPlan(){
       //...
    }
}
```

```
<structural-specification>
 <role-definitions>
  <role id="Patient"/>
  <role id="Carer"/>
  <role id="FamilyMember"/>
   <role id="ResponsibleForPatient">
      <extends role="FamilyMember"/> </role>
   <role id="AdultFamilyMember">
      <extends role="FamilyMember"/> </role>
   <role id="NotAdultFamilyMember">
      <extends role="FamilyMember"/> </role>
 </role-definitions>
 <group-specification id="patient_group">
  <roles>
   <role id="Patient"               min="1" />
   <role id="ResponsibleForPatient" min="1" />
   <role id="Carer"                 min="1" />
  </roles>
 </group-specification>
</structural-specification>
```

Figure 6.1 – Converting ontology to MAS code (images adapted from [FSP$^+$15]).

While when using drag-and-drop programmers are iteratively transforming elements from OntoMAS into code, this other code generation technique uses another perspective,

---

[1] Currently, we claim that in some situations subclasses play the desired role better them instances.

which is to generate an initial structure of a corresponding project in JaCaMo to what is specified in the ontology model.

The generation of initial agent files and code for Jason considers mainly the subclasses and instances of the OntoMAS agent dimension. For example, we have that each subclass of **Agent** becomes an .asl file with its corresponding plans, actions, goals, beliefs, and messages. However, characteristics defined at other dimensions, such as the environment, although not directly applicable to generate the initial code at the agent level, may be considered to suggest implementation alternatives for programmers (at least for them to be aware of). For example, for an agent that is expected to receive a given percept, a plan triggered by the addition event of that percept may be suggested as a situation that is likely desired to be handled by the programmers.

Similarly, the initial files of the CArtAgO part of a JaCaMo project derive mainly from the environment dimension of OntoMAS, and the Moise initial code is generated based the organisation dimension. Subclasses of **Artifact** becomes the java files with their corresponding operations as methods, and observable properties initialised. And all the organisation elements (subclasses, instances, and relationships) are considered in the generation of the initial XML file of a Moise organisation. Lastly, the JCM file considers characteristics from all the three dimensions, and relationships from the integration of them.

To exemplify the initial project generation, lets take a loot at the instance of **ObservableProperty** that we were using previously, the $temperature$. If it is said that a type of artifact (e.g., $Computer$) has this property, then the definition of such observable property must appear inside the $init()$ method in the class of $Computer$ artifacts in the format:

$$defineObsProperty(``temperature", initialValue);$$

Considering Jason, Moise or JCM file, instances of observable properties are not directly applicable for automatic code generation in this case. However, a plan triggered by the addition event of the related observable property could be suggested to the programmer of the agents as a situation that could be desired to be handled.

How each element from OntoMAS models can be transformed into the initial structure of files and code for Jason is depicted in Table 6.5. This same principle applied to CArtAgO is explained in Table 6.6, for Moise in Table 6.7, and for the JCM file in Table 6.8. These tables contemplate the use of OntoMAS models as starting point to generate skeleton code to each part of programming in JaCaMo (Jason, CArtAgO, Moise, and JCM file). Appendix E complements the explanation about this topic in a textual manner.

Table 6.5 – Template code generation for Jason from ontology elements.

Instance of the Ontology Element	Base Code for Jason	Explanation
Agent Dimension		
Agent (subclass)	agentSubclass.asl file	a type of agent that contains all related elements such as plans, goals, and beliefs.
Belief	beliefName[source(value)]	an initial belief in the corresponding .asl file with the source's value defined by the belief's type (self, percept, or other agent).
AchievementGoal	!achievementGoal.	an initial achievement goal in the corresponding .asl file.
Message	!sendMsgName <- .send(receiver, illocutionaryForce, propositionalContent);	a plan to send the corresponding message. Also, a plan for the receiver agent may be created taking as triggering condition the receiving such propositionalConcent and illocutionaryForce.
Plan	is_triggered_by : true <- actions, goals.	a plan in which it has a triggering condition, a context, and a body composed (mainly) of actions and goals. The plan is inserted in the .asl file of the agent type that has it.
ExternalAction, InternalAction, TestGoal, Agent	not directly applicable for automatic code generation for Jason.	
Environment Dimension		
ObservableProperty	+propName : true <- planBody.	it is not directly applicable for automatic code generation. However, a plan triggered by the addition event of the related observable property may be suggested as a situation that is likely desired to be handled.
ObservableEvent	+eventName : true <- planBody.	it may be suggested (however, it is not mandatory) a plan triggered by the observation of the corresponding event.
Space, Artifact (instance and subclass), Operation	not directly applicable for automatic code generation for Jason.	
Organisation Dimension		
Group (instance and subclass), Role, Mission, Norm, OrganisationGoal	not directly applicable for automatic code generation for Jason.	

Table 6.6 – Template code generation for CArtAgO from ontology elements.

Instance of the Ontology Element	**Base Code for CArtAgO**	**Explanation**
Agent Dimension		
ExternalAction, InternalAction, Agent (instance and subclass), Belief, AchievementGoal, TestGoal, Message, Plan	not directly applicable for automatic code generation for CArtAgO.	
Environment Dimension		
Space, Artifact (instance)	not directly applicable for automatic code generation for CArtAgO.	
Artifact (subclass)	SubclassName.java	a new .java file composed with all related elements such as operations, and observable properties.
Operation	@OPERATION void operationName(){ }	a method representing the corresponding operation which may contain zero or more parameters.
ObservableProperty	defineObsProperty("propertyName",value);	the definition of the observable property inside the "init" method in the class of the related artifact.
ObservableEvent	signal(eventName);	the artifact’s class code may indicate that this signal can be sent by this artifact.
Organisation Dimension		
Group (subclass and instance), Role, Mission, Norm, OrganisationGoal	not directly applicable for automatic code generation for CArtAgO.	

Table 6.7 – Template code generation for Moise from ontology elements.

Instance of the Ontology Element	Base Code for Moise	Explanation
Agent Dimension		
ExternalAction, InternalAction, Agent (subclass and instance), Belief, TestGoal, AchievementGoal, Message, Plan	not directly applicable for automatic code generation for Moise.	
Environment Dimension		
Space, Artifact (subclass and instance), Operation, ObservableProperty, ObservableEvent	not directly applicable for automatic code generation for Moise.	
Organisation Dimension		
Group (subclass)	-specification id="groupName"> id="roleN"/></group-specification>	a declaration of a group specification that contains all related roles.
Group (instance)	not directly applicable for automatic code generation for Moise.	

Role	id="roleName">
	role="extendsRole"/></role>
		a role definition which may extend some other role.

Mission	id=""missionName" min="1" max="1">
	id="goalN"/></mission>
		the specification of the mission that contains a set of N goals.

Norm	id="normName" type="normType"
	role="targetRole" mission="targetMission"/>
		the specification of a norm which targets a role and a mission.

OrganisationGoal	id="goalName"/>	a specification of an organisational goal.

Table 6.8 – Template code generation for JCM file from ontology elements.

Instance of the Ontology Element	Base Code for JCM file	Explanation
Agent Dimension		
Agent (subclass)	although each subclass of Agent does not generate a specific base code for the JCM file, the sources that agent instances are allowed to have are given by these types.	
Agent (instance)	agent agentID : className.asl { parameters }	a declaration that instantiates an agent of the given type which may be configured by some parameters.
Belief	beliefs : beliefName	an initial self belief of the related instances of agents in the corresponding individual agents' definition.
AchievementGoal	goals : goalName	an initial achievement goal of the related instances of agents in the corresponding individual agents' definition.
ExternalAction, InternalAction, TestGoal, Message, Plan	not directly applicable for automatic code generation for JCM file.	
Environment Dimension		
Space	workspace spaceID { parameters }	the definition of a workspace which can be configured by some parameters.
Artifact (subclass)	although each subclass of Artifact does not generate a specific base code for the JCM file, the sources that artifact instances are allowed to have are given by these types.	
Artifact (instance)	artifact artifactName : className()	an instantiation of an artifact from the given class, in which it should be placed inside a workspace's declaration.
Operation, ObservableProperty, ObservableEvent	not directly applicable for automatic code generation for JCM file.	
Organisation Dimension		
Group (subclass), Mission,Norm, OrganisationGoal	not directly applicable for automatic code generation for JCM file.	
Group (instance)	group groupID : className { parameters }	the definition of a group with the given type which can be configured by some parameters.
Role	roles : roleName in groupName	the definition that an agent will be playing the given role defined in a specific group.

6.3 OntoJaCaMo Tool for Ontology-based Development of MAS

We implemented the techniques previously explained in subsections 6.1 and 6.2 in a software tool, which we refer to as OntoJaCaMo. It consists of a plug-in for Eclipse that loads an instantiated model based on the OntoMAS ontology to provide code generation for JaCaMo. Eclipse [Bud04] is an open source software development project that provides an IDE in which a basic unit of function, or a component, is called a plug-in. IDEs are software applications, which combine different development tools under a unified user interface. Eclipse is already the standard IDE for developing for JaCaMo, and it was indeed an interesting choice since Eclipse is pointed out as a mature IDE, and one of the most widely used by programmers [PB09].

The installation of OntoJaCaMo just requires that the inclusion of the .jar file corresponding to the plug-in in the directory named “plugins” in the folder where the Eclipse is located. The plug-in can be activated to appear visually in the graphical interface of Eclipse by following these sequence: $Window \rightarrow Show\ View \rightarrow Other... \rightarrow JaCaMo\ Ontology \rightarrow Ok$. Figure 6.2 shows how to follow these steps for activing the plug-in. When it is enabled, the plug-in requests to be informed about the OWL file corresponding to an OntoMAS instantiated ontology so that it can be loaded in OntoJaCaMo.

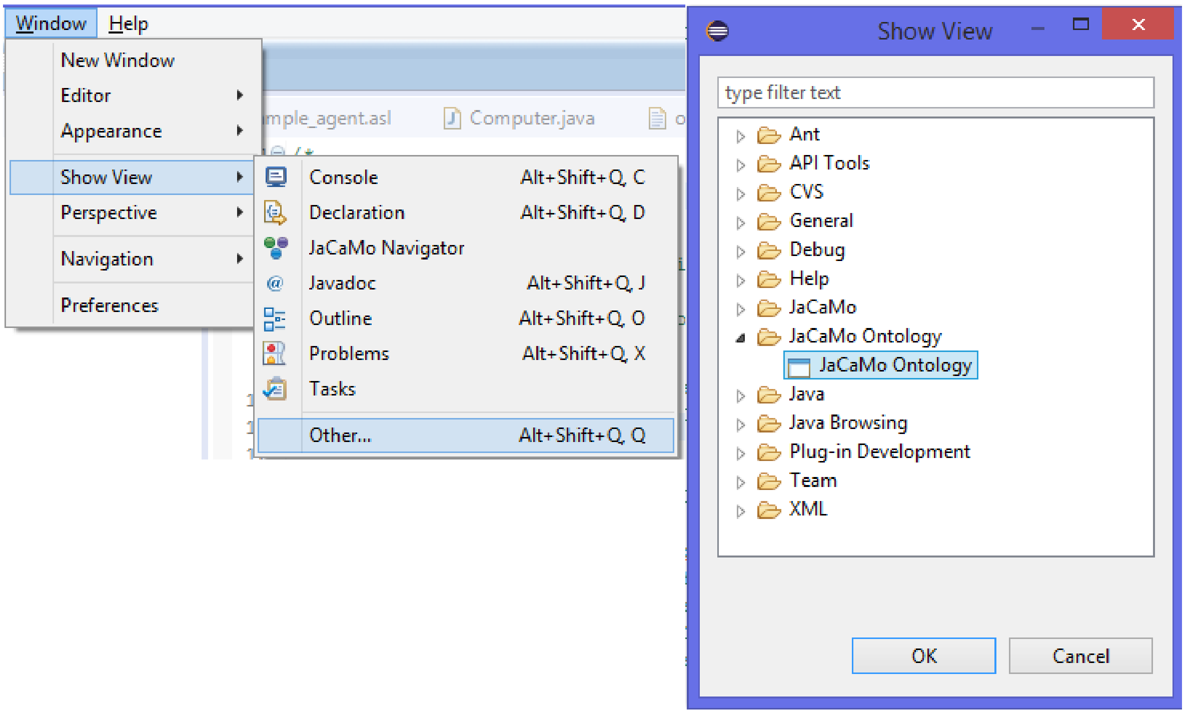
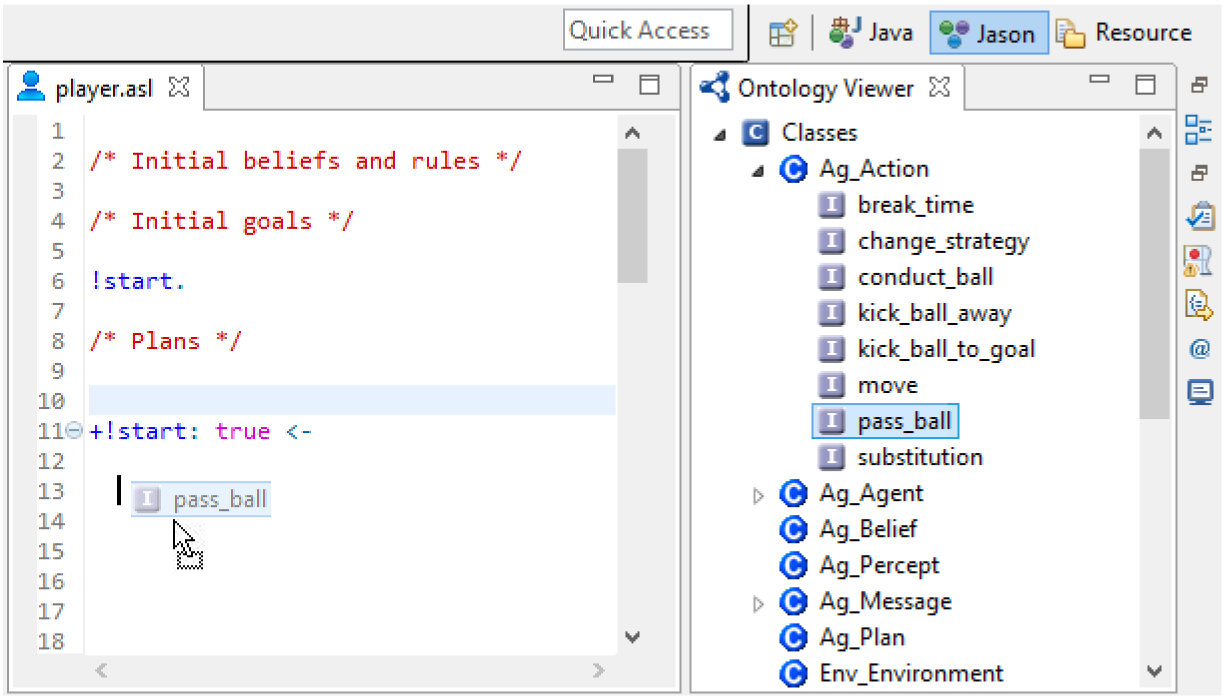


Figure 6.2 – Activating OntoJaCaMo in Eclipse (first published in [FBV17]).

The plug-in was designed to be used in the “JaCaMo Perspective” of Eclipse (or related perspectives, such as Jason). The tool loads OWL ontologies and provides three model-based programming features to generate MAS code: drag-and-drop, conversion from ontology to code, and auto-complete from instantiated ontologies. It was developed using the OWL API [HB11], which is an open source Java API (Application Programming Interface) for creating, manipulating, and serialising ontologies in the OWL format.

The drag-and-drop functionality from ontology to agent code can be seen in Figure 6.3, which depicts the Eclipse in Jason perspective. In the right side of the image, the developer can visualise and navigate in the ontology concepts, instances, and properties (from the new Eclipse component developed as part of this thesis). These elements from the model on the right side can be dragged to the left side that represents the AgentSpeak code of a Jason agent (in this case *player.asl*). As exemplified in Figure 6.3, the programmer is dragging and dropping the action *pass_ball* to be inserted in a plan of agents of type "player". Similarly, it is possible to provide developers the auto-complete feature from ontology to agent code, which is activated when the developer is typing MAS code (or press the shortcut "ctrl+space"). Then, the available options based on the ontology are presented to programmers as suggestions. One example is when coding the plan's context, which may be composed of ontology-based queries (e.g., verifying if an individual belongs to a concept).



```

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */


+!start: true <-
```

Figure 6.3 – Drag-and-drop in Eclipse for MAS coding (first published in [FHM$^+$15]).

To change the desired context of JaCaMo code to be obtained from the ontology click on the illustrated icon identified with the message "Choose target code platform" at the top of the plug-in (or activate this option by right-clicking inside the area that displays the ontology). Then, select which platform to target, and click "ok" as shown in Figure 6.4.

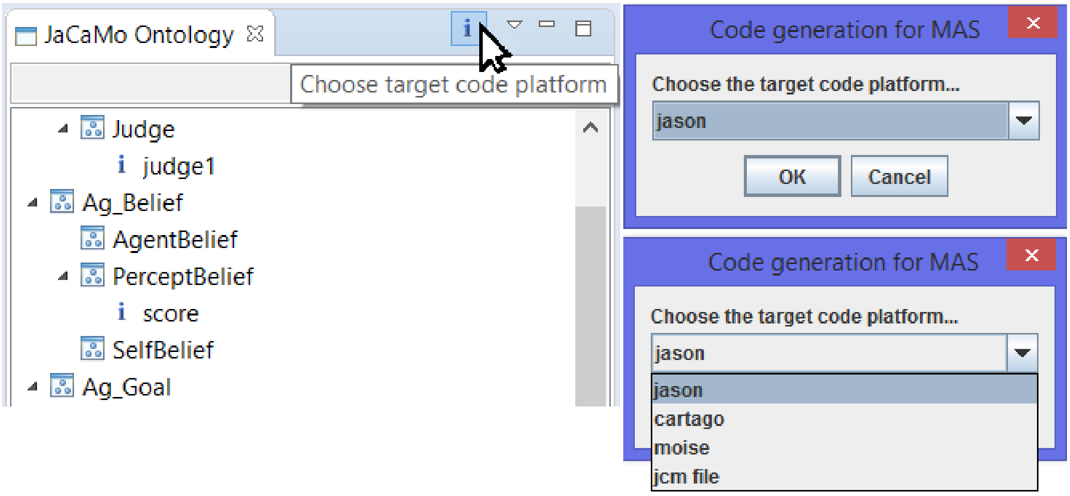


Figure 6.4 – Changing the JaCaMo target platform in OntoJaCaMo.

6.4 Considerations on the OntoJaCaMo Tool and its Techniques

For an effective and efficient software development it is essential that preferably all tasks and activities during the development process are adequately supported by tools [PB09]. A broad overview over the state of the art in the area of software tools [GH01] has identified 18 different kinds of tools (e.g., design tools, IDEs, as well as testing and debugging tools). The quality of any tool support can be assessed by considering the degree of support for the different phases and tasks [PB09], e.g., design tools, which besides the creation and editing of design models also often support consistency checking and/or code generation.

Our presented tool is able to generate code fragments based on design information, which is known as forward engineering [PB09]. This is in the opposite direction of extracting design information out of existing application code, the so called reverse engineering. A drawback of forward or reverse engineering techniques is that after a once generated artifact has been changed manually, forward or reverse engineering cannot be reapplied without loosing the changes, as stated by the called ”post editing problem´´. The combined support of forward and reverse engineering, such that changes in one artifact can always be merged into the other without compromising consistency or loosing changes, is referred as round-trip engineering [PB09]. Currently, the tool presented in this thesis does not address yet such advanced and complex concepts of synchronisation.

This Section have described our techniques for supporting the programming of MAS based on OntoMAS models and the implementation of such techniques in OntoJaCaMo. In next Section, we present the empirical results that we have obtained from putting these techniques and OntoJaCaMo in practice to support MAS development.

7. EVALUATING THE PROGRAMMING TECHNIQUES AND TOOL

> *“Each problem that I solved became a rule, which served afterwards to solve other problems."*
>
> *Rene Descartes* — (1596 - 1650)

The previous Section of this thesis has explained our proposed model-based programming techniques for supporting AOSE and their implementations in the OntoJaCaMo tool. In this Section, we provide the complete details on the empirical results obtained when such techniques and tool were put in practice for developing different agent systems.

This part of evaluating OntoJaCaMo and its programming techniques was conducted with the same group of 5 participants that have joined our evaluation of OntoMAS. Thus, we refer to Section 5 for details on the participants profile and background knowledge.

This part of our experiments took place after the participants have finished the experiments in which OntoMAS was used for the modelling of agent systems (described in Section 5). Before starting the experiments regarding the evaluation of the programming techniques implemented in OntoJaCaMo, the participants received the required prior instructions on these topics in order to perform the activities with the minimum required knowledge, such as, for example, how to load and how to use OntoMAS models in OntoJaCaMo.

The participants received the OntoJaCaMo plug-in, where they had to load their previously instantiated ontology models and use the tool to support the model-based development of their agent code. Different from Section 5 in which all participants created OntoMAS (and Prometheus) models for the same scenarios, in the evaluation of the programming part, each participant had defined his/her own unique scenario to work with. Then, such specification should be modelled using OntoMAS and developed using OntoJaCaMo.

Subsection 7.1 explains the feedback obtained from the use of OntoJaCaMo with its drag-and-drop model-based programming technique through the evaluation provided by the participants. Subsection 7.2 makes a comparison and an analysis between the model specified in OntoMAS, and the JaCaMo code that was delivered, to point out the importance of generating the initial project code from models as proposed by our other model-based programming technique.

7.1 Evaluations on the Use of Drag-and-Drop and OntoJaCaMo

After finishing the programming of their MAS using the drag-and-drop provided by OntoJaCaMo, the participants were surveyed by means of questionnaires to extract their perceptions and opinions about the techniques and tool. The participants were queried through affirmations using a 5-point Likert scale [Lik32]. Also, all participants have answered to us anonymously in a web page. Their answers can be seen in the frequency diagram depicted in Figure 7.1, which investigates the following affirmations:

- **Easy to understand:** it is easy to understand the functioning of the OntoJaCaMo;
- **Easy to use:** OntoJaCaMo is easy to use;
- **Faster programming:** OntoJaCaMo enables faster programming of MAS, thus it can be considered an efficient tool;
- **Easy to visualise:** it is easy to visualise the ontology components in the way they appear in the OntoJaCaMo graphical interface;
- **Clear code generation:** the code generation is clear and it is intuitive how the ontology elements are transformed into MAS code (e.g., drag-and-drop);
- **Improved model-code transition:** OntoJaCaMo approximates and improves the transition between the specification/modelling and the programming of MAS;
- **Coding support:** OntoJaCaMo offers support for the coding of MAS;
- **Avoid mistakes:** programmers make less mistakes or inconsistencies when the code is generated from OntoMAS models;
- **Advantages for programming:** OntoMAS and OntoJaCaMo can be useful when programming, as they bring developers new functionalities without impeding the use of other development tools (in this case, Eclipse); and
- **Better JaCaMo understanding**: OntoJaCaMo promotes an improved learning and a more didactic understanding about JaCaMo.

Some criteria have received only positive evaluations from all participants, such as that OntoJaCaMo is easy to understand, provides coding support, offers advantages for programming, and enables a better understanding of JaCaMo. OntoJaCaMo was evaluated negatively by only one participant, and only in one issue, which says that it has a clear code generation in the sense that it is intuitive how elements from the ontology are transformed into MAS code. In the remaining criteria, despite being well evaluated by the majority, at least one participant have chosen to neither agree or disagree, which represents a neutral or undecided position. Another interesting point to highlight is that these most recent results evaluate better most of the issues that were investigated in our previous study [FBV17].

We have also made open questions to the participants about OntoJaCaMo. One of these questions was: “In your opinion, what advantages can be gained in the current implementation of the OntoJaCaMo plug-in?”. The following **advantages** were enumerated:

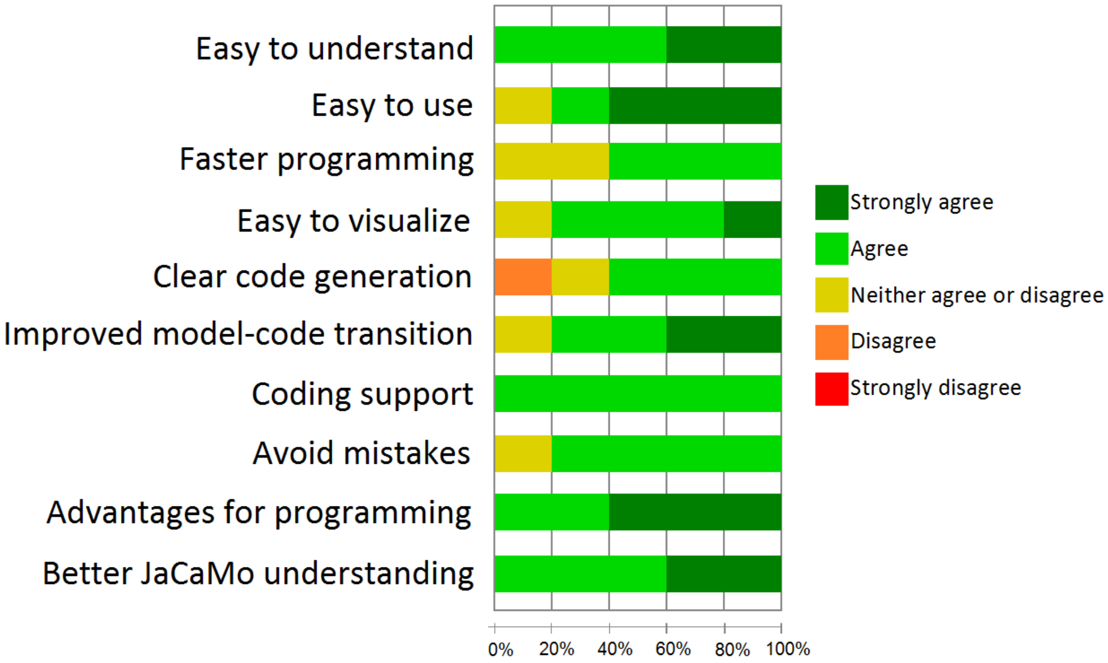


Figure 7.1 – Participant opinions on their use of OntoJaCaMo plug-in.

- OntoJaCaMo facilitates the implementation in JaCaMo, mainly for beginners or for those who are not fully aware about how to implement some concepts.

- The plug-in improves the understanding about the operation (behaviour) of JaCaMo, such as how the programming occurs.

- OntoJaCaMo helps with syntax issues since it brings code templates (it is important because the traditional "ctrl + space" shortcut does not work in all JaCaMo extensions).

- More agility can be obtained from generating JaCaMo source code.

- During development, it is interesting to visualise the system's ontology, so that the idea defined in the ontology may be followed easier when programming.

We have also asked to each participant what **disadvantages** were detected in the current implementation of the OntoJaCaMo plug-in. The following points were enumerated:

- One participant pointed out that some options should not be suggested given a current context. For example, if you have the .asl file open, dragging CArtAgO or Moise should not be available. However, with prior knowledge on the use and context of each feature, this would certainly not be a problem.

- There is no possibility of changing concepts; *i.e.*, in the course of the implementation, new concepts have been added/changed.

- Need to always select the ontology at each time that the eclipse is opened.

- The lack of code generation when dragging some elements.

We highlight some alternatives to address these disadvantages pointed out by the participants. It is important to observe that not all elements generate some code to every platform of JaCaMo, as can be observed from our explanation on the proposed programming technique. The suggestion of updating OntoMAS models inside the IDE context of Eclipse is indeed interesting (instead of using an ontology editor as we have currently investigated in this research). Some shortcuts such as to save the address of a working ontology in a worskpace may be implemented in future releases. Also, more intelligent mechanisms of context awareness regarding the programming location in which the drag-and-drop is being applied may be considered as well.

The third open question made to the participants was the following: “Is there anything else that could be improved or changed in OntoJaCaMo to enhance the features that it offers? If so, what?”.

- The option to refresh the ontology (without needing to reload the IDE), or even to be able to edit the ontology in the eclipse plug-in itself.
- It was suggested to highlight which of the ontology elements shown in OntoJaCaMo are valid alternatives for drag-and-drop in the current JaCaMo file being edited (asl, java, XML, or jcm file). Thus, pointing out the valid options as suggestions, and omitting the alternatives not applicable in that context;
- Changing the destination of the code generation (Jason, CArtAgO, Moise) could be more intuitive. It was suggested that the target of the drag-and-drop could be inferred automatically based on the context of the files opened in the IDE.
- Correction of the code generations that in some cases were presenting minor bugs.

All of the suggestions for improvements previously mentioned and the feedback obtained directly from their target users is very important and should orientate the next releases of OntoJaCaMo towards its community. Our final request was if the participants would you like to leave any final comment, question, or suggestion. In this space, one participant pointed out that the first impression in using the plug-in is that it really helps in understanding the technologies inside JaCaMo: Jason, CArtAgO, and Moise. Other participant have highlighted that it was quite creative to implement such functionalities using a plug-in format in an already consolitadate IDE. According to this participant, this greatly facilitates the development and adds value to the product.

Added to the results obtained in this last round of evaluation and proof of concept, it is important to mention the considerations that we have highlighted based on our initial round of experiments [FBV17]. In that occasion, it was observed that such plug-in helps in code consistency (e.g., it facilitates coding using the same terms), and it prevents developers from using terms outside the ontology-based model. In summary, it was observed that the proposed approach provides an overview about agent systems to be visualised inside the programming context, combined with features of dragging content from models to MAS code.

It was also identified in that time that more code of MAS could be generated from the proposed modelling approach, and that the ontology could be used in a technique to constrain the MAS coding (i.e., indicate errors or mismatches between model and code). We also have indicated that OntoJaCaMo could allow to edit the model (for example, to include new instances), which would discard the need of using an ontology editing tool to update the OntoMAS model. Another point that was highlighted, although a very complex one, is the automatic update of the ontology when the MAS code changes [FBV17], in the direction of synchronising model and code. Currently, programmers have to manually change the ontology to reflect changes in the code and such maintenance can be very laborious. This might be solved by implementing features to highlight mismatches between MAS code and its corresponding model in order to keep both aligned (in other words, refactoring mechanisms for model and code synchronising).

7.2 Evaluating the Generation of an Initial JaCaMo Project from OntoMAS Models

In order to evaluate the use of our technique for generating JaCaMo code from instantiated ontologies, we compare the code that we can created automatically from the OntoMAS models with the code actually programmed by the participants. Through these comparisons, we want to demonstrate the correspondences and similarities between elements in the code that was automatically generated from the specification in contrast with the code that was manually programmed. These similarities between these two sources of code indicative the correctness of the proposed model-based code generation technique.

For example, one participant have worked with a so called “Rescue Scenario”, in which the MAS simulates agents rescuing injured victims. We highlight in Table 7.1 some key elements in the OntoMAS model created by the participant, the corresponding code that can be automatically generated from these elements by using the proposed techniques, and the code actually programmed by the participant. We want to emphasise that the model-based technique for generating code is indeed offering a program equivalent with the code created by a programmer[1], given the analysed aspects.

From analysing the elements in Table 7.1, we observe that there are strong similarities when comparing the code generated from the model with the code actually programmed. The only aspect of divergence observed was the amount (and the distribution) of the agents and of the artifacts, represented by their instances. This is a minor change in the code,

---

[1] The model is in a higher abstraction level than the code, i.e., sometimes only a structure or skeleton of code may be created and programmers have to complete to obtain a fully executable and running system.

and it have occurred probably because the developer decided to adjust his/her simulation by including more agents of a given type, less of some others, and so on. We also have noted, in this scenario, that the characteristics about the environment were mostly maintained as well, however the terminology changed a little bit but the semantics and meanings were maintained. For example, by transitioning from modelling to coding, the same set of 4 types of artifacts appear in both cases, however Map changed its name to $RescuePlanet$ and $Tent$ to $Agenda$. This, however is a minor changed in nomenclature, and observing the code that can be model-based generated and the code actually programmed we concluded that they are strongly aligned.

Table 7.1 – Similarities between code generation from model and the code actually programmed by the participant for the Rescue Scenario.

OntoMAS elements	**JaCaMo target of code generation**	**Code that can be auto generated**	**Programmed by the participant**
2 subclasses of Agent: Rescuer, Victim	Jason (see Table 6.5)	2 .asl files, one for each type of agent	Yes
6 instances of Agent: 4 Rescuers, 2 Victims	JCM file (see Table 6.8)	Declaration of the corresponding agents	2 Rescuers and 10 Victims in the coded .jcm file
4 subclass of Artifact: Clock, Report, Map, Tent	CArtAgO (see Table 6.6)	4 .java files, one for each type of artifact	Yes
8 instances of Artifact: 2 Clocks, 1 Report, 4 Tents, 1 Map	JCM file (see Table 6.8)	Declaration of the corresponding artifacts in the .jcm file	1 Clock,1 Report, 3 Agendas and 16 RescuePlanets in the coded .jcm file
2 subclass of Group: MedicalUnit, RescueUnit	Moise (see Table 6.7)	Declaration of each type of group in the .xml file	Yes
2 instances of Group: 1 MedicalUnit, 1 RescueUnit	JCM file (see Table 6.8)	Declaration of all groups in the .jcm file	Yes
4 instances of Role: Doctor, Rescuer, Scout, FireFighter	Moise (see Table 6.7)	Declaration of each role in the .xml file	Yes

We argue that if the starting codes were created based on converting their corresponding models, then it would be easier for programmers to align their initial code with the design and continue their programming based on that. The similarities between what can be automatically generated with what was manually created indicate that the technique of generating code is aligned in the correct direction. Also, it provides more agility for developers that have their systems modelled.

In the experiment, conducted in the context of a graduate course, we have observed that the code was more complete that the model, but that was probably due to the fact that they had to handle the code at the final stage of the course. In other words, sometimes it is the case that new elements are included in the code of the system without being mentioned in its corresponding model. For example, in the Rescue Scenario highlighted in Table 7.1, the model defines 2 types of agents, $Rescuers$ and $Victims$, which also appear in the final code, however, 3 additional types of agents were included during the programming: $Scouts$, $Doctors$, and $FireFighters$. In this sense, the presented results lead to conclude that it would be interesting to have such proposed model-based code generation. This is particularly important if the model is given as a specification, then something that has to be added in the code, that was not in the original model, should appear also in the model, and as such communicated back to the model designer (in case of different persons working on modelling and programming).

This Section have described our evaluations on the practical use of the programming techniques of MAS based on OntoMAS models as implemented in OntoJaCaMo. Next Sections presents our final remarks about the research in this thesis, which contemplates a discussion about its possible limitations, stances of MAS not addressed by OntoMAS models, and new research directions for future work.

8. FINAL REMARKS

> *“Science may set limits to knowledge, but*
> *should not set limits to imagination."*
> *Bertrand Russell* — (1872 - 1970)

Although the advantages of ontologies for MAS have been considered in many ways, few agent-based systems development platforms currently integrate ontology techniques [FBV17]. The use of ontologies for MAS modelling and development is emerging, however, ontologies for MAS only cover parts of the whole picture, such as the dimension of environment or organisation. On the other hand, there are models and MDE approaches addressing the overall MAS, but without using ontologies, semantic reasoning, or employing the models during the programming step. This context have led this research to investigate, to propose, and to evaluate the integration of MDE, MAS, and ontologies. Thus, its fundamental idea is the use of ontologies in the model-based design and programming of MAS.

This research claims that the proposed MAS modelling and development approaches (i.e., OntoMAS and OntoJaCaMo) increase the flexibility and ease the engineering of agent systems. First, the MAS starts to be modelled in a single formalism and the ontology allows to connect and reuse knowledge of one dimension into others, improving the interoperability of agent platforms. For example, the characteristics of one dimension (e.g., environment) can be used to define properties on another (e.g., organisational). Our MDE approach also enables the conversion of MAS defined in ontologies to programming code in specific agent platforms while remaining flexible enough to accommodate the needs of MAS modellers.

OntoMAS integrates the dimensions of MAS at the semantic level, since they are already being integrated in the programming level, for example, in JaCaMo [BBH+13]. Agent programmers benefit from an integration among these ontological levels with each programming dimension since the knowledge represented in one dimension can be reused in another, thus resulting in a greater interoperability of agent platforms. This enables to convert MAS defined in ontologies to code in specific agent platforms, as done with OntoJaCaMo. Also, a system designed with a higher degree of modularity is easier to maintain, given that it separates different concerns yet enables relations between them. In fact, it is often the case that the concepts of one level are related to another but current MAS platforms do not allow for such relations to be explicitly represented [FBV17].

In terms of MAS design, an ontology model provides a global conceptual view which in combination with MDE can result in tools, for example, to verify model consistency, perform inferences with semantic reasoners, query instantiated models, develop/visualise MAS specifications in ontologies, support programming, and so on. As result, developers obtain new features for developing complex software systems with an infrastructure that combines and applies modelling, software, and knowledge engineering principles. For example, MDE can obtain unambiguous definitions from meta-models formally defined in ontology languages, and reasoners can validate meta-models automatically or generate MAS code from models, all of which contribute to more principled ways to develop MAS.

As result of our work in the areas of MDE, ontology, and MAS, the practical evaluations of this research provides sufficient evidences to indicate that the use of ontology facilitates the modelling of MAS, supports agent programming, and provides a basis for reasoning about the modelled system. Our experiments help to highlight advantages as well as limitations and possibilities for improvements in the current state of the proposed techniques. Future work might make more comparisons about the processes of modelling and programming with standard approaches versus OntoMAS and OntoJaCaMo, as well as different analysis of the resulting models and codes. However, the evaluations reported in Sections 5 and 7 provide sufficient evidence (although in a simulated laboratory environment) to suggest that our techniques are feasible and correct for providing useful support for modelling and coding agent-based software systems.

This thesis has a great focus on JaCaMo as the main target programming platform for the techniques that it proposes and investigates for covering the code generation and the modelling capabilities. Limitations in JaCaMo may impact our proposals, as well as whether new resources appear in JaCaMO, changes may be required in our proposal to keep up with and reflect such new ideas. JaCaMo is claimed by their authors to be the first successful combination of AOP, OOP, and EOP in a specific programming platform [BBH$^+$13]. They recognise that each dimension had been independently developed sufficiently to be put together in a single platform, however, many open problems in each of these dimensions are also identified [BBH$^+$13]. For example, in AOP, modularity [vRDMdB06] and debugging [PPW03] are two aspects that still require significant research to ensure practitioners to have the best possible techniques. Similarly, in OOP there are many complex theories regarding both agent organisations and normative systems [BvdTV08] that are discussed in the literature which it is not yet implemented in practical platforms. The author of JaCaMo [BBH$^+$13] expect that advances in those separate areas of research will be integrated into JaCaMo as soon as they are made sufficiently practical in computational terms. Therefore, when new techniques become available in programming platforms, the modelling approaches should evolve together as well to follow such advances.

One could also consider the possibility of applying UML as a substitute to OWL for describing a given content or domain of knowledge. Some authors argue that UML, in its original form, provides insufficient support for modelling MAS [dSdL03]. However, it could be the case that some new extension of UML would enable a suitable application of this paradigm for agents' development. It would be also the case that one of these alternative choices brings different sets of tools to work with, with more advantages in some aspects, however there would also be drawbacks in other parts. Our work points out an investigation that considers the use of ontology as an alternative, but we recognise that studies comparing different paradigms would be interesting for the communities working on both of them, specially when considering the application's context of this work, which is to support AOSE.

8.1 Stances of MAS not Addressed in OntoMAS

There are characteristics of MAS that only begin to exist when a JaCaMo project is running. These characteristics simply are not tangible while the project is being programmed (for example, a reference to the execution of a plan or to the execution of a action – since the code deals with descriptions of these ideas). Thus, the execution of a plan only takes place when a JaCaMo system is running. In order to represent this views in OntoMAS, new elements should be included, for example a new concept to represent $PlanExecution$ together with a property to specify which is its plan specification: $PlanExecution$ $is\text{-}execution\text{-}of$ $Plan$. The same applies to actions and their executions, the existence of goals and beliefs that appear only after that the system is initialised, the real exchange of messages that occurred between individual agents, and so on. Similarly, there are characteristics about the environment or about the organisation dimension that transcend the initial specification of MAS, thus fitting into this category of being tangible only when a JaCaMo system is running. Extensions and future work on OntoMAS may address such run-time issues of MAS.

This work emphasises an ontology for modelling agents, the environment in which they operate, and the organisation to support the coordination of autonomous agents. A perspective that is receiving increased attention by researchers in MAS is the idea of *interaction*. Many researchers in MAS believe that interactions among the agents are crucial issues to be considered when developing such systems, thus, methodologies to support interactions explicitly are desirable [FBV17]. Interactions include communications, intentions, obligations, and commitments. The authors of JaCaMo, in [BBH$^+$13], considered a future work the investigation of *interaction* as a main dimension to be integrated synergistically with the other ones. Only in recent work [ZRH16] that JaCaMo is starting to be extended in order to provide such features, including a new *interaction* dimension. Considering such context, the modelling of interactions to extend our ontology would be an interesting future work to explore. This has not yet been fully addressed so far given how recent such developments are in the current available programming abstractions provided by the IDEs for JaCaMo.

Another possibility to deal with the interaction concepts would be to incorporate them into the organisation dimension of MAS. In this way, the organisation dimension would address ontological support for interactions among the agents, as first-class objects, by means of protocols and commitments. Existing ontologies for commitments in MAS provide a conceptual point of view for comprehending advanced organisational details in the form of interactions and commitments. One of these ontologies [Sin99] defines a *commitment* as involving a proposition with three participating agents: a debtor, a creditor, and a context group. Also, different kinds of commitments are defined, such as obligation, taboo, convention, and pledge. It is important to be aware of these high-level conceptual viewpoints when aiming to integrate such ideas from theory to practice both at the modelling and at the programming levels.

Currently, Moise implements a limited view of the many advanced theoretical ideas and possibilities regarding MAS organisations, and so does OntoMAS since they are both aligned. However, future work could expand and enrich both the modelling and the programming capabilities of these techniques in order to better represent these rich and complex details regarding social aspects in the development of agent systems. At the programming level, in JaCaMo, an approach to code “commitment patterns” could be proposed, as well as new native constructs directly at the development platform. At the modelling level, in OntoMAS, new concepts to represent commitments, debtors, creditors, and contexts of commitments could be created [FBV17]. In its current version, OntoMAS addresses models of agent systems that are aligned with JaCaMo and contain, for example, the initial and static representations of organisations for Moise which are applied to generate code for agent platforms. It is currently future work to extend the modelling approach to address run-time characteristics of MAS, such as to represent and monitor organisational properties for when the modelled system is in execution. This includes, for example, Moise’s notion of an agent committing to achieve certain goals, a simpler notion than commitments as described above.

8.2 Future Work and New Research Directions

We end this thesis in this subsection that discusses future work and new research directions. To not sound repetitive, ideas recently mentioned, such as expanding OntoMAS with concepts of interactions discussed in the previous subsection, are not repeated here.

This research opens possibilities of applying the proposed ontology in many other ways. In terms of MAS design, such ontology model provides a global conceptual view which in combination with MDE can result in techniques, for example, to verify model consistency, perform inferences with semantic reasoners, query instantiated models, develop/visualise MAS specifications in ontologies, support programming, and so on. As result, developers obtain new features for developing complex software systems with an infrastructure that combines and applies modelling, software and knowledge engineering principles. For example, MDE can obtain unambiguous definitions from meta-models formally defined in ontology languages, and reasoners can validate meta-models automatically or generate MAS code from models, all of which contribute to more principled ways to develop MAS [FBV17]. As future work, it can be investigated other possibilities of applications and techniques that might be derived from this research.

We believe that producing software code for complex and highly detailed systems directly in programming environments without first using any specification, modelling, or design mechanism may cause many problems [FCVB16]. Without a proper modelling of the system it can be difficult to find potential bugs when they eventually appear in the implementation. Advantages derived from such approach are techniques for: *(i)* integrating design and code; *(ii)* supporting MAS programming with automatic code generation through model-based development; and *(iii)* performing verification with focus on the use of semantic reasoning and model checking [FBV17].

In order to facilitate the specifications of a MAS in OntoMAS, a graphical user notation could be designed for it. Such notation should define one graphical identification for each element from the ontology. Then, an environment may be configured so that an interface provides an oriented process based on the proposed guidelines. These would replace the need of working with an ontology editor, such as Protégé, which was employed during this research. The ideal would be that the modelling of a project taking place in the same IDE used for its programming, which in case of JaCaMo is Eclipse. Also, techniques and tools using OntoMAS could be investigated and reused for other agent programming frameworks in a similar way that OntoJaCaMo was investigated for JaCaMo.

One of the main contributions of this thesis is an ontology of agent systems – OntoMAS – that considers the dimensions of agents, environments, and organisations. This ontology could be handled by agents using the artifact that we have implemented in other research of ours [FPH^{+}15, FPH^{+}17]. Thus, agents could reason about other systems, or even about themselves. These would allow agents to be able to share their implementation with others, or to execute inferences about its own implementation.

8.3 Publications in the Main Theme of this Thesis

Here we briefly explain 5 of the papers produced during this thesis that are directly related with its main research topic, in a chronological order:

- [FSP^{+}14] 2014. **Semantic Representations of Agent Plans and Planning Problem Domains**. International Workshop on Engineering Multi-Agent Systems. Artur Freitas, Daniela Schmidt, Alison R. Panisson, Felipe Meneguzzi, Renata Vieira, Rafael H. Bordini.

  In this paper we explore the use of ontology as semantic representations of agent plans, which is an important characteristic of the agent dimension.

- [FBMV15] 2015. **Towards Integrating Ontologies in Multi-Agent Programming Platforms**. International Conference on Web Intelligence and Intelligent Agent Technology. Artur Freitas, Rafael H. Bordini, Felipe Meneguzzi, Renata Vieira.

  This paper was our first presentation on the idea of using ontologies to integrate elements from the different dimensions of MAS (agent, environment, and organisation).

- [FSP+15] 2015. **Applying Ontologies and Agent Technologies to Generate Ambient Intelligence Applications**. Joint Proceedings Collaborative Agents – Research & Development, CARE for Intelligent Mobile Services & Agents, Virtual Societies and Analytics. Artur Freitas, Daniela Schmidt, Alison R. Panisson, Felipe Meneguzzi, Renata Vieira, Rafael H. Bordini.

  This article continues the investigation on our initial ideas and it is the first time that we have proposed the mapping of elements from an ontology model to code into the three dimensions of MAS.

- [FHM+15] 2015. **A Multi-Agent Systems Engineering Tool based on Ontologies**. International Conference on Conceptual Modeling. Artur Freitas, Lucas Hilgert, Sabrina Marczak, Felipe Meneguzzi, Rafael H. Bordini, Renata Vieira.

  This demonstration paper introduces the practical aspects of our model and ontology-based code transformation techniques for JaCaMo, and their implementations in the format of a tool as a plug-in for Eclipse to support the development of JaCaMo projects.

- [FBV17] 2017. **Model-Driven Engineering of Multi-Agent Systems based on Ontologies**. Applied Ontology Journal. Artur Freitas, Rafael H. Bordini, Renata Vieira.

  In this journal article we provide a more comprehensively explanation on the details of the idea investigated in this thesis: techniques for modelling and programming MAS that are formalised in the OntoMAS ontology and in the OntoJaCaMo tool. This thesis, however, provides more details on these techniques, more results of evaluations, a larger review of literature, more discussions about the investigated approaches, etc.

8.4 Publications in Supplementary Areas of this Thesis

Some papers produced in complementary areas to the main topic of this thesis are also important to be mentioned, as briefly explained here:

- [PFF$^+$14] 2014. **Planning Interactions for Agents in Argumentation-Based Negotiation**. International Workshop on Argumentation in Multiagent Systems. Alison R. Panisson, Giovani Farias, Artur Freitas, Felipe Meneguzzi, Renata Vieira, Rafael H. Bordini.

  In this research it is addressed an approach for planning the interactions of agents in argumentation-based negotiations so that agents may form their strategy of arguments more intelligently.

- [PFS$^+$15] 2015. **Arguing About Task Reallocation Using Ontological Information in Multi-Agent Systems**. International Workshop on Argumentation in Multiagent Systems. Alison R. Panisson, Artur Freitas, Daniela Schmidt, Lucas Hilgert, Felipe Meneguzzi, Renata Vieira, Rafael H. Bordini.

  In this paper we investigate the use of ontological information in agent communication, so that the agents can argue about task reallocations on the basis of an ontology.

- [FPH$^+$15] 2015. **Integrating Ontologies with Multi-Agent Systems through CArtAgO Artifacts**. International Conference on Intelligent Agent Technology. Artur Freitas, Alison R. Panisson, Lucas Hilgert, Felipe Meneguzzi, Renata Vieira, Rafael H. Bordini.

  This paper proposes and evaluates an implemented mechanism for agents to interact with ontologies that may be used in any MAS platform that supports CArtAgO.

- [SPF$^+$16] - 2016. **An Ontology-based Mobile Application for Task Managing in Collaborative Groups**. International Florida Artificial Intelligence Research Society (FLAIRS) Conference. Daniela Schmidt, Alison R. Panisson, Artur Freitas, Rafael H. Bordini, Felipe Meneguzzi, Renata Vieira.

  In this paper, we present an ontology for task representation and its use in the context of collaborative groups implemented as MAS.

- [FCVB16] 2016. **Limitations and Divergences in Approaches for Agent-Oriented Modelling and Programming**. International Workshop on Engineering Multi-Agent Systems. Artur Freitas, Rafael C. Cardoso, Renata Vieira, Rafael H. Bordini.

  This article assesses which are the implications on the combined use of Prometheus for modelling and JaCaMo for programming using as case study the complex MAS scenario of the MAPC in 2016. Although the solution developed at PUCRS have won the MAPC[1] of 2016 [CPK$^+$17], we have noticed the need of a modelling approach more aligned with JaCaMo, as the one investigated in this thesis.

---

[1] MAPC (Multi-Agent Programming Contest) is an annual competition carried out as an attempt to stimulate research in the area of MAS programming. Since when it was released, JaCaMo was used by different winning teams that have competed against several other possible platforms, such as JACK, GOAL, JIAC, Python, Java,

- [FPH$^+$17] 2017. **Applying Ontologies in the Development and Execution of Multi-Agent Systems**. Web Intelligence Journal. Artur Freitas, Alison R. Panisson, Lucas Hilgert, Felipe Meneguzzi, Renata Vieira, Rafael H. Bordini.

  This journal paper expands the previous research [FPH$^+$15] of enabling the use of ontologies in the development and execution of MAS.

C++, etc. Teams from the Federal University of Santa Catarina (UFSC, Brazil) have won in 2012 using Jason (JaCaMo [BBH$^+$13] was not published yet), and in 2013 and 2014 using JaCaMo. The scenario in all these years was called “Agents on Mars”. Then, due to the complexity of developing the new scenario “Agents in the City”, the contest was pushed to 2016. In 2016 a team from the Pontifical Catholic University of Rio Grande do Sul (PUCRS, Brazil), which was participating for its first time, have won using JaCaMo as well (teams from UFSC did not participated this year). More information about the MAPC can be found at the official website of the competition: https://multiagentcontest.org/.

REFERENCES

[AGK06] Atkinson, C.; Gutheil, M.; Kiko, K. “On the relationship of ontologies and models.” In: Proceedings of the 2nd Workshop on Meta-Modelling, 2006, pp. 47–60.

[AK03] Atkinson, C.; Kühne, T. “Model-driven development: A metamodeling foundation”, *IEEE Software*, vol. 20–5, Sep 2003, pp. 36–41.

[BBB$^+$97] Bayardo, Jr., R. J.; Bohrer, W.; Brice, R.; Cichocki, A.; Fowler, J.; Helal, A.; Kashyap, V.; Ksiezyk, T.; Martin, G.; Nodine, M.; Rashid, M.; Rusinkiewicz, M.; Shea, R.; Unnikrishnan, C.; Unruh, A.; Woelk, D. “InfoSleuth: Agent-based semantic integration of information in open and dynamic environments”, *ACM SIGMOD International Conference on Management of Data*, vol. 26–2, 1997, pp. 195–206.

[BBH$^+$13] Boissier, O.; Bordini, R. H.; Hübner, J.; Ricci, A.; Santi, A. “Multi-agent oriented programming with JaCaMo”, *Science of Computer Programming*, vol. 78–6, 2013, pp. 747–761.

[BCG07] Bellifemine, F. L.; Caire, G.; Greenwood, D. “Developing Multi-Agent Systems with JADE”. John Wiley & Sons, 2007.

[BDW06] Bordini, R. H.; Dastani, M.; Winikoff, M. “Current issues in multi-agent systems development.” In: Engineering Societies in the Agents World, O’Hare, G. M. P.; Ricci, A.; O’Grady, M. J.; Dikenelli, O. (Editors), 2006, pp. 38–61.

[Béz06] Bézivin, J. “Model driven engineering: An emerging technical space”. In: *Generative and transformational techniques in software engineering*, Springer, 2006, pp. 36–64.

[BHS04] Baader, F.; Horrocks, I.; Sattler, U. “Description logics”. In: *Handbook on Ontologies*, 2004, pp. 3–28.

[BHW07] Bordini, R. H.; Hübner, J. F.; Wooldridge, M. “Programming multi-agent systems in AgentSpeak using Jason”. John Wiley & Sons, 2007.

[BS08] Bromuri, S.; Stathis, K. “Situating cognitive agents in GOLEM”. In: International Workshop on Engineering Environment-Mediated Multi-Agent Systems (EEMMAS), Weyns, D.; Brueckner, S. A.; Demazeau, Y. (Editors), 2008, pp. 115–134.

[Bud04] Budinsky, F. “Eclipse Modeling Framework: A Developers Guide”. Addison-Wesley, 2004.

[BvdTV08] Boella, G.; van der Torre, L.; Verhagen, H. “Introduction to the special issue on normative multiagent systems”, *Autonomous Agents and Multi-Agent Systems*, vol. 17–1, 2008, pp. 1–10.

[BvHH+04] Bechhofer, S.; van Harmelen, F.; Hendler, J.; Horrocks, I.; McGuinness, D. L.; Patel-Schneider, P. F.; Stein, L. A. “OWL Web Ontology Language Reference”, Technical Report, W3C, 2004.

[Car02] Carson, J. S. “Model verification and validation”. In: Proceedings of the 2002 Winter Simulation Conference, 2002, pp. 52–58.

[CPK+17] Cardoso, R. C.; Pereira, R. F.; Krzisch, G.; Magnaguagno, M. C.; Baségio, T.; Meneguzzi, F. “Team PUCRS: a decentralised multi-agent solution for the agents in the city scenario”, *International Journal of Agent-Oriented Software Engineering (IJAOSE)*, 2017.

[DGMT09] Dastani, M.; Grossi, D.; Meyer, J.-J. C.; Tinnemeier, N. “Normative multi-agent programs and their logics”. In: 1st International Workshop on Knowledge Representation for Agents and Multi-Agent Systems (KRAMAS), Meyer, J.-J. C.; Broersen, J. (Editors), 2009, pp. 16–31.

[dSdL03] da Silva, V. T.; de Lucena, C. J. “MAS-ML: a multi-agent system modeling language”. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 2003, pp. 304–305.

[FBMV15] Freitas, A.; Bordini, R. H.; Meneguzzi, F.; Vieira, R. “Towards integrating ontologies in multi-agent programming platforms”. In: Proceeding of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 2015, pp. 225–226.

[FBV17] Freitas, A.; Bordini, R. H.; Vieira, R. “Model-driven engineering of multi-agent systems based on ontologies”, *Applied Ontology Journal*, vol. 12, 2017, pp. 157–188.

[FCVB16] Freitas, A.; Cardoso, R. C.; Vieira, R.; Bordini, R. H. “Limitations and divergences in approaches for agent-oriented modelling and programming”. In: International Workshop on Engineering Multi-Agent Systems, Baldoni, M.; Müller, J. P.; Nunes, I.; Zalila-Wenkstern, R. (Editors), 2016, pp. 88–103.

[FHM+15] Freitas, A.; Hilgert, L.; Marczak, S.; Meneguzzi, F.; Bordini, R. H.; Vieira, R. “A multi-agent systems engineering tool based on ontologies”. In: 34th International Conference on Conceptual Modeling, 2015.

[FPH+15] Freitas, A.; Panisson, A. R.; Hilgert, L.; Meneguzzi, F.; Vieira, R.; Bordini, R. H. “Integrating ontologies with multi-agent systems through CArtAgO artifacts”. In: Proceeding of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2015, pp. 143–150.

[FPH+17] Freitas, A.; Panisson, A. R.; Hilgert, L.; Meneguzzi, F.; Vieira, R.; Bordini, R. H. “Applying ontologies in the development and execution of multi-agent systems”, *Web Intelligence Journal*, 2017, accepted for publication.

[FSP+14] Freitas, A.; Schmidt, D.; Panisson, A.; Meneguzzi, F.; Vieira, R.; Bordini, R. H. “Semantic representations of agent plans and planning problem domains”. In: International Workshop on Engineering Multi-Agent Systems, Dalpiaz, F.; Dix, J.; van Riemsdijk, M. (Editors), 2014, pp. 351–366.

[FSP+15] Freitas, A.; Schmidt, D.; Panisson, A.; Meneguzzi, F.; Vieira, R.; Bordini, R. H. “Applying ontologies and agent technologies to generate ambient intelligence applications”. In: Joint Proceedings Collaborative Agents – Research & Development, CARE for Intelligent Mobile Services & Agents, Virtual Societies and Analytics, Koch, F.; Meneguzzi, F.; Lakkaraju, K.; Ahmad, M.; Sukthankar, G. (Editors), 2015, pp. 22–33.

[GH01] Grundy, J.; Hosking, J. “Software tools”, *Encyclopedia of Software Engineering*, 2001.

[GNFC12] Gascueña, J. M.; Navarro, E.; Fernández-Caballero, A. “Model-driven engineering techniques for the development of multi-agent systems”, *Engineering Applications of Artificial Intelligence*, vol. 25–1, 2012, pp. 159–173.

[Gru93] Gruber, T. R. “A translation approach to portable ontology specifications”, *Knowledge Acquisition Journal*, vol. 5–2, 1993, pp. 199–220.

[HB11] Horridge, M.; Bechhofer, S. “The OWL API: A Java API for OWL ontologies”, *Semantic Web Journal*, vol. 2–1, 2011, pp. 11–21.

[HBKR10] Hübner, J. F.; Boissier, O.; Kitio, R.; Ricci, A. “Instrumenting multi-agent organisations with organisational artifacts and agents”, *Autonomous Agents and Multi-Agent Systems*, vol. 20–3, 2010, pp. 369–400.

[HWDC09] Hadzic, M.; Wongthongtham, P.; Dillon, T.; Chang, E. “Ontology-Based Multi-Agent Systems”. Springer, 2009.

[KB08] Klapiscak, T.; Bordini, R. H. “JASDL: a practical programming approach combining agent and semantic web technologies”. In: Proceedings of the 6th

International Workshop on Declarative Agent Languages and Technologies, 2008, pp. 91–110.

[KB15] Kravari, K.; Bassiliades, N. “A survey of agent platforms”, *Journal of Artificial Societies and Social Simulation*, vol. 18–1, 2015, pp. 11.

[KKK$^+$06] Kappel, G.; Kapsammer, E.; Kargl, H.; Kramler, G.; Reiter, T.; Retschitzegger, W.; Schwinger, W.; Wimmer, M. “Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages”, *Model Driven Engineering Languages and Systems*, vol. 4199, 2006, pp. 528–542.

[Lik32] Likert, R. “A technique for the measurement of attitudes.”, *Archives of Psychology Journal*, vol. 22–140, 1932, pp. 1–55.

[MAB$^+$14] Mascardi, V.; Ancona, D.; Barbieri, M.; Bordini, R. H.; Ricci, A. “CooL-AgentSpeak: Endowing AgentSpeak-DL agents with plan exchange and ontology services”, *Web Intelligence and Agent Systems*, vol. 12–1, 2014, pp. 83–107.

[Mus15] Musen, M. A. “The protégé project: A look back and a look forward”, *AI Matters*, vol. 1–4, 2015, pp. 4–12.

[MVBH05] Moreira, A. F.; Vieira, R.; Bordini, R. H.; Hübner, J. F. “Agent-oriented programming with underlying ontological reasoning”. In: Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies, 2005, pp. 155–170.

[OVBdRC06] Okuyama, F. Y.; Vieira, R.; Bordini, R. H.; da Rocha Costa, A. C. “An ontology for defining environments within multi-agent simulations”. In: Workshop on Ontologies and Metamodeling in Software and Data Engineering, 2006.

[PB09] Pokahr, A.; Braubach, L. “A Survey of Agent-oriented Development Tools”. Boston, MA: Springer US, 2009, pp. 289–329.

[PFF$^+$14] Panisson, A. R.; Farias, G.; Freitas, A.; Meneguzzi, F.; Vieira, R.; Bordini, R. H. “Planning Interactions for Agents in Argumentation-Based Negotiation”. In: 11th International Workshop on Argumentation in Multiagent Systems (ArgMAS), 2014.

[PFS$^+$15] Panisson, A. R.; Freitas, A.; Schmidt, D.; Hilgert, L.; Meneguzzi, F.; Vieira, R.; Bordini, R. H. “Arguing about task reallocation using ontological information in multi-agent systems”. In: 12th International Workshop on Argumentation in Multiagent Systems (ArgMAS), 2015.

[PGSF06] Pavón, J.; Gómez-Sanz, J.; Fuentes, R. “Model driven development of multi-agent systems”. In: *Model Driven Architecture – Foundations and Applications*, Rensink, A.; Warmer, J. (Editors), Springer Berlin Heidelberg, 2006, *Lecture Notes in Computer Science*, vol. 4066, pp. 284–298.

[PPW03] Poutakidis, D.; Padgham, L.; Winikoff, M. “An exploration of bugs and debugging in multi-agent systems”. In: ISMIS, Zhong, N.; Ras, Z. W.; Tsumoto, S.; Suzuki, E. (Editors), 2003, pp. 628–632.

[PTW05] Padgham, L.; Thangarajah, J.; Winikoff, M. “Tool support for agent development using the Prometheus methodology”. In: Fifth International Conference on Quality Software, 2005, pp. 383–388.

[PTW08] Padgham, L.; Thangarajah, J.; Winikoff, M. “The prometheus design tool – a conference management system case study”. In: 8th International Workshop on Agent-Oriented Software Engineering (AOSE), Luck, M.; Padgham, L. (Editors), 2008, pp. 197–211.

[PW03] Padgham, L.; Winikoff, M. “Prometheus: A methodology for developing intelligent agents”. In: Agent-Oriented Software Engineering III, Giunchiglia, F.; Odell, J.; WeiB, G. (Editors), 2003, pp. 174–185.

[Roe12] Roebuck, K. “Model-driven architecture (MDA): High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors”. Emereo Publishing, 2012.

[RVO06] Ricci, A.; Viroli, M.; Omicini, A. “CArtAgO: an infrastructure for engineering computational environments in MAS”. In: 3rd International Workshop Environments for Multi-Agent Systems (E4MAS), Weyns, D.; Parunak, H. V. D.; Michel, F. (Editors), 2006, pp. 102–119.

[Sel03] Selic, B. “The pragmatics of model-driven development”, *IEEE Software*, vol. 20–5, 2003, pp. 19–25.

[Sin99] Singh, M. P. “An ontology for commitments in multiagent systems: Toward a unification of normative concepts”, *Artificial Intelligence and Law*, vol. 7–1, 1999, pp. 97–113.

[SPF$^+$16] Schmidt, D.; Panisson, A. R.; Freitas, A.; Bordini, R. H.; Meneguzzi, F.; Vieira, R. “An ontology-based mobile application for task managing in collaborative groups”. In: 29th International Florida Artificial Intelligence Research Society (FLAIRS) Conference, 2016, pp. 522–525.

[SPG$^+$07] Sirin, E.; Parsia, B.; Grau, B. C.; Kalyanpur, A.; Katz, Y. “Pellet: a practical OWL-DL reasoner”, *Journal of Web Semantics*, vol. 5–2, 2007, pp. 51–53.

[SWGP10] Staab, S.; Walter, T.; Groner, G.; Parreiras, F. “Model driven engineering with ontology technologies”. In: *Reasoning Web. Semantic Technologies for Software Engineering*, Abmann, U.; Bartho, A.; Wende, C. (Editors), Springer Berlin / Heidelberg, 2010, *Lecture Notes in Computer Science*, vol. 6325, pp. 62–98.

[TL08] Tran, Q.-N. N.; Low, G. “MOBMAS: A methodology for ontology-based multi-agent systems development”, *Information and Software Technology Journal*, vol. 50–7-8, 2008, pp. 697–722.

[UBSA10] Urovi, V.; Bromuri, S.; Stathis, K.; Artikis, A. “Initial steps towards run-time support for norm-governed systems”. In: International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems (COIN), Vos, M. D.; Fornara, N.; Pitt, J. V.; Vouros, G. A. (Editors), 2010, pp. 268–284.

[UH14] Uez, D. M.; Hübner, J. F. “Environments and organizations in multi-agent systems: From modelling to code”. In: 2nd International Workshop on Engineering Multi-Agent Systems, 2014, pp. 181–203.

[vRDMdB06] van Riemsdijk, M. B.; Dastani, M.; Meyer, J.-J. C.; de Boer, F. S. “Goal-oriented modularity in agent programming.” In: Proceedings of the 5th International Conference on Autonomous Agents and Multiagent Systems, Nakashima, H.; Wellman, M. P.; Weiss, G.; Stone, P. (Editors), 2006, pp. 1271–1278.

[Win05] Winikoff, M. “Jack intelligent agents: An industrial strength platform”. In: *Multi-Agent Programming: Languages, Platforms and Applications*, Bordini, R. H.; Dastani, M.; Dix, J.; El Fallah Seghrouchni, A. (Editors), Springer, 2005, *Multiagent Systems, Artificial Societies, and Simulated Organizations*, vol. 15, chap. 7, pp. 175–193.

[Zar12] Zarafin, A.-M. “Semantic description of multi-agent organizations”, Master’s Thesis, Automatic Control and Computers Faculty, Computer Science and Engineering Department – Politehnica University of Bucharest, 2012.

[ZH14] Zatelli, M. R.; Hübner, J. F. “The interaction as an integration component for the JaCaMo platform”. In: 2nd International Workshop on Engineering Multi-Agent Systems, 2014, pp. 431–450.

[ZRH16] Zatelli, M. R.; Ricci, A.; Hübner, J. F. “Integrating interaction with agents, environment, and organisation in JaCaMo”, *International Journal of Agent-Oriented Software Engineering*, vol. 5–2-3, 2016, pp. 266–302.

APPENDIX A – GUIDELINES FOR PROJECT CONCEPTION USING ONTOMAS

The system has to be specified in three dimensions: Agent, Environment, and Organisation. It is possible to start in any dimension since they will be interconnected in the end. The agent dimension specifies mainly the types of agents, plans, actions, agent goals, beliefs, and messages. The environment dimension specifies mainly the spaces, types of resources in the environment, their operations and observable properties. The organisation dimension specifies mainly groups, roles, missions, organisational goals, and norms to regulate the behaviour of the agents that participate in such organisation.

Taking in account the agent dimension, consider the following:

- Which are the **types of agents** in your system? For each type, create a subclass of Agent. Each subclass of Agent corresponds to a .asl file. Each concrete agent (an instance) should have at least one of these types.

- Restrictions applicable to every agent of a type should be specified as “subclass of restrictions”, such as: (Builder subclass of Agent) and (Builder subclass of has-plan value buildHouse). This example states that every instance of Builder agents has the plan to build a house.

- Conceive **situations** in which the system being designed will be applied. These can be considered as purposes, use cases, functionalities, scenarios, things to be achieved, and so on. This guideline suggests that these things must be represented as goals that agents will pursue in the system. Each goal must be modelled as an instance of the Goal concept.

- Conceive which individual agents you initially need. Create instances for each subclass of Agent that your system requires to accomplish the desired tasks.

- Consider which **types of information** would be required to achieve each goal. Information that agents get from the environment becomes instance of Percept Beliefs; information that agents get from other agents becomes Agent Beliefs; and information that an agent can conclude by itself from internal reasoning.

- Since your agents **act** in an environment to achieve their goals, consider which actions your agents could perform in their environments. Then, create one instance to represent each type of action.

- Agents require a **method** to handle each goal to be achieved and each expected reaction to a belief. These are the plans known by the agents. Instances of plans must be created. These plans must be associated with their triggering conditions, with actions that they may perform, and to which agent that should include such plans.

- Agents can send and receive messages during their existence. Consider which types of messages each agent would need to send or receive. Create one instance to represent each type of messages. Use relations to say that "agent sends message" and "message has receivers".

Taking in account the environment dimension, consider the following:

- Consider the types of **artifacts** in your system. For each type, create a subclass of Artifact. Each subclass of Artifact corresponds to a .java file. Each concrete artifact (an instance) should have at least one of these types.
- Restrictions applicable to every agent of a type should be specified as subclass of restrictions, such as: (Printer subclass of Artifact) and (Printer subclass of has-operation value print). It states that every instance of Printer has the operation to perform print.
- Conceive which **resources** agents can interact with (operate and/or observe), those will become the available artifacts in the environment. Each concrete artifact is modelled as an instance of a subclass of Artifact.
- Conceive which **actions** are available in the environment, those will become the operations of some artifact. Create the operations provided by artifacts as instances of Operation. Connect the instance of an artifact with an instance of operation through the "has operation" property to indicate which operations an artifact has. In code, each operation of an artifact will result in a method in its Java class.
- Conceive which **information** can be obtained from the environment, those will become observable properties of some artifact. Create instances of observable property and connect them with instances of artifacts using the relation named "has property". In code, each observable property is a percept that the artifact should update.
- Conceive if there are special locations in your environment. These are the spaces where your agents and artifacts are situated and can occupy. Agents can only operate and observe artifacts if they share a location. Create instances of Space as desired in order to organise your environment. In the coding platform, spaces represent the idea of workspaces.

Taking in account the organisation dimension, consider the following:

- Consider the **types of groups** in your system. For each type, create a subclass of Group. Each subclass of Groups corresponds to a group (abstractly). Each concrete group (an instance) should have at least one of these types.

- Restrictions applicable to every group of a type should be specified as subclass of restrictions, such as: (FootballTeam subclass of Group) and (FootballTeam subclass of contains role value goalkeeper). It states that every instance of FootballTeam should contain the role goalkeeper.
- Conceive if there are **tasks** that require the collaboration or coordination for their executions. These tasks are referred as missions and are composed of goals. Create the types of missions as instances.
- Organisational **goals** are part of missions. Consider which individual agent goals are part of missions. Create the goals as instances.
- Consider that your agents could adopts some roles. Roles are used to form groups, receive missions, and be regulated by norms. Hierarchy of roles is allowed (a role specialises other role, and thus inherit its characteristics). Create the roles as instances.
- Your agents that adopt roles can be organised in concrete **groups**. You have to create the groups as instances of the subclasses of Groups.
- Your organisation can be regulated by **norms**. Consider if there are some obligation, prohibition or permission for each role in the organisation. Create the norms as instances of some of the subclasses of Norm.

Taking in account the integration among agents and environments, consider the following:

- Consider that your agents can interact with some artifacts. To do so, they must be focused on them, so they are allowed to execute operations and perceive observable properties. Use this property to relate that: Agent is-focused Artifact.
- Consider that your agents and artifacts are situated in spaces. Normally, for each of these elements should be set to where they are initially located. To do so, use the property: Agent is-in Space.

Taking in account the integration among agents and organisations, consider the following:

- You can have individual agents adopting roles. For each instance of (subclasses of) Agent consider if your want them to adopt one or more of the available roles. This can be done using the property: Agent adopts-role Role.
- Groups are composed of your individual agents. For each instance of (a subclasses of) Group specify its members. To do so, use the property: Group contains-agent Agent.

APPENDIX B – HELLO WORLD SPECIFICATION AS USED IN THE EXPERIMENTS

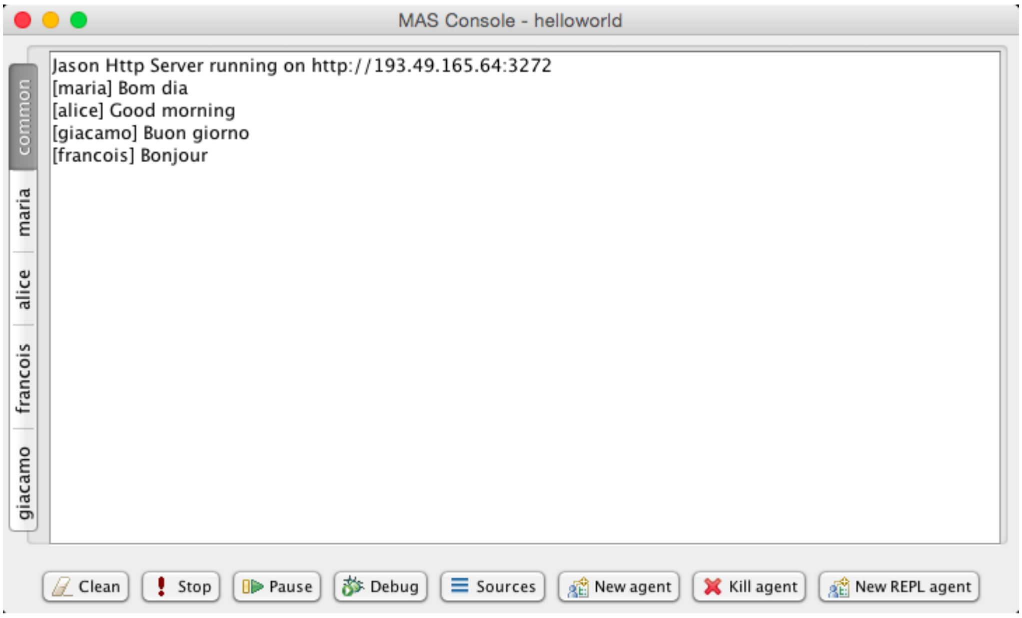
This specification was used as our learning scenario and delivered to the participants as follows. Your task is to use a modelling approach to initiate the design of a Multi-Agent System (MAS). Consider that the JaCaMo programming platform is going to be the coding framework for your solution. You have two alternatives for modelling your MAS: OntoMAS or Prometheus.

The specification of the MAS is as follows. The system should have only one unique type of agent. From this requirement, the initial application will have four agents that will print different hello world messages. The source code for all of them will be the same (a Jason .asl file), but they will have different names and different initial beliefs. The four agents (named francois, maria, giacomo, and alice) share the same program file hello.asl, and each agent will have an initial belief message(“xxxxx”) corresponding to the message that it should use to say hello. The agent francois believes in message(“Bonjour”); the agent maria believes in message(“Bom dia”); the agent giacomo believes in message(“Buon giorno”); and the agent alice believes in message(“Good morning”).

The hello.asl agents have a plan that can be read by the agent as “whenever I have the goal !start and I believe in message(X), I will achieve this goal by doing .print(X). X is a variable that gets value by matching message(X) with some agent’s belief. If the agent belief is message(“Bom dia”), the value of X will be “Bom dia”. If this plan cannot be used (because the agent does not believe in message(X)), then other plan is used and it executes the command .print(?hello world!?). The expected result for your system, so far, is the following:

The environment of this application is quite simple, it has a graphical display artifact where agents can print messages and perceive the number of already printed messages. The artifact has thus one observable property (numMsg) and one operation (printMsg(String)). Initially all agents will share the same display artifact and latter we will have displays in several countries.

The type of this artifact is referred as GUIConsole. Your project will have to create an instance of the display artifact and named it gui. This artifact will be placed in a workspace identified by jacamo. In order to perceive this artifact, the agents need to focus on it. This is why we add for each agent a focus instruction focusing on the artifact gui in the workspace jacamo. In the plan that performs print in the agent source code (file hello.asl), include the printMsg operation which is provided by the gui artifact. When the agent uses the artifact operation, its message is printed in the gui artifact, and when the agent uses the .print action, the message is displayed in the Jason MAS Console.



```
Jason Http Server running on http://193.49.165.64:3272
[maria] Bom dia
[alice] Good morning
[giacamo] Buon giorno
[francois] Bonjour
```

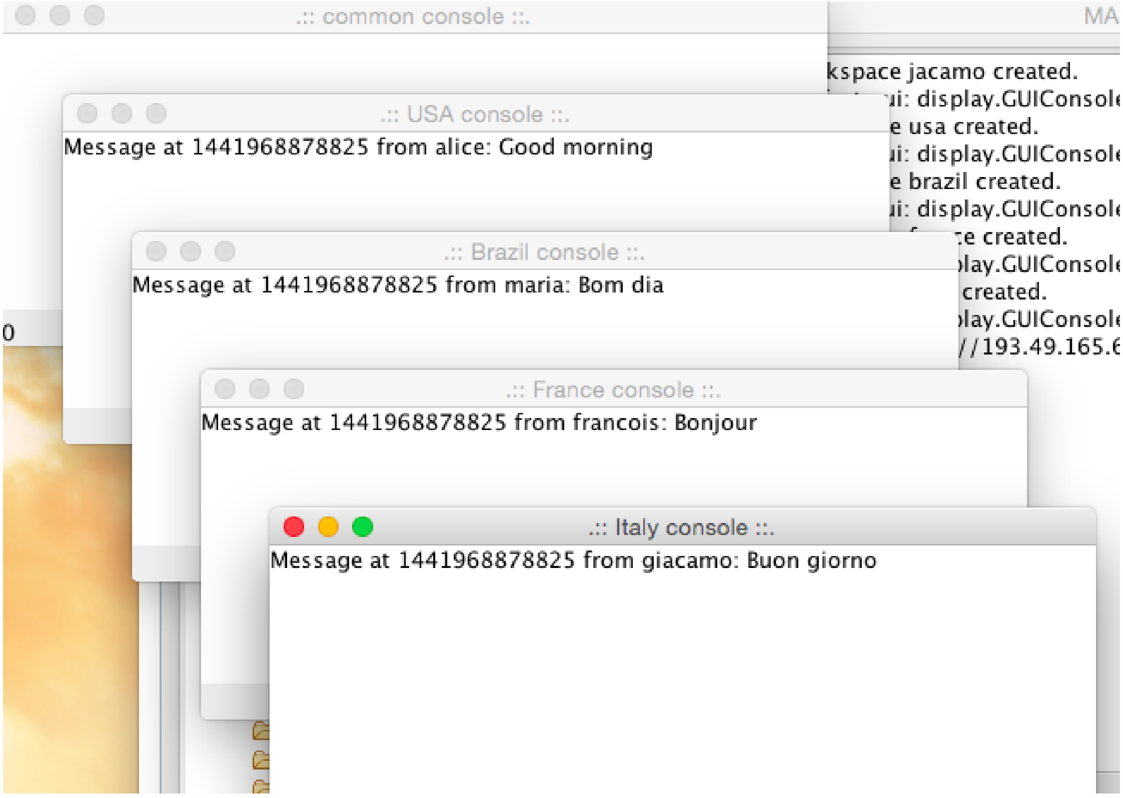
Figure B.1 – Console of Hello World implemented until this point.

Instead of having a shared display artifact, we will now create one display artifact for each country. Since artifacts are inside workspaces, we will also create a workspace for each country (france, italy, brazil, and usa). In order to perceive the artifacts (by focusing on them), the agents should be placed in their proper workspaces: francois in france, maria in brazil, giacomo in italy, and alice in usa. Each agent is focusing on the artifact of its workspace. The expected result for your system, so far, is the following:

We will change our example so that the printing of "Hello World" will be a coordinated task for our four agents: each agent will print one character of the message. For instance, francois will print the “H”, maria the “e”, giacomo the “l”, and so on. Notice that it is very important that they coordinate for the task, for instance, maria should print the "e" only after francois has printed the "H". One way to coordinate the execution of joint tasks is by mean of an organisation. In JaCaMo the organisation is programmed based on the Moise model, where groups, roles, missions, goals, global plans, and schemes are defined. Our organisation has one global goal print_hello that is decomposed into several sub-goals, one for each letter. The sub-goals have to be achieved in sequence, so that the message will be printed correctly.

These goals are distributed to the agents by means of missions (a set of goals an agent can commit to). The following missions are proposed:

- print_vowel: the agent responsible for this mission will print the vowels of the message.
- print_l: the mission to print the character l.
- print_consonant: the mission to print the remaining consonants.
- print_special_chars: the mission to print spaces and exclamation marks.



```
kspace jacamo created.
ui: display.GUIConsol
e usa created.
ui: display.GUIConsol
e brazil created.
ui: display.GUIConsol
e created.
lay.GUIConsol
created.
lay.GUIConsol
//193.49.165.
```

```
Message at 1441968878825 from alice: Good morning
```

```
Message at 1441968878825 from maria: Bom dia
```

```
Message at 1441968878825 from francois: Bonjour
```

```
Message at 1441968878825 from giacamo: Buon giorno
```

Figure B.2 – Console of Hello World using artifacts.

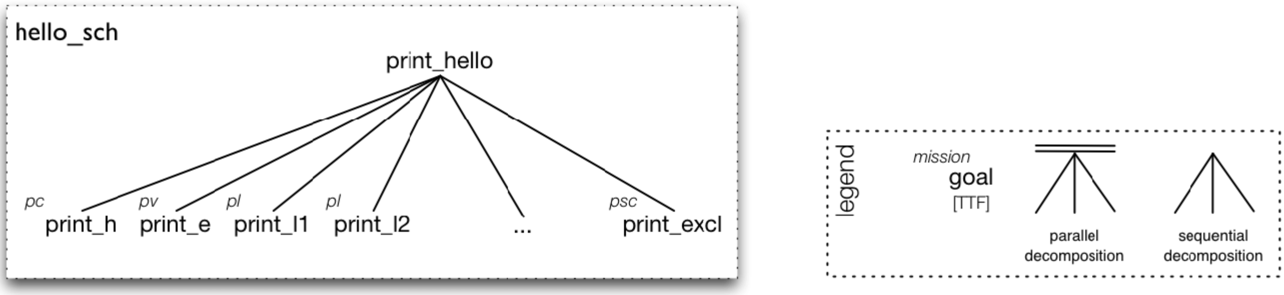


Figure B.3 – Illustrating a mission of the desired Hello World organisation.

The combination of goals, plans and missions is called a social scheme in Moise. In our example, the social scheme is identified by hello_sch. The diagram shown above, in Moise notation, represents the social scheme. Before committing to the missions, the agents have to play roles in the group responsible for the social scheme. As illustrated in Figure B.4 using Moise notation, a group should be defined with roles corresponding to the missions:

- rv: the agent playing this role is obliged to commit to the mission print_vowel.
- rl: the role obliged to commit to the mission print_l.
- rc: the role obliged to commit to the mission print_consonant.
- rs: the role obliged to commit to the mission print_special_chars.

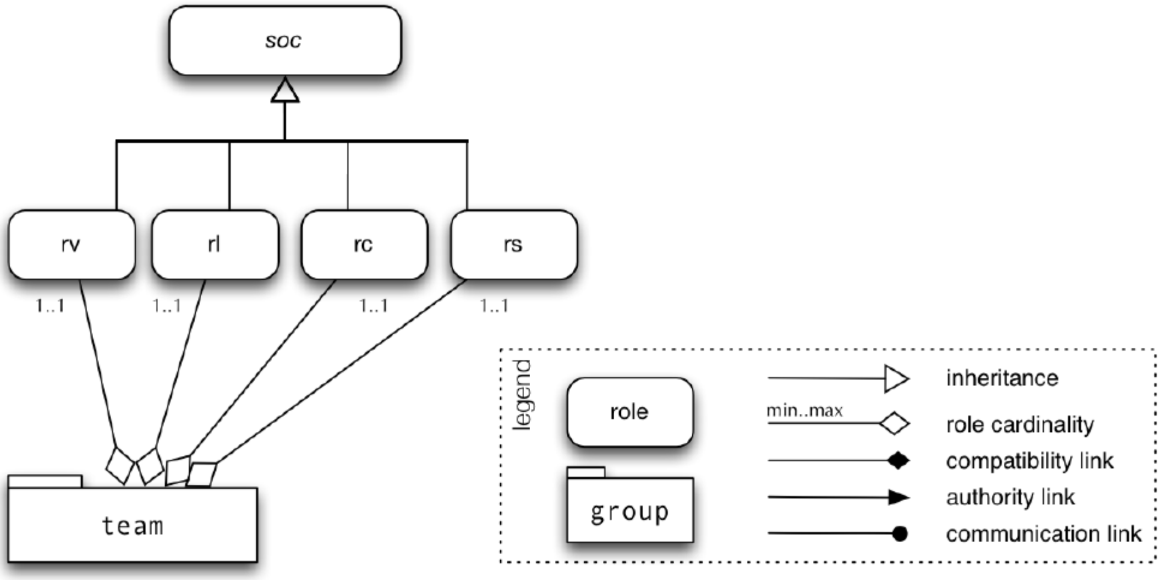
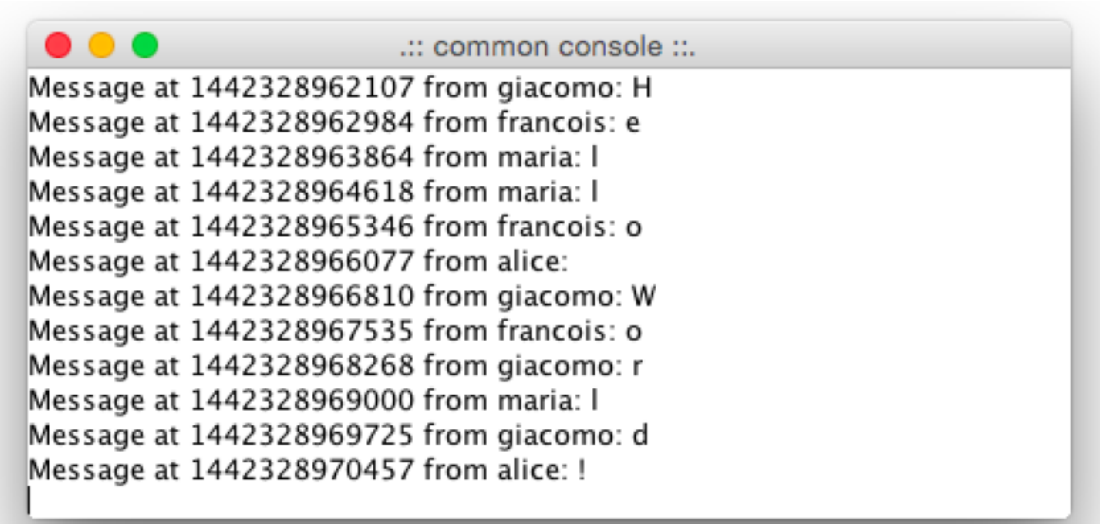


Figure B.4 – Illustrating a group of the desired Hello World organisation.

Based on this specification, francois will play role rv, maria the role rl, giacomo the role rc, alice the role rs in the group jacamo_team which is of type team. This group is responsible for executing the social scheme hello_eng of type hello_sch.

The expected result for your system, considering the organisation, is the following:



```
Message at 1442328962107 from giacomo: H
Message at 1442328962984 from francois: e
Message at 1442328963864 from maria: l
Message at 1442328964618 from maria: l
Message at 1442328965346 from francois: o
Message at 1442328966077 from alice: 
Message at 1442328966810 from giacomo: W
Message at 1442328967535 from francois: o
Message at 1442328968268 from giacomo: r
Message at 1442328969000 from maria: l
Message at 1442328969725 from giacomo: d
Message at 1442328970457 from alice: !
```

Figure B.5 – The console output resulting from this Hello World scenario.

APPENDIX C – GOLD MINERS SPECIFICATION AS USED IN THE EXPERIMENTS

This specification was used as our experiment scenario and delivered to the participants as follows. Your task is to use modelling approaches to initiate the design of a Multi-Agent System (MAS). Consider that JaCaMo is going to be the coding framework for your solution. You have two alternatives for modelling your MAS: OntoMAS or Prometheus. This textual specification describes a problem, and your design should include as much detail as possible in order to facilitate the life of the programmers following your models. Things that you should consider for improving the level of detail of the solution that you will propose:

- Think about which strategies could be used and formalize them in your models.
- Think about the characteristics of your system in terms of required and desirable characteristics for your agents, environment and organisation.
- Always have in mind that your models should be for JaCaMo-oriented implementation.

Scenario. Recently, rumours about the discovery of gold became public and hordes of gold miners appear in the hope to collect as much gold nuggets as possible. You have a small team of gold miners exploring the area, avoiding trees, and competing for the gold nuggets spread around the woods. Your gold miners? team coordinates its actions in order to collect as much gold as possible, while competing with an unknown opponent team. Your miners have to deliver the gold in a depot so it can be safely stored.

Technical Description. In each simulation, the simulation server works in a cyclic fashion: it provides sensory information about the environment to the participating agents and expects agent's reaction within a given time limit. Each agent reacts to the received sensory information by indicating which action (including the skip action) it wants to perform in the environment. If no reaction is received from the agent within the given time limit, the simulation server assumes that the agent performs the skip action. Agents have only a local view on their environment, i.e., their perceptions can be incomplete. After a finite number of steps the simulation server stops the cycle and participating agents receive a notification about the end of a match.

Team, Match, and Simulation. An agent team consist of 4 software agents with distinct IDs. Each simulation in a match starts by notifying the agents from the participating teams and distributing them the details of the simulation. These will include for example the size of the grid, depot position, and the number of steps the simulation will perform. A simulation consists of a number of simulation steps. Each step consists of:

1. sending sensory information to agents (one or more), and

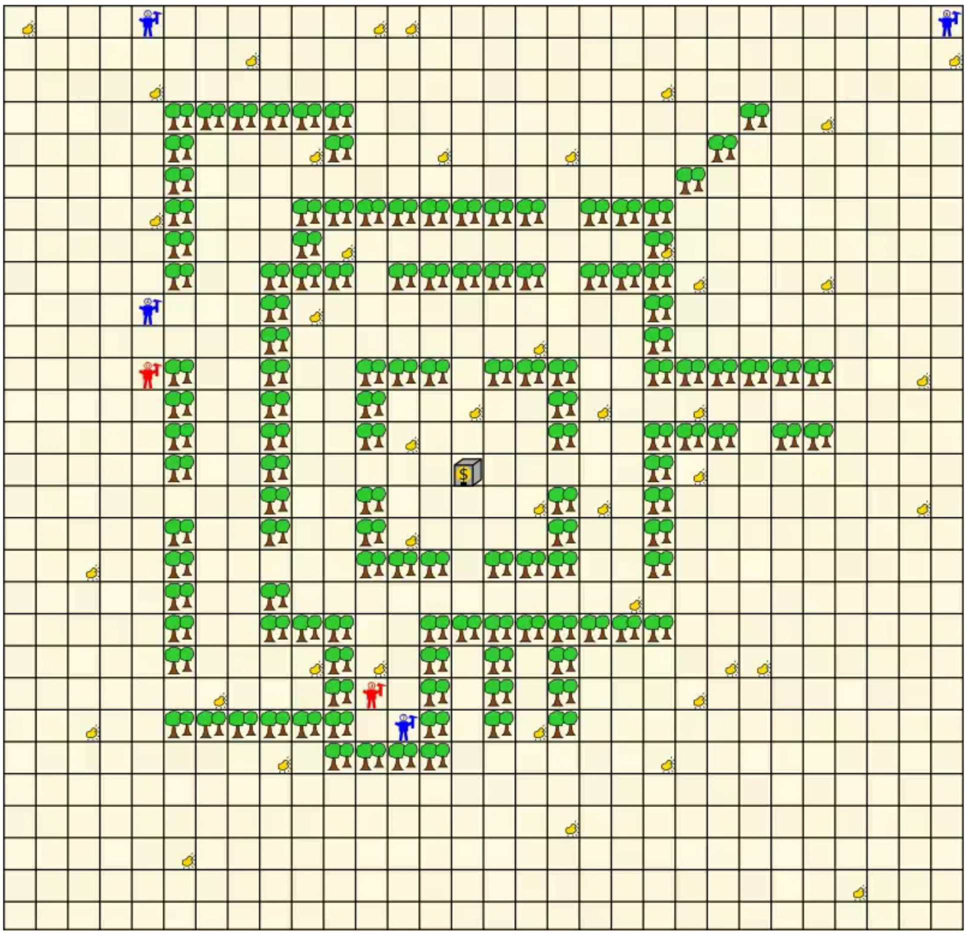


Figure C.1 – Gold Miners execution illustrated.

2. waiting for their actions.

In the case that agent will not respond within a timeout (specified at the beginning of the simulation) by a valid action, it is considered to perform the skip action in the given simulation step.

Environment objects. The (simulated) environment has its size specified at the start of each simulation (it is variable). However, it cannot be more than 100x100 cells. The [0,0] coordinate of the grid is in the top-left corner (north-west). The depots serve for both teams as a location of delivery of gold items. The environment can contain the following elements in its cells:

- obstacle (a cell with an obstacle cannot be visited by an agent)
- gold (an item which can be picked from a cell)
- agent
- depot (a cell to which gold items are to be delivered to earn a simulation point)

There can be only one object in a cell, except that an agent can enter cells containing gold, or depot. At the beginning of a simulation the grid contains obstacles, gold items and agents of both teams. Distribution of obstacles, gold items and initial positions of agents can be either hand crafted for the particular scenario, or completely random. During the simulation, gold items can appear randomly (with a uniform distribution) in empty cells of the grid. The frequency and probability of gold generation will be simulation specific, however not known to neither agents, nor participants. At the start of each simulation agents will get details of the environment (grid size, depot position, etc.). Agents will get information about their initial position in the perception information of the first simulation step.

Perception. Agents are located in cells of the grid and the simulation server provides each agent the following information: absolute position of the agent in the grid the content of the cells surrounding the agent and the content of the cell in which the agent currently stands in (9 cells in total)

Actions. Agents are allowed to perform one action in a simulation step. Agents do not get immediate feedback on their actions, but can learn about the effects of their actions (and the actions of other agents) from the perception information that will be sent to them in the next simulation step. All actions, except the skip action, can fail. An action can fail because the conditions for its successful execution are not fulfilled. The result of a failed action is the same as the result of the skip action. The following actions are allowed:

- move east, move north, move west and move south: An agent can move in four directions in the grid. The execution of move east, move north, move west, and move south changes the position of the agent one cell to the left, up, right, and down, respectively. A movement action succeeds only when the cell to which an agent is about to move does not contain another agent or obstacle. Moving to and from the depot cell is regulated by additional rules described later in this description.

- pick: An agent can pick up a gold item if 1) the cell in which an agent currently stands in contains the gold, and 2) the agent is currently not carrying another gold item. An agent can carry only one gold item which it successfully picked up before. The result of a successful pick action is that in the next simulation step the acting agent will be considered to carry a gold item and the cell, it currently stands in, will not contain the gold item any more.

- drop: An agent can drop the gold item it carries only in the cell it currently stands in. The result of a drop action is that the acting agent is not carrying the gold item any more and that the cell it currently stands in will contain the gold item in the next simulation step. Dropping a gold item to a depot cell increases the score of the agent's team by one point. The depot cell will never contain a gold item that can be picked by an agent.

- skip: The execution of the skip action does not change the state of the environment (under the assumption that other agents did not change it). When an agent does not respond to a perception information provided by the simulation server within the given time limit, the agent is considered as performing the skip action.

Depot cell. There are strong conditions imposed on the depot cell:

1. an agent not carrying a gold item is unable to enter the depot cell (the result of such an action is the same as if the depot was an obstacle);
2. agent which entered the depot cell should drop the gold item as the very next action it execute;
3. after dropping the gold item in a cell, an agent has to leave the cell in the first subsequent simulation step when he will be able to move (i.e. when there was an empty cell at the time of agent's move action). If an agent does not leave the depot in the first possible opportunity, or will not drop the gold item as the very next action after entering the depot, the simulation server will punish it by "teleporting" it away (it will be moved to a random cell not containing another agent, or obstacle in the grid by the environment simulator).

APPENDIX D – DRAG-AND-DROP TRANSFORMATIONS FROM ONTOMAS TO JACAMO

The transformations of drag-and-drop are defined as follows:

- An instance of the **ExternalAction** concept becomes, in:
  - **Jason**, an external action invocation inside a plan's body that represents an agent acting in the environment. The action may contain zero or more parameters and programmers have to complete using the parenthesis. This construction uses the following format:

    $actionName();$
  - **CArtAgO**, an operation invocation within a method which represents an artifact executing the given operation, which also may receive parameters inside the parenthesis, according with the following format:

    $operationName();$
  - **Moise or JCM file**, it is not applicable for drag-and-drop code transformations.

- An instance of the **InternalAction** concept becomes, in:
  - **Jason**, an internal action invocation inside a plan's body that represents an action that an agent performs as part of its reasoning cycle, in the format:

    $.actionName();$
  - **CArtAgO, Moise or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of a subclass of the **Agent** concept becomes, in:
  - **Jason**, the corresponding $agentName$, which identifies an individual agent in order to send messages, or perform some other tasks.
  - **CArtAgO**, the corresponding $agentName$ in a string format, which identifies an individual agent so the environment may be programmed to react accordingly.
  - **JCM file**, an instantiation of an individual agent of type $agentType$ with the identification $agentName$ which can be configured by some parameters in the format:

    $agent\ agentName : agentType.asl \ \{\ parameters\ \}$

    In some cases, depending on the context, only the $agentName$ may be required.
  - **Moise**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Belief** concept (or one of its subclasses) becomes, in:
  - **Jason**, a belief addition event using the given $beliefName$ and $source$ in the format described below. The value in the $source$ is defined by the belief's subtype: self, percept, or other agent. If the programmer wants to remove that belief instead of adding it, then the $+$ symbol must be replaced for $-$. Also, in the context of a plan, the $beliefName(ParameterList)$ could be used to query parameter values for the given type of belief.

    $$+beliefName[source(value)];$$

  - **CArtAgO**, an update in the value of the corresponding observable property in the format given below, if it is a percept-type belief. For beliefs of other sources, e.g. self or another agent, there is no related code to be generated regarding the environment.

    $$getObsProperty(perceptName).updateValue(newValue);$$

  - **JCM file**, the definition of a belief identified as $beliefName$ that may be used for example as initial beliefs parameters of a given instantiated agent in the format given below, if it is a self-type belief. For beliefs of other sources, e.g. percept or another agent, there is no related code to generated regarding the initial configuration file.

    $$beliefs : beliefName$$

  - **Moise**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **AchievementGoal** concept becomes, in:
  - **Jason**, an achievement goal on the format given below that means one of the following two alternatives, depending on the context: *(i)* if dropped outside a plan body, then it is an initial goal for that agent; or *(ii)* if dropped inside a plan body, then it is a goal that has to be achieved during the execution of that plan.

    $$!achievementGoalName;$$

  - **CArtAgO**, an information that artifacts can send to agents in order to initiate the achievement of a goal in the form of:

    $$signal(!achievementGoalName);$$

  - **JCM file**, the definition of an initial goal identified as $goalName$ that can be assigned to agent instances in the format:

    $$goals : goalName$$

  - **Moise**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **TestGoal** concept becomes, in:
  - **Jason**, a test goal that may be dropped inside a plan body to represent a goal that has to be tested during the execution of that plan. The format is as follows:

    $$?testGoalName;$$
  - **CArtAgO**, an information that artifacts can send to agents in order to initiate the test of a goal in the form of:

    $$signal(?testGoalName)$$
  - **Moise or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Message** concept, or one of its subclasses, becomes, in:
  - **Jason**, the act of sending the corresponding instance of message in the format given below, where the value of $receiver$ is obtained from the ontology representation, and the $illocutionaryForce$ value is obtained from the instance type. There are 9 illocutionary forces, one for each subclass of $Message$ in the ontology: $tell$, $untell$, $achieve$, $unachieve$, $tellHow$, $untellHow$, $askOne$, $askAll$, and $askHow$. The (propositional) $concent$ of messages may be instances of beliefs, goals, or plans.

    $$.send(receiver, illocutionaryForce, content);$$
  - **CArtAgO**, a signal where an artifact requests the sending of the corresponding message in the format:

    $$signal(!send(receiver, act, value));$$
  - **Moise or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Plan** concept becomes, in:
  - **Jason**, a plan in the format given below, in which it has a triggering condition, a context (that by default is $true$), and a body composed (mainly) of actions and goals.

    $$is\_triggered\_by : true <- actions; goals.$$
  - **CArtAgO**, a signal with the triggering condition corresponding to that plan in the format:

    $$signal(triggeringCondition);$$
  - **Moise**, the description of how the plan is decomposed in its goals as follows:

$$< plan\ operator = \text{“}sequence\text{”} >< goal\ id = \text{“}goalN\text{”} / >< /plan >$$

  - **JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Space** concept becomes, in:
  - **Jason**, an action for that agent to join the corresponding workspace according with the following format:

    $joinWorkspace(\text{“}workspaceName\text{”});$
  - **CArtAgO**, an update to the position of this artifact in the format:

    $updatePosition(new\ AbstractWorkspacePoint());$
  - **JCM file**, the definition of a workspace with the given $spaceID$ which can be configured by some parameters in the format:

    $workspace\ spaceID\ \{\ parameters\ \}$
  - **Moise**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of a subclass of the **Artifact** concept becomes, in:
  - **Jason**, the action of focusing on that instance of artifact identified by the token $artifactName$ using the following format:

    $focus(artifactName);$
  - **CArtAgO**, the declaration and initialisation of a new instance identified by the token $individualName$ and belonging to the type $ClassName$ according to the following format:

    $ClassName\ individualName = new\ ClassName();$
  - **JCM file**, an instantiation of an artifact of type $className$ with the identification $artifactName$ in the format:

    $artifact\ artifactName$: $className()$
  - **Moise**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Operation** concept becomes, in:
  - **Jason**, the invocation of the corresponding $operationName$ in the body of a plan representing the execution of that operation by an agent such as follows:

    $operationName();$
  - **CArtAgO**, the invocation of the given operation in the format given below which represents an artifact executing the related operation:

$operationName();$

  - **Moise or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **ObservableProperty** concept becomes, in:
  - **Jason**, a plan triggered by the observation of the corresponding property identified as $propertyName$ in the format:

    $+propertyName : true <- planBody.$

    If dropped in the middle of a plan, then just the corresponding belief identified by $propertyName$ may be generated.
  - **CArtAgO**, an update in the value of $propertyName$ according with the following syntax:

    $getObsProperty(propertyName).updateValue(newValue);$
  - **Moise or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **ObservableEvent** concept becomes, in:
  - **Jason**, a plan triggered by the observation of the corresponding event identified as $eventName$ in the format:

    $+eventName : true <- planBody.$
  - **CArtAgO**, the generation of a signal identified as $eventName$, which may be observed by the agents focused in artifacts of the corresponding type, with the following syntax:

    $signal(eventName);$
  - **Moise or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of a subclass of the **Group** concept becomes, in:
  - **Jason**, an action of joining in the given group identified by $groupName$ in the format $join\_group(groupName);$ or it may be suggested the action of creating an instance of that group as follows:

    $create\_group(groupName);$
  - **JCM file**, the definition of a group of the type $groupType$ with the given $groupName$ identification which can be configured by some parameters in the format:

$group\ groupName$: $groupType$ { $parameters$ }

  - **CArtAgO or Moise**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Role** concept becomes, in:

  - **Jason**, the action for an agent plan where it adopts the instance of role defined by the token $roleName$ in the group identified by $groupName$ according with the following syntax:

    $adopt\_role(roleName, groupName);$

  - **Moise**, a role definition specified by $roleName$ which may extend some role referred as $extendsRole$ in the format:

    $< role\ id = “roleName” >< extends\ role = “extendsRole”/ >< /role >$

  - **JCM file**, the definition that an agent will be playing the given role defined by $roleName$ in the group identified by $groupName$, as specified in the format:

    $roles$: $roleName\ in\ groupName$

  - **CArtAgO**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Mission** concept becomes, in:

  - **Jason**, an action to commit with the instance of mission identified by $missionName$ in the scheme defined as $schemeId$. That action may be placed in the body of a plan and it follows the format:

    $commit\_mission(missionName, schemeId);$

  - **Moise**, the specification of the mission defined as $missionName$ which contains a set of goals ranging from $goal1$ until $goalN$ in the following format:

    $< mission\ id = “missionName”\ min = “1”\ max = “1” >< goal\ id = “goal1”/ > ... < goal\ id = “goalN”/ >< /mission >$

  - **CArtAgO or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **Norm** concept becomes, in:

  - **Jason**, a plan triggered by the perception of the norm identified by $normName$ in the format:

    $+normName$: $true$ <- $planBody.$

  - **Moise**, the specification of the norm referred as $normName$ which targets a role and a mission identified by the tokens $targetRole$ and $targetMission$ in the format:

    $< norm\ id\ =\ \text{“}normName\text{”}\ type\ =\ \text{“”}\ role\ =\ \text{“}targetRole\text{”}\ mission\ = \text{“}targetMission\text{”} / >$

  - **CArtAgO or JCM file**, it is not applicable for drag-and-drop code transformations in this case.

- An instance of the **OrganisationGoal** concept becomes, in:

  - **Jason**, a plan triggered by the addition event of the goal identified by $goalName$ according with the syntax:

    $+!goalName$: $true$ <− $planBody.$

  - **Moise**, a specification of a goal referred to as $goalName$ in the following format:

    $< goal\ id = \text{“}goalName\text{”} / >$

  - **CArtAgO or JCM file**, it is not applicable for drag-and-drop code transformations.

APPENDIX E – TEMPLATE CODE GENERATION FROM ONTOMAS SPECIFICATIONS

The transformations to generate template code for JaCaMo are defined as follows:

- A subclass of the **Agent** concept becomes, in:
  - **Jason**, a new .asl file that contains all related elements such as plans, goals, and beliefs is generated for each subclass of **Agent**.
  - **JCM file**, the sources that agent instances are allowed to have are given by the subclasses of **Agent** in the ontology.
  - **CArtAgO or Moise**, it is not directly applicable for automatic code generation in this case.

- An instance of one of the subclasses of **Agent** becomes, in:
  - **JCM file**, a declaration that instantiates an agent of type $className$ with the identification $agentID$, which can be configured by some parameters in the format:

    $agent\ agentID : className.asl\ \{\ parameters\ \}$
  - **Jason, CArtAgO or Moise**, it is not directly applicable for automatic code generation in this case.

- An instance of the **Belief** concept (or one of its subclasses) becomes, in:
  - **Jason**, an initial belief in the corresponding .asl file (the agent type is defined by the related subclass of agent). The belief may be annotated with $[source(value)]$, where the $value$ is defined by the belief's subtype: self, percept, or other agent. The adopted syntax is as follows:

    $beliefName[source(value)]$
  - **JCM file**, an initial belief of the related instances of agents in the corresponding individual agents' definition.
  - **CArtAgO or Moise**, it is not directly applicable for automatic code generation.

- An instance of the **AchievementGoal** concept becomes, in:
  - **Jason**, an initial achievement goal in the corresponding .asl file, where the agent type is defined by the related subclass of agent.

  - **JCM file**, an initial achievement goal of the related instances of agents in the corresponding individual agents' definition.
  - **CArtAgO or Moise**, it is not directly applicable for automatic code generation in this case.

- An instance of the **Message** concept becomes, in:
  - **Jason**, a plan to send the corresponding message in the format given below, where the value of $receiver$ is obtained from the ontology representation, and the $illocutionaryForce$ value is obtained from the instance type. There are 9 illocutionary forces, one for each subclass of $Message$ in the ontology: $tell$, $untell$, $achieve$, $unachieve$, $tellHow$, $untellHow$, $askOne$, $askAll$, and $askHow$. The (propositional) $concent$ of messages may be instances of beliefs, goals, or plans. A plan for the agent $receiver$ may be created taking as triggering condition the consequences of receiving such $concent$ in the given $illocutionaryForce$.

    $$!sendMsgName <- .send(receiver, illocutionaryForce, content);$$

  - **CArtAgO, Moise or JCM file**, it is not directly applicable for automatic code generation in this case.

- An instance of the **Plan** concept becomes, in:
  - **Jason**, a plan in the format given below in which it has a triggering condition, a context (that by default is $true$), and a body composed (mainly) of actions and goals. The plan is inserted in the .asl file of the corresponding type of agent that has it.

    $$is\_triggered\_by : true <- actions, goals.$$

  - **CArtAgO, Moise or JCM file**, it is not directly applicable for automatic code generation in this case.

- Instances of the concepts **Action**, **InternalAction**, **ExternalAction**, and **TestGoal** are not directly applicable for automatic code generation in any dimension of JaCaMo. However, these instances are used when converting entities of **Plan** to code.

- An instance of the **Space** concept becomes, in:
  - **JCM file**, the definition of a workspace with the given $spaceID$ which can be configured by some parameters in the format:

    $$workspace\ spaceID\ \{\ parameters\ \}$$

  - **Jason, CArtAgO or Moise**, it is not directly applicable for automatic code generation in this case.

- A subclass of the **Artifact** concept becomes, in:
  - **CArtAgO**, Each subclass of **Artifact** generates a new .java file composed with all related elements such as operations, and observable properties.
  - **JCM file**, the sources that artifact instances are allowed to have are given by the subclasses of **Artifact** in the ontology.
  - **Jason or Moise**, it is not directly applicable for automatic code generation in this case.

- An instance of any subclass of the **Artifact** concept becomes, in:
  - **JCM file**, an instantiation of an artifact of type $className$ with the identification $artifactName$ in the format given below, in which it should be placed inside a declaration of a workspace.

    $$artifact\ artifactName : className()$$

  - **Jason, CArtAgO or Moise**, it is not directly applicable for automatic code generation in this case.

- An instance of the **Operation** concept becomes, in:
  - **CArtAgO**, a method representing the operation, with the specified $operationName$, in the form given below that may contain zero or more parameters.

    $$@OPERATION\ void\ operationName()\{\}$$

  - **Jason, Moise or JCM file**, it is not directly applicable for automatic code generation in this case.

- An instance of the **ObservableProperty** concept becomes, in:
  - **CArtAgO**, the definition of the corresponding observable property $propertyName$ inside the $init$ method in the class of the corresponding artifact in the following format:

    $$defineObsProperty(\text{``}propertyName\text{''}, value);$$

  - **Jason, Moise or JCM file**, it is not directly applicable for automatic code generation in this case. However, a plan triggered by the addition event of the related observable property could be suggested to programmers of the agents as a situation that could be desired to be handled.

- An instance of the **ObservableEvent** concept becomes, in:
  - **Jason**, it may be suggested (however, it is not mandatory) a plan triggered by the observation of the corresponding event identified as $eventName$ in the format:

    $+eventName : true$ <− $planBody.$
  - **CArtAgO**, the artifact's class code may indicate that this signal can be sent by this artifact. This would be implemented as exemplified below considering an observable event identified by $eventName$.

    $signal(eventName);$
  - **Moise or JCM file**, it is not directly applicable for automatic code generation in this case.

- A subclass of the **Group** concept becomes, in:
  - **Moise**, a declaration of a group specification identified by $groupName$ that contains all related roles in the format:

    $< group - specification\ id = \text{“}groupName\text{”} >< role\ id = \text{“}role1\text{”} / > ... < role\ id = \text{“}roleN\text{”} / >< /group - specification >$
  - **Jason, CArtAgO or JCM file**, it is not directly applicable for automatic code generation in this case.

- An instance of any subclass of **Group** becomes, in:
  - **JCM file**, the definition of a group of the type $className$ with the given $groupID$ identification which can be configured by some parameters in the format:

    $group\ groupID$: $className$ { $parameters$ }
  - **Jason, CArtAgO or Moise**, it is not directly applicable for automatic code generation in this case.

- An instance of the **Role** concept becomes, in:
  - **Moise**, a role definition specified by $roleName$ which may extend some role referred as $extendsRole$ in the format:

    $< role\ id = \text{“}roleName\text{”} >< extends\ role = \text{“}extendsRole\text{”} / >< /role >$
  - **JCM file**, the definition that an agent will be playing the given role defined by $roleName$ in the group identified by $groupName$, as specified in the format:

    $roles$: $roleName\ in\ groupName$
  - **Jason or CArtAgO**, it is not directly applicable for automatic code generation.

- An instance of the **Mission** concept becomes, in:
  - **Moise**, the specification of the mission defined as $missionName$ which contains a set of goals ranging from $goal1$ until $goalN$ in the following format:

    $< mission\ id = “missionName”\ min = “1”\ max = “1” >< goal\ id = “goal1”/ > ... < goal\ id = “goalN”/ >< /mission >$
  - **Jason, CArtAgO or JCM file**, it is not directly applicable for automatic code generation in this case.

- An instance of the **Norm** concept becomes, in:
  - **Moise**, the specification of the norm referred as $normName$ which targets a role and a mission identified by the tokens $targetRole$ and $targetMission$ in the format given below. The $normType$ is defined by the subclass of the Norm concept: obligation, prohibition, or permission.

    $< norm\ id = “normName”\ type = “normType”\ role = “targetRole”\ mission = “targetMission”/ >$
  - **Jason, CArtAgO or JCM file**, it is not directly applicable for automatic code generation in this case.

- An instance of the **OrganisationGoal** concept becomes, in:
  - **Moise**, a specification of a goal referred to as $goalName$ in the following format:

    $< goal\ id = “goalName”/ >$
  - **Jason, CArtAgO or JCM file**, it is not directly applicable for automatic code generation in this case.