A Model-Driven Approach for Organizations in Multiagent Systems

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Cristián Madrigal Mora

Deutsches Forschungszentrum für Künstliche Intelligenz

Saarbrücken
2012

Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr. Mark Groves
Vorsitzender der Prüfungskommission	Prof. Dr. Antonio Krüger
Berichterstatter	Prof. Dr. Jörg Siekmann
Berichterstatter	Prof. Dr. Philipp Slusallek
Berichterstatter	Prof. Dr. Andreas Zeller
Tag des Promotionskolloquiums	20.09.2013

Acknowledgements

First of all, I would like to thank the members of my PhD committee for their guidance and support in the final stages of my PhD.

To Prof. Dr. Jörg Siekmann, thank you for giving me the opportunity to work under your supervision. I will always treasure all the lessons—both personal and professional—that I learned from you and other members of the multiagent group during this part of my life.

To the members of the multiagents group at DFKI, especially Prof. Dr. Philipp Slusallek and Dr. Klaus Fischer, thank you for all the fruitful discussions, the constructive criticism and for making the institute such a great place to work in.

To my officemates, Esteban, Stefan and Sven, thanks for always being there to bounce ideas around no matter how bad they were.

To all my friends, thanks giving me a small push when I felt stuck and for celebrating with me every time I reached a milestone in this journey.

Dedication

To my family, for always pushing and allowing me to follow my dreams

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den 1.11.2012

(Unterschrift)

Kurzfassung

In dieser Arbeit wird ein neuer Modell-basierter Ansatz für die Agenten-orientierte Softwaretechnik vorgestellt, bei dem Agenten-Organisationen nicht nur eine entscheidende Rolle spielen, sondern auch auf allen Abstraktionsebenen vertreten sind. In der dargestellten Methodik werden Multiagenten-Systeme auf einer Plattform-unabhängigen Ebene modelliert und dann in ein Plattform-spezifisches Modell umgewandelt, wobei die Organisationsstrukturen erhalten bleiben. Der Ansatz wurde über einige Jahre kontinuierlich verfeinert und bereits in zwei Projekten der Europäischen Union implementiert.

Abstract

This thesis introduces a new model-driven approach to agent-oriented software engineering in which agent organizations not only play a crucial role, but are also represented in every abstraction level. In our methodology, multiagent systems are modeled at a platform-independent level and transformed into a platform-specific level preserving the organizational structures. The approach has been refined through several years and has been used in two European Union projects.

Zusammenfassung

In dieser Arbeit wird ein neuer Modell-basierter Ansatz für die Agentenorientierte Software-Technologie vorgestellt, bei dem Agenten-Organisationen nicht nur eine entscheidende Rolle spielen, sondern auch auf allen Abstraktionsebenen vertreten sind. In der dargestellten Methodik werden Multiagenten-Systeme auf einer Plattform-unabhängigen Ebene modelliert und dann in ein Plattform-spezifisches Modell umgewandelt, wobei die Organisationsstrukturen erhalten bleiben.

Um Systementwickler bei der effektiven Modellierung zu unterstützen, wird hier eine Methodik beschrieben, die die Erstellung verschiedener Modellansichten anleitet, und zwar so, dass zugleich auch die Abhängigkeiten zwischen diesen Ansichten berücksichtigt werden können. Abhängig von den Systemvoraussetzungen oder Entwicklerpräferenzen hat der Systementwickler die Möglichkeit, das System ziel- oder verhaltensorientiert zu modellieren. Zielorientiert bedeutet hier: die Verantwortlichkeiten für jede Rolle in der Organisation werden als Ziele modelliert. Diese Ziele umfassen der Organisation übergeordnete Ziele, Organisationsziele und Agentenziele. Beim verhaltensorientierten Ansatz werden die Verantwortlichkeiten für die Rollen als Verhalten modelliert. Hierbei ist das Erreichen von Zielen bereits in der erfolgreichen Durchführung des Verhaltens impliziert. Mit Ausnahme der Phasen der Zieldefinition ist die Methodik für beide Ansätze die gleiche. Definiert werden müssen: ein Informationsmodell, Rollen gemäß ihrer Verantwortlichkeiten (Zielen oder Verhaltensweisen), Organisationsstrukturen und ihre Beziehungen mittels Rollen, Kommunikationsprotokolle, detaillierte Abläufe der Verhaltensweisen sowie die Konfiguration für die Bereitstellung des MAS.

Die Plattform-unabhängigen Konzepte werden in einem Metamodell namens PIM4Agents definiert. Die Modellierungstools für PIM4Agents unterstützen jede Phase der Methodik und liefern für jede Ansicht ein grafisches

Modell. Zudem wird bei Speicherung der Modelle eine Modell-Validierung durchgeführt. Diese verhindert, dass das Modell inkonsistent wird, und garantiert, dass die nachfolgenden Transformationen problemlos funktionieren.

Ist eine Agentenplattform als Ziel gewählt, werden die Modelle in ein Plattform-spezifisches Modell (PSM) für diese Plattform umgewandelt. In dieser Arbeit wird das PSM JadeOrgs für die Jade Agentenplattform vorgestellt. JadeOrgs bietet die in der Agentenplattform verfügbaren Modellkonstrukte und ergänzt diese durch weitere Konstrukte die für die Repräsentation der Organisationsstrukturen, ihrer Rollen und ihrer Verantwortlichkeiten benötigt werden. Darüber hinaus wurde mithilfe der Spezifikationssprache Object-Z eine formale Definition dieser Strukturen erstellt. Da diese Konstrukte nicht in der Jade Agentenplattform implementiert sind, umfasst JadeOrgs auch eine Programmierschnittstelle und eine Laufzeitkomponente, so dass diese Strukturen auch während der Ausführung des modellierten MAS verfügbar sind. Um die verschiedenen Abstraktionsebenen miteinander zu vereinbaren, wird eine Reihe von Transformationen definiert. Diese bestehen einerseits aus einer Reihe von Konzept-Mappings von einer Abstraktionsebene zur nächsten sowie andererseits aus einem Satz an Quelltextvorlagen für die Serialisierung des PSMs zu Java-Quelltext.

Um die Realisierbarkeit solcher Modelle und Transformationen aufzuzeigen, werden zwei Anwendungsszenarien beschrieben. Im ersten wird eine frühe Version von JadeOrgs für einen Proof-of-Concept im Kontext einer Service-orientierten Architektur verwendet. Im zweiten werden PIM4Agents und JadeOrgs für ein Szenario in der Stahlproduktion verwendet.

Summary

In this thesis, we introduce a new model-driven approach to agent-oriented software engineering in which agent organizations not only play a crucial role, but are also represented in every abstraction level. In our approach, multi-agent systems are modeled at a platform-independent level and transformed into a platform-specific level preserving the organizational structures.

In order to assist the system designer to model effectively, we describe a methodology that guides the creation of various model views in a fashion consequent with the dependencies between these views. Depending on the system requirements or designer preference, the system designer has the option to model the system in a goal-driven or a behavior-driven fashion. In the goal-driven fashion, the responsibilities for each role are modeled as goals. These goals include system overall goals, organization goals and the goals for each agent type. In the behavior-driven way, the achievement of goals is implicit in the successful completion of the behaviors and the roles in the system depend on their required behaviors. Aside for the goal definition stages, the methodology is the same for both variations by defining: the information model, the roles with respect to the responsibilities (goals or behaviors), the organizational structures and their relationships through roles, the communication protocols, the detailed process entailed by each behavior, and the deployment configuration of the MAS.

The platform independent concepts are defined in a metamodel called PIM4Agents. The modeling tools for PIM4Agents support each of the methodology stages and provide a graphical model for each of the views. In addition to the graphical modeling support, model validation is performed on the saved models. The validation avoids the introduction of inconsistencies in the model and guarantees that the following transformations will work successfully.

Once a target agent platform is chosen, the models are transformed into

a Platform Specific Model (PSM) for the given platform. In this thesis, we introduce a PSM for the Jade Agent Platform called JadeOrgs. JadeOrgs provides the modeling constructs available in the agent platform and extends this set with the constructs necessary to represent the organizational structures, their roles and their responsibilities. In addition, a definition for these structures was formalized using the Object-Z specification language. As these constructs are not implemented in the Jade Agent platform, JadeOrgs also includes a programming API and a runtime component so that these structures are also available during the execution of the modeled MAS. In order to connect the different abstraction levels, a series of transformations are defined. They consist in a series of maps of concepts from one abstraction level to the next and in a set of code templates in the serialization stage of the PSM to Java code.

In order to demonstrate the viability of such models and transformations, we describe two application scenarios. In the first one, an early version of JadeOrgs is applied to a proof-of-concept system in the context of service oriented architectures. In the second one, PIM4Agents and JadeOrgs are applied to a scenario in steel production.

Contents

I Introduction 1

1 Motivation 3
- 1.1 Objectives . . . 5
- 1.2 Contributions . . . 5
- 1.3 Structure . . . 7

2 Background 9
- 2.1 Foundations of Multiagent Systems . . . 10
  - 2.1.1 Basic concepts . . . 10
  - 2.1.2 Types of agents . . . 11
  - 2.1.3 Interaction . . . 16
  - 2.1.4 Communication . . . 18
- 2.2 Model-Driven Software Development . . . 22
  - 2.2.1 Models and Metamodels . . . 23
  - 2.2.2 Model Transformations . . . 24
  - 2.2.3 The Abstraction Levels of the Model Driven Architecture 25

3 State of the Art in AOSE 28
- 3.1 Agents and the Unified Modeling Language . . . 29
- 3.2 MAS Metamodels and Frameworks . . . 30
- 3.3 Agent-Oriented Programming Languages . . . 32
- 3.4 Model-Driven Development of MAS . . . 33

II A Model-Driven Approach for Organizations in Multiagent Systems 35

4 Modeling Agent Organizations 37
- 4.1 Modeling of the Abstract Goals . . . 39

4.2 Definition of the Information Model 40
4.3 Refinement of the Abstract Goals into Concrete Goals 40
4.4 Modeling of the Roles in the System 41
4.5 Modeling of Organizational Structures and Relations between Roles 42
4.6 Modeling of the Communication Patterns 43
4.7 Definition of the Detailed Behaviors of the Agents 43
4.8 Establishment of the Initial Deployment Configuration of the System 44

5 PIM4Agents 46
5.1 Multiagent System Viewpoint 48
5.2 Agent Viewpoint 48
5.3 Organization Viewpoint 50
5.4 Goal Viewpoint 52
5.5 Role Viewpoint 54
5.6 Behavior Viewpoint 55
5.7 Interaction Viewpoint 59
5.8 Information Model Viewpoint 61
5.9 Deployment Viewpoint 63
5.10 Case Study: The Conference Management System 64
5.10.1 The CMS Goal Model 65
5.10.2 The CMS Information Model 68
5.10.3 The CMS Role Model 69
5.10.4 The CMS Organization Model 71
5.10.5 The CMS Agent Model 73
5.10.6 The CMS Interaction Model 75
5.10.7 The CMS Behavior Model 77
5.10.8 The CMS Deployment Model 78
5.11 Summary 80

6 JadeOrgs 82
6.1 Formal Specification of Organizations in JADE 83
6.1.1 Basic types 84
6.1.2 Condition 85
6.1.3 Goal 85
6.1.4 Variable 86

6.1.5 Message . . . 86
6.1.6 Plan . . . 87
6.1.7 Role . . . 88
6.1.8 Agent . . . 90
6.1.9 Organization . . . 92
6.1.10 AgentPlatform . . . 95
6.2 The JadeOrgs metamodel . . . 96
6.2.1 The JadeOrgs Project View . . . 97
6.2.2 The JadeOrgs Core View . . . 97
6.2.3 The JadeOrgs Behavioral View . . . 99
6.2.4 The JadeOrgs Process View . . . 101
6.2.5 The JadeOrgs Ontology View . . . 102
6.2.6 The JadeOrgs Deployment View . . . 105
6.3 JadeOrgs protocols and interactions . . . 106
6.3.1 Publishing to the Directory Facilitator . . . 106
6.3.2 Establishment of the Organization . . . 107
6.3.3 Task Request and Goal Achieve . . . 109
6.4 Small Example: Product Sale with Loan . . . 110
6.5 Related works . . . 114
6.5.1 Metamodel comparison . . . 114
6.5.2 Other approaches to runtime organizations in JADE . 117
6.6 Summary . . . 119

7 Transforming PIM4Agents into JadeOrgs 120
7.1 The Mapping Rules . . . 121
7.2 Generated JadeOrgs models . . . 125
7.3 Code Serialization . . . 127
7.4 Summary . . . 128

III Applications and Conclusions 129

8 Proof of Concept: Modeling e-Procurement with PIM4SOA and JadeOrgs 132
8.1 Metamodel for Service-Oriented Architectures . . . 135
8.1.1 Service Metamodel . . . 135
8.1.2 Process Metamodel . . . 138
8.1.3 Information Metamodel . . . 140

8.2 Model to Model Transformations 140
8.3 From PIM4SOA to JadeMM 142
8.4 Use Case Scenario 147
8.4.1 SOA Model in accordance to the PIM4SOA 147
8.4.2 Applying the transformation from PIM4SOA to JadeMM 149
8.5 Summary 153

9 Case Study: Applying Multiagent Systems to a Steel Production Process 154
9.1 Scenario Description 156
9.2 Methodology 159
9.3 Modeling the PIM Layer with SoaML 161
9.4 Corresponding PIM4Agents Models 164
9.5 Scenario Evaluation 170
9.5.1 SWOT Analysis 172
9.6 Summary 174

10 Conclusions and Future Work 176
10.1 Summary 177
10.2 Future Work 178
10.2.1 Role Deployment Dynamics 178
10.2.2 Norms and Electronic Institutions 179
10.2.3 Bottom-up Approach 180
10.2.4 Other Application Domains 180

List of Figures 184

References 184

Index 196

Part I

Introduction

Chapter 1

Motivation

Agent-oriented software engineering (AOSE) is rapidly emerging in response to needs in both software engineering and agent-based computing. While these two disciplines co-existed without much interaction until some years ago, today there is rich and fruitful interaction among them and various approaches are available that bring together techniques, concepts and ideas from both sides.

Model-Driven Development (MDD) and Model-Driven Architecture (MDA), as its most prominent initiative, are a recent trend in the area of software engineering [Obj03]. This thesis focuses on translating the basic ideas of MDD into methodologies, techniques and tools for the design of agent- and organization-based systems, and in doing so contributes to bridging the gap between traditional software engineering approaches and agent-based system design. Moreover, we not only need to integrate MDD into the methodologies of agent-based system design but also demonstrate how such methodologies can be utilized in practical development frameworks for agent-based system design. In accordance with these goals, some basic issues arise:

- Agent-oriented methodologies often do not rely on existing agent-based development tools, i.e. they do not provide a straightforward interface for implementation.

- Even if existing methodologies have different advantages when applied to particular problems, usually a unique methodology cannot be applied to every problem without some (minor) level of customization and the tools that support these methodologies usually are not open enough for users to perform these customizations.

- Multiagent System (MAS) implementation requires deep knowledge regarding technical details of agent architectures, multiagent development tools, and agent concepts.

The question how to fill the gap between agent methodologies and agent-based development tools leads to the development of a framework that (i) standardizes the design, (ii) simplifies the implementation of agent systems and (iii) allows to integrate already existing agent frameworks into a single tool box in order to increase the degree of utilization in practice. The development of such a framework has been a team effort. Correspondingly, we will provide an overview of the complete framework of metamodels, transformations, runtime components and tools, but we will concentrate on the

elements related to organizational concepts like organizations, roles, goals and the runtime components developed to support the realization of these concepts in running systems.

1.1 Objectives

The main objectives of this thesis are the following:

- Design a model-driven framework for modeling agent organizations, focusing on issues like coordination, task distribution, and role management.
- Determine the necessary methodology steps and tool support for such an agent organization developing process.
- Demonstrate the usability of such a framework by applying it in business and service-oriented scenarios.

The framework that we present in this work aims at providing an open, extensible, and generic way to model MAS and the organizational structures in them. In order to do this, we have aimed at using open source tools, or at least freely available if possible, as the building blocks for the framework. Its tools, metamodels and transformations are also open for third parties to extend, modify and improve. In the same spirit, it is our aim that the methodology provides a guidance for designers new to MAS development, but it should not restrict them in the customization of their own development process, which could include methodology fragments from other tools and methodologies.

1.2 Contributions

The contributions that have resulted from the work in this thesis are as follows:

- Development of the tool support for the realization of the methodology steps related to organization composition and responsibility distribution between roles.

- Specification of the organizational concepts for the metamodels at the Platform Independent and Platform Dependent levels of abstraction.
- Definition of transformations between the different metamodels and a serialization that will produce the application source code.
- Development of a formal specification in Object-Z for the runtime organizational structures used in our framework.
- Development of a runtime component that deals with the runtime representation and dynamics of agent organizations on a target agent platform does not natively support them.

These results have been presented in various workshops, conferences and journals and have resulted in the following publications (in chronological order):

- Christian Hahn, **Cristián Madrigal Mora**, Klaus Fischer, Brian Elvesæter, Arne-Jørgen Berre, and Ingo Zinnikus. *Meta-models, Models, and Model Transformations: Towards Interoperable Agents.* In: MATES. Multiagent System Technologies (MATES-2006), Erfurt, Germany, Pages 123-134, Lecture Notes in Computer Science (LNCS), Vol. 4196, ISBN 3-540-45376-8, Springer, 2006.
- Christian Hahn, **Cristián Madrigal Mora**, and Klaus Fischer. *Interoperability through a Platform-Independent Model for Agents.* In: Enterprise Interoperability II: New Challenges and Approaches. Pages 195-206, ISBN 978-1-84628-857-9, Springer London, 2007.
- Klaus Fischer, Christian Hahn, and **Cristián Madrigal Mora**. *Agent-oriented software engineering: a model-driven approach.* In: International Journal of Agent-Oriented Software Engineering (IJAOSE), Vol. 1, No. 3/4, Pages 334-369, Inderscience, 2007.
- **Cristián Madrigal Mora**, Esteban León Soto, and Klaus Fischer. *Implementing Organisations in JADE.* In: MATES. Conference on Multi-Agent System Technologies (MATES-2008). Kaiserslautern, Germany, Pages 135-146, Lecture Notes in Computer Science (LNCS), Vol. 5244, ISBN 978-3-540-87804-9, Springer, 2008.

- **Cristián Madrigal Mora** and Klaus Fischer. Adding Organisations and Roles to JADE with JadeOrgs. In: Agent-Based Technologies and Applications for Enterprise Interoperability. Pages 98-117, Lecture Notes in Business Information Processing (LNBIP), Vol. 25, ISBN 1865-1348 (Print) 1865-1356 (Online), Springer Berlin Heidelberg, 2009.

- Christian Hahn, **Cristián Madrigal Mora**, and Klaus Fischer *A platform-independent metamodel for multiagent systems.* In: Journal of the International Foundation for Autonomous Agents and Multi-Agent Systems, Vol. 18, No. 2, Pages 239-266, Springer Netherlands, 2009.

1.3 Structure

The remainder of this thesis is structured as follows:

Chapter 2 presents the foundations of multiagent systems to provide the background to the presented work, as well as the basics of Model-Driven Development.

Chapter 3 shows an overview of the state of the art in Agent-Oriented Software Engineering in order to provide the context in which this research was developed.

Chapter 4 describes the recommended methodology to model multiagent systems and their organizational structures.

Chapter 5 presents the PIM4Agents, our metamodel for creating platform independent models of multiagent systems, its views and concrete syntax.

Chapter 6 introduces our platform specific metamodel, JadeOrgs. It presents agent organizations in addition to the concepts supported natively by the JADE agent platform.

Chapter 7 consists of the transformations between the metamodels and the serialization of JadeOrgs in Java code.

Chapter 8 describes a proof concept transformation between PIM4SOA and JadeOrgs.

Chapter 9 presents the details of an industrial application scenario in the context of steel production and evaluates the performance of our approach under this scenario.

Chapter 10 concludes this thesis and proposes areas for future research based on the results presented here.

Chapter 2

Background

We aim at a framework for the design and implementation of multiagent systems in a model-driven approach. In the following we present some background on Multiagent Systems and Model Driven Development, which should help to understand the foundations of these areas and the benefits that the application of a model-driven approach can bring.

2.1 Foundations of Multiagent Systems

2.1.1 Basic concepts

According to Russell and Norvig [RN02], the goal of Artificial Intelligence (AI) is "not just to understand but also to build intelligent entities" and these intelligent entities are what is referred to as agents. However, in the literature, there is no universal agreement on the exact definition of what constitutes an agent. Therefore, we provide a couple of definitions that, in our view, present the basic characteristics. First, a compact, but intuitive definition:

Definition 2.1.1 *An* ***agent*** *is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators [RN02].*

The second definition is more detailed and proposed by Wooldridge and Jennings [WJ95]. This definition is also known as the *weak notion of agency*:

Definition 2.1.2 *Perhaps the most general way in which the term* ***agent*** *is used is to denote a hardware or (more usually) software-based computer system that enjoys the following properties:*

autonomy: *agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*

social ability: *agents interact with other agents (and possibly humans) via some kind of agent-communication language;*

reactivity: *agents perceive their environment, and respond in a timely fashion to changes that occur in it;*

pro-activeness: *agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative [WJ95].*

Thus, the idea of agenthood provides a paradigm for study, while focusing on how to make them *intelligent*. Within the subarea of Distributed Artificial Intelligence (DAI) the main focus is to make agents *autonomous* with less emphasis on making them intelligent [Bog07]. Therefore in DAI, and in accordance to the *weak notion of agency* previously presented, agents are usually referred to just as “autonomous agents”.

Definition 2.1.3 ***Autonomous Agents*** *are computational entities such as software programs or robots that can be viewed as perceiving and acting upon their environment and that are autonomous in that their behavior at least partially depends on their experience within the environment [Wei00].*

Hence, DAI concentrates on how agents—and groups of agents—perform tasks together in order to reach their objectives. This leads us to the definition for Multiagent Systems and Distributed Artificial Intelligence, respectively:

Definition 2.1.4 ***Multiagent Systems*** *are systems in which several interacting, autonomous agents pursue some set of goals or perform some set of tasks [Wei00].*

Definition 2.1.5 ***Distributed Artificial Intelligence*** *is a study, construction, and application of Multiagent Systems [Wei00].*

2.1.2 Types of agents

Agents can be classified with respect to the way they choose and execute their actions. In [Woo02] there are the following classes:

- Deductive Reasoning Agents,
- Practical Reasoning Agents,
- Reactive Agents, and
- Hybrid Agents.

Deductive Reasoning Agents

Deductive Reasoning Agents are based on logical formulae to represent their desired behavior and the environment. This representation is manipulated through the use of logical deduction or theorem proving. The idea is that the agent programmer needs to encode a set of deduction rules ρ and the agent's knowledge database Δ. Let α be an action, if a formula $Do(\alpha)$ can be derived, given ρ and Δ, then α is the best action to perform. Let L be the set of sentences of classical first-order logic, and let $D = \mathscr{P}(L)$ be the set of all L databases. Thus, the agent's knowledge database Δ is then an element of D and the agent's action selection function can be defined as

$$action : D \rightarrow Ac,$$

where Ac is the set of possible actions. Then through the function *action*, the agent attempts to prove the formula $Do(\alpha)$ from its database using the deduction rules ρ. If the agent succeeds proving $Do(\alpha)$, then α is chosen as the action to be performed. If the agent fails to prove $Do(\alpha)$, then it attempts to find an action that is consistent with the database and the rules, namely one that is not forbidden. Therefore it looks for an action such that $\neg Do(\alpha)$ cannot be derived from Δ given its deduction rules. If no consistent action is found, then the agent performs a *noop*, in order words, it chooses to perform *no* action.

Practical Reasoning Agents

Practical Reasoning Agents are also designed to reason about actions. Bratman [Bra90] defines this kind of reasoning in the following manner:

Definition 2.1.6 ***Practical reasoning*** *is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes [Bra90].*

Practical reasoning is considered to be composed of two stages: *deliberation* and *means-ends reasoning*. The former involves deciding *what* states of affairs to achieve, while the latter deals with deciding *how* to achieve these states of affairs. The state of affairs that an agent chooses and commits itself to achieve is reffered to the agent's **intentions**.

Deliberation is defined as follows. Let B be the agent's current beliefs; Bel, the set of all such beliefs; D, the agent's desires; Des, the set of all desires; I, the agent's intentions; and Int, the set of all intentions. Deliberation is then modeled via two functions: an option generation function *options* and a filtering function *filter* [Woo02]. These functions are defined as follows:

$$options : \mathscr{P}(Bel) \times \mathscr{P}(Int) \rightarrow \mathscr{P}(Des), and$$

$$filter : \mathscr{P}(Bel) \times \mathscr{P}(Des) \times \mathscr{P}(Int) \rightarrow \mathscr{P}(Int).$$

The *options* function produces a set of possible options or desires, based on the agent's current beliefs and intentions. In order to choose among competing options, the agent applies the *filter* function, which selects the 'best' option(s) for the agent to commit to. Additionally, the agent's beliefs are updated through the belief revision function *brf*:

$$brf : \mathscr{P}(Bel) \times Per \rightarrow \mathscr{P}(Bel)$$

Means-end reasoning is the process of deciding how to achieve an intention (i.e. an end) with the actions that the agent can perform (i.e. the available means). In the AI community, this kind of reasoning is known as *classical planning* or, simply, *planning*. A planning algorithm takes three parameters:

1. The *goal* or *intention*, a state of affairs that the agent wants to achieve or a state of affairs that the agent wants to maintain or avoid,
2. The current *state of the environment*, namely the agent's beliefs, and
3. The *actions* available to the agent.

With these inputs, the planner generates a *plan*: a sequence of actions that should allow the agent to reach the desired state of affairs. Therefore, the agent's means-end reasoning capability can be modeled as

$$plan : \mathscr{P}(Bel) \times \mathscr{P}(Int) \times \mathscr{P}(Ac) \rightarrow Plan.$$

In [GL87], the Procedural Reasoning System (PRS) was proposed as one of the first agent architecture to explicitly embody what we know today as the belief-desire-intention paradigm (BDI) [RG95, GPP+98]. In PRS, the agent does not perform planning from first principles, but it makes use of a library of plans that have been manually constructed by the agent's programmer. These plans are constituted by:

- A *goal* — the effect or postcondition of the plan;
- A *context* — the precondition of the plan; and
- A *body* — the actions to execute.

A PRS agent starts with a collection/library of plans and a set of beliefs about the world. Additionally, the agent usually has a top level goal. This goal is then pushed onto the agent's *intention stack*. The stack contains all goals that are yet to be achieved. The agent then searches the plan library for plans that have this goal as effect. Only some of these plans will have their preconditions satisfied in accordance to the agent's current beliefs. The set of plans that can (i) achieve the goal and (ii) have their preconditions satisfied are the set of *options* for the agent. Selection among the available *options* is the deliberation process previously described. In PRS, this deliberation is also achieved with plans. Since these plans are "plans about plans", they are usually known as *meta-level plans* or *meta-level reasoning* respectively. The meta-level plans change the focus of the agent's practical reasoning by changing its intention structures on-the-fly. That is, the meta-level plans choose which plan to execute. While the chosen plan is being executed, it may push additional goals onto the intention stack. In turn, raising a new requirement to find corresponding plans to achieve the additional goals. The process ends when the plans, required to achieve the goals in the intention stack, are individual actions that can be directly executed/calculated without posting additional subgoals. When a particular plan fails to achieve the goal, the agent simply chooses the next plan from its set of candidate plans for that goal.

Reactive Agents

As the naming suggests, reactive agents are agents that react directly to their perceptions of the environment. Brooks [Bro91], who is the first proponent of the theory of reactive agents, states that intelligent behavior can be generated without explicit representations and without explicit abstract reasoning of the kind that symbolic AI proposes. His claim is that intelligence is an *emergent* property of certain complex systems. His *Subsumption Architecture* [Bro90] builds on the assumption that an agent's decision making process is realized through task-accomplishing behaviors, by taking perceptual input

from the environment and mapping it to an action to perform. This mapping can be implemented as rules of the form *situation* $\rightarrow$ *action*.

The decision function *action* is realized through a set of behaviors, along with an *inhibition* relation holding between these behaviors. Let P be the set of all percepts and A be the set of all actions. A *behavior* is then a pair of the form (c, a), where $c \subseteq P$ is a set of percepts called the condition and $a \in A$ is an action. The behavior (c, a) fires when the environment is in state $s \in S$ if and only if the agent's perception function $see(s) \in c$. The set of all such rules can be defined as $Beh = \{(c, a) \mid c \subseteq P \text{ and } a \in A\}$. The previously mentioned *inhibition* relation $\prec$ is associated with the agent's set of behavioral rules $R \subseteq Beh$ as follows $\prec \subseteq R \times R$. This relation is a total ordering on R. Meaning that if $(b_1, b_2) \in \prec$, b_1 inhibits b_2 and b_1 therefore has higher priority than b_2.

The action selection process involves firstly computing the set of behaviors (c, a) that are triggered because their condition is met. Secondly, all triggered behaviors have to be checked to see which of them has the highest priority according to $\prec$. Finally, the action a of the selected behavior is returned as the action to perform. If no behavior is triggered, then a *noop* is returned instead to indicate that no action is to be performed.

Hybrid Agents

The main goal for hybrid agent systems is the combination of proactive and reactive behavior. These agent architectures are usually composed of at least two layers, one that deals with the reactive behavior and another that deals with the proactive behavior. Wooldridge [Woo02] characterizes such architectures in terms of the information and control flows within the layers as:

Horizontal layering all the software layers are connected to the sensory input and action output. Each layer produces suggestions of what action to perform.

Vertical layering sensory input and action output are each dealt with by at most one layer.

Horizontal layering offers the advantage that each type of behavior that the agent should have is just implemented in another layer and added to the action output. However it may require the use of a *mediator* function,

that decides which layer is in control at a given time, which ensures that the overall behavior of the agent is consistent. Vertical layering can be further subdivided into *one-* and *two-pass* architectures. In one-pass architectures, the control flows sequentially through each layer and the last layer produces the action output. In two-pass architectures, the control passes twice by every layer, once as the control passes towards the inner layers and a second time, as the flow returns towards the outermost layer. The action output is modified by every layer as the control flows.

Vertical layering has the advantage that there are only $n-1$ interfaces between n layers and each layer can only suggest m actions. This means that there are at most $m^2(n-1)$ interactions to be considered between layers. In contrast, when horizontal layering is used, then we can distinguish m^n such possible interactions. However, horizontal layering provides greater fault tolerance than vertical layering, since the failure of any given layer would have less impact on the agent's performance. Examples of hybrid agent architectures are the vertically layered two-pass architecture InterRRap by [MP93] and TouringMachines represent the horizontally layered architecture by [Fer92].

A natural question that arises is which type of agent is the best? The answer is as usual: there is no single golden bullet. However, guidance on how to choose a particular type of agent architecture for a given kind of task is presented in [Mül98].

2.1.3 Interaction

All but the most trivial systems contain a number of subsystems that must interact with one another in order to successfully carry out their tasks [Woo02]. Therefore in highly interactive systems like MAS, it is critical to understand the interactions that take place in the system as result of the actions taken by the agents.

The approach and methodology presented in this thesis adhere to the *benevolence assumption*[Woo02]: "agents in a system implicitly share a common goals, and thus that there is no potential for conflict." This assumption is made since it greatly simplifies the system designer's task. For the sake of completeness, we will present agent interactions in the more general case,

where each agent pursues its own benefit.

Let us look at the basic case of two agents i and j. Let each of them be *self-interested*, namely each of them has its own preferences and desires of how the world should be. Let $\Omega = \{\omega_1, \omega_2, \ldots\}$ be all the states over which agents have preferences over. These preferences can be represented by a **utility function**. Each agent possesses his own utility function. For every outcome, the function produces a real number as output, indicating how 'good' or 'desirable' that outcome is for the particular agent. In other words, the larger the output value of the utility function, the better that outcome is from the agent's point of view. Therefore, we can represent agent i's utility function as:

$$u_i : \Omega \rightarrow \mathbb{R}$$

Analagously, agent j's utility function is represented as:

$$u_j : \Omega \rightarrow \mathbb{R}$$

The utility function leads to a preference ordering over the outcomes/states. When there are two possible outcomes, e.g. ω_1 and ω_2, the function u_i allows agent i to rank these outcomes according to its preference.

In order to analyze the interactions that occur between i and j, we have to define the way their environment is modeled. Let us assume that both agents will simultaneously choose an action to perform. As a result of these actions an outcome $\omega \in \Omega$ is produced. This outcome depends directly on the combination of actions that take place. Also, let us assume that they have to perform an action, and they do not know what action the other agent is performing. If the agents could perform only two actions, say "C" for "cooperate" and "D" for "defect", then the way the environment behaves can be described by a function:

$$\tau : Ac_i \times Ac_j \rightarrow \Omega$$

where $Ac = \{C, D\}$, and $Ac_i, Ac_j \in Ac$ are the actions chosen by i and j respectively. Now that we have defined the notation, let us look at an example [Woo02]. Suppose we have the environment function:

$$\tau(D,D) = \omega_1, \quad \tau(D,C) = \omega_2, \quad \tau(C,D) = \omega_3, \quad \tau(C,C) = \omega_4. \qquad (2.1)$$

Suppose that the agents i and j have utility functions defined in the following manner:

$$\begin{array}{llll} u_i(\omega_1) = 1, & u_i(\omega_2) = 1, & u_i(\omega_3) = 4, & u_i(\omega_4) = 4, \\ u_j(\omega_1) = 1, & u_j(\omega_2) = 4, & u_j(\omega_3) = 1, & u_j(\omega_4) = 4. \end{array} \tag{2.2}$$

Based on the information from the utility function 2.2, we order the outcomes from the environment function 2.1 for agent i as follows:

$$u_i(\omega_4) \geq u_i(\omega_3) > u_i(\omega_2) \geq u_i(\omega_1)$$

This ordering indicates that agent i prefers all the outcomes in which it cooperates over all the outcomes in which it defects. Therefore agent i should cooperate, regardless of what action agent j chooses.
Although the decision is simple and clear cut in this example, it does exemplify how the agent's utility function assists the agent in choosing an action at any given point in time, and how these interactions relate to each other in a system. Most real world scenarios are more complicated and may require agents to engage in strategic thinking, i.e. considering the actions the other agent(s) may take. The strategies to choose under different kinds of circumstances are well studied in different areas like Game Theory, Economic Theory and others, thus, we will not address them here in detail.

2.1.4 Communication

As discussed previously, agents pursue the actions that reflect their preferences or that increase their utility. In some situations, agents will try to perform a certain action or achieve a goal for which they do not have access to all the resources necessary to perform such an action or achieve such a goal. For instance, an agent i needs to 'ask' agent j to perform a certain action on its behalf. Agent i may not assume that agent j will necessarily comply, as agent i is an autonomous agent. Therefore, the agents perform *communicative actions* in order to express themselves. These communicative actions that agents perform in order to try to influence other agents are known as **speech acts**.

Speech acts

In speech act theory [Sea69], communication is treated as action. It is based on the assumption that speech actions are performed by agents, just as any

other physical action, in the pursuit of their intentions. Seminal work on speech act theory includes the work by John Austin [Aus62]. He claims that certain language utterances—the speech acts—have the characteristics of actions, in the sense that they change the state of the world in a way analogous to physical actions. This does not mean that the world may have changed by the an utterance in an physically obvious fashion, but it still may be changed in a tangeable way. Austin uses a declaration of war as an example of such utterances. Austin also identifies a number of **performative verbs** which correspond to different types of speech acts. Examples of such verbs include *request*, *inform* and *promise*. Additionally, Austin distinguishes three different aspects of speech acts:

The locutionary act: The act of making an utterance,

The illocutionary act: The action performed in saying something, and

The perlocution: The effect of the act.

Further work by John Searle [Sea69] identifies properties that must hold for a speech act to be successful when performed between a *hearer* and a *speaker*. Such conditions include:

1. **Normal I/O conditions** the basic conditions to transmit and receive the speech act. For instance, that the hearer is able to hear the speech act.

2. **Preparatory conditions** the conditions that must hold so that the speaker can correctly choose the speech act.

3. **Sincerity conditions** the conditions that distinguish a sincere performance of, for example, the requested action in the case of the request speect act.

The further development of the speech act theory contributes and influences the development of various languages designed specifically for agent communications. Examples of these **Agent Communication Languages** (ACL) include the Knowledge Interchange Format (KIF) [Gen91, GF92], the Knowledge Query and Manipulation Language (KQML) [FFMM94, MLF95] and the Foundation for Intelligent Physical Agents' (FIPA) Agent Communication Language (FIPA ACL) [Fou02b, Fou02a, Fou02e]. The FIPA ACL is

closely related to KQML, but it attempts to address some of it shortcomings, such as the lack of clearly defined semantics.

In this thesis, agent communication and agent messaging is modeled using FIPA ACL. Therefore we briefly describe it below.

The FIPA Agent Communication Language

The Foundation for Intelligent Agents has defined standards for agent systems since 1995. One of the main targets of this initiative is the development of an Agent Communication Language (ACL). The FIPA ACL is similar to KQML [FFMM94, MLF95] as it defines a language for messages, defines formally performatives that determine how the messages are to be interpreted. It does not require any specific message content language (despite that FIPA later did specify a content language: the FIPA-SL Content Language [Fou02e]). To illustrate how these messages are structured, we show an example of a FIPA ACL message [Fou02e]:

```
(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B)
  :content
    " ((= (iota ?x (p ?x)) a)) "
  :language fipa-sl
  :in-reply-to query1)
```

This example could be read as follows: A informs B, in reply to $query1$ and described in $fipa - sl$ that, the x such that p is true of x is equal to a. A complete description of the FIPA ACL message parameters is presented in Table 2.1.

One difference between KQML and FIPA ACL is the collection of performatives that they provide. Another important difference between the languages is that FIPA ACL has a comprehensive formal semantics. The semantics of the FIPA ACL maps each ACL message to a formula of a language called *SL*. This formula defines a constraint that the sender of the message must satisfy in order to conform with the FIPA ACL standard. This constraint is referred to as the *feasibility condition*. The semantics also map each message to an SL-formula that expresses the *rational effect* of the action, in other words, the purpose of the message. As an example, we show the se-

Parameter	**Description**
performative	Denotes the type of the communicative act of the ACL message
sender	Denotes the identity of the sender of the message, that is, the name of the agent of the communicative act.
receiver	Denotes the identity of the intended recipients of the message.
reply-to	Indicates that subsequent messages in this conversation thread are to be directed to the agent named in the reply-to parameter, instead of to the agent named in the sender parameter.
content	Denotes the content of the message; equivalently denotes the object of the action. The meaning of the content of any ACL message is intended to be interpreted by the receiver of the message.
language	Denotes the language in which the content parameter is expressed.
encoding	Denotes the specific encoding of the content language expression.
ontology	Denotes the ontology(s) used to give a meaning to the symbols in the content expression.
protocol	Denotes the interaction protocol that the sending agent is employing with this ACL message.
conversation-id	Introduces an expression (a conversation identifier) which is used to identify the ongoing sequence of communicative acts that together form a conversation.
reply-with	Introduces an expression that will be used by the responding agent to identify this message
in-reply-to	Denotes an expression that references an earlier action to which this message is a reply.
reply-by	Denotes a time and/or date expression which indicates the latest time by which the sending agent would like to receive a reply.

Table 2.1: FIPA ACL Message Parameters

mantics of the *inform* speech act [Woo02]. For the complete specification and semantics of all the FIPA speech acts, we direct the reader to [Fou02b].

$$\langle i, inform\,(j, \psi)\rangle$$

feasibility precondition: $B_i\,\psi \wedge \neg B_i\,(Bif_j\,\psi \vee Uif_j\,\psi)$

rational effect: $B_j\,\psi$

where the $B_i\,\psi$ means that agent i believes ψ; $Bif_j\,\psi$ means that agent j has a definite opinion on whether ψ is true or not; and $Uif_i\,\psi$ means that agent i is uncertain about ψ.

Therefore, the feasibility precondition can be interpreted as:

1. Agent i believes ψ, and
2. It is not the case that agent i believes either
   (a) that agent j believes whether ψ is true or false, or
   (b) that agent j is uncertain about the truth of ψ.

If agent i's inform speech act succeeds, then agent j will believe ψ.

In addition to speech act, message structure and content language specifications, the FIPA agent communication specifications also include a library of commonly known interaction protocols such as Request Interaction Protocol [Fou02d] or the ContractNet Protocol [Fou02c].

2.2 Model-Driven Software Development

Model-Driven Development (MDD) is becoming more and more important for developing modern enterprise applications and software systems. MDD frameworks define a model-driven approach to software development in which visual modeling languages are used to integrate the huge diversity of technologies used in the development of software systems. The MDD paradigm provides us with a better way of addressing and solving interoperability issues in comparison to earlier non-modeling approaches [D'S01]. The current state of the art in MDD is highly influenced by the Object Management Group's ongoing standardization activities related to the Model Driven Architecture (MDA) [Obj03]. The MDA approach and its supporting standards allow the realization and integration of one model on multiple platform specific target models.

2.2.1 Models and Metamodels

Models consist of sets of elements that describe some physical or hypothetical reality [MKUW04]. Models are a way of communicating ideas without having to build the actual object or element in the real world. They allow us to concentrate on the important features of the topic at hand by abstracting away from the irrelevant elements and features. These different levels of abstraction provide the basis for MDA. In MDA, the models are transformed from one level of abstraction to another one to allow the adaptation of models to other levels of abstraction or other execution environments.

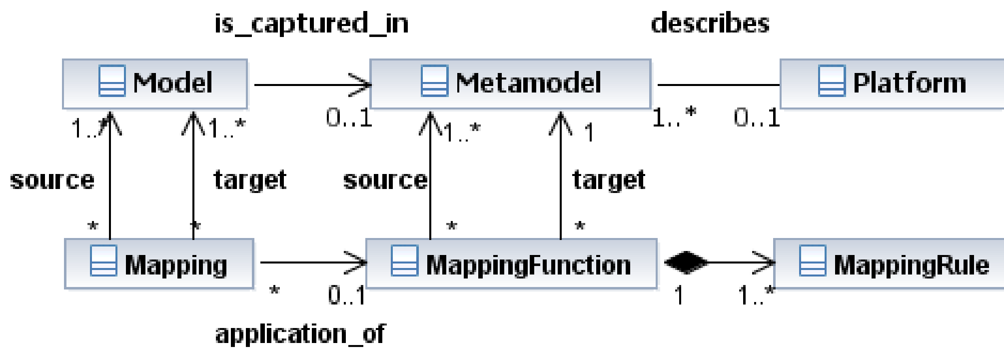


Figure 2.1: Relation among models, metamodels and platforms (based on [MKUW04])

In order to express a model, we use a metamodel. A **metamodel** is a model of a modeling language which defines the structure, semantics and constraints for a family of models. The term *meta* means “transcending” or “above”, emphasizing the fact that a metamodel describes a modeling language at a higher level of abstraction compared to the model itself. A metamodel can also describe the specification of a particular execution environment. In other words, it describes a **platform** in which compliant models can be executed. Using UML class diagram notation, Mellor et al. [MKUW04] summarize how models, metamodels and platforms are related. This summary is shown in the top half of Figure 2.1. To better understand the meaning of a metamodel, we discuss the difference between a metamodel and a model. While a metamodel is also a model, a metamodel has two main distinguishing characteristics:

1. A metamodel must capture the essential features and properties of the language that is being modeled, and
2. A metamodel must be part of a metamodel architecture.

In a metamodel architecture, all metamodels can be formulated via a single metamodel, the so-called **meta-metamodel**, that defines the key to metamodeling as it enables all modeling languages to be described in a unified way. System development is fundamentally based on the use of languages to capture and relate different aspects of the problem domain. This means that the languages can uniformly be managed and manipulated and thus tackle the problem of language diversity. Another benefit is the ability to define semantically rich languages that abstract from implementation specific technologies and instead focus on the problem domain at hand. Using metamodels, many different abstractions can be defined and combined to create new languages that are specifically tailored for a particular application domain. The **Meta Object Facility** (MOF) [Obj04] is the common foundation that provides the standard modeling and interchange constructs for defining metamodels and therefore can be considered a meta-metamodel.

2.2.2 Model Transformations

Models may have relations to other models, for example representing the same system at different levels of abstraction or its implementation on different execution environments. These relationships can be expressed in a **mapping** which is an application of a **mapping function** composed of **mapping rules**. Each rule describes how one or more elements in the source model should be transformed to the target model. The mapping takes a series of source models and produces target output models. Therefore, mappings are also referred to as **transformations**, given that through the execution of the mapping function source models are transformed into target models. Mapping functions and its rules are applied to models, but they are defined against the metamodels that capture these models. This entails that the mapping function is not specific to a model in particular, but to the family of models that comply to the corresponding metamodel. Model mappings are intended to be executed automatically so that models can be always synchronized. This is the focus of the *MOF Query, Views and Transformations* (QVT) language specification. QVT provides a standard specification of a

language suitable for querying and transforming models that are represented according to a MOF(-based) metamodel. The QVT specification defines various implementation approaches, such as QVT-Operational, in which rules are defined imperatively, or QVT-Relations, which allows the definition of rules declaratively.

2.2.3 The Abstraction Levels of the Model Driven Architecture

MDA defines three main abstraction levels of a system that support a business-driven approach to software development:

- The *Computation Independent Model* (CIM),
- The *Platform Independent Model* (PIM), and
- The *Platform Specific Model* (PSM).

Model transformations are applied to translate models from one abstraction level to the next as depicted in Figure 2.2. The CIM represents the desired system at a high level of abstraction. It is used to describe the context and requirements of a software system where only business or application domain concepts are used. For instance, a simplistic banking CIM, like the one depicted in Figure 2.2, may describe only the processes linked with accounts and loans and how these concepts relate to one another. When a PIM is used, the domain concepts are expressed in computational terms, providing details about the structure of the domain concepts. The latter is implemented using computational notions such as classes, attributes, types and methods, without adding information dependant on the deployment platform that is to be used. The PSM describes the detailed realization of the software system with respect to the chosen software technology platform(s).

The MDA initiative refers mainly to Object Oriented software development and has proven to be effective in these application domains. In this thesis, we propose a way to exploit the MDD ideas and techniques in Agent-Oriented Software Engineering (AOSE). This application of MDD brings the general benefit of improving (i) quality by allowing to reuse models and mappings between models and (ii) software maintainability by favoring a better consistency between models and code. In addition, our framework

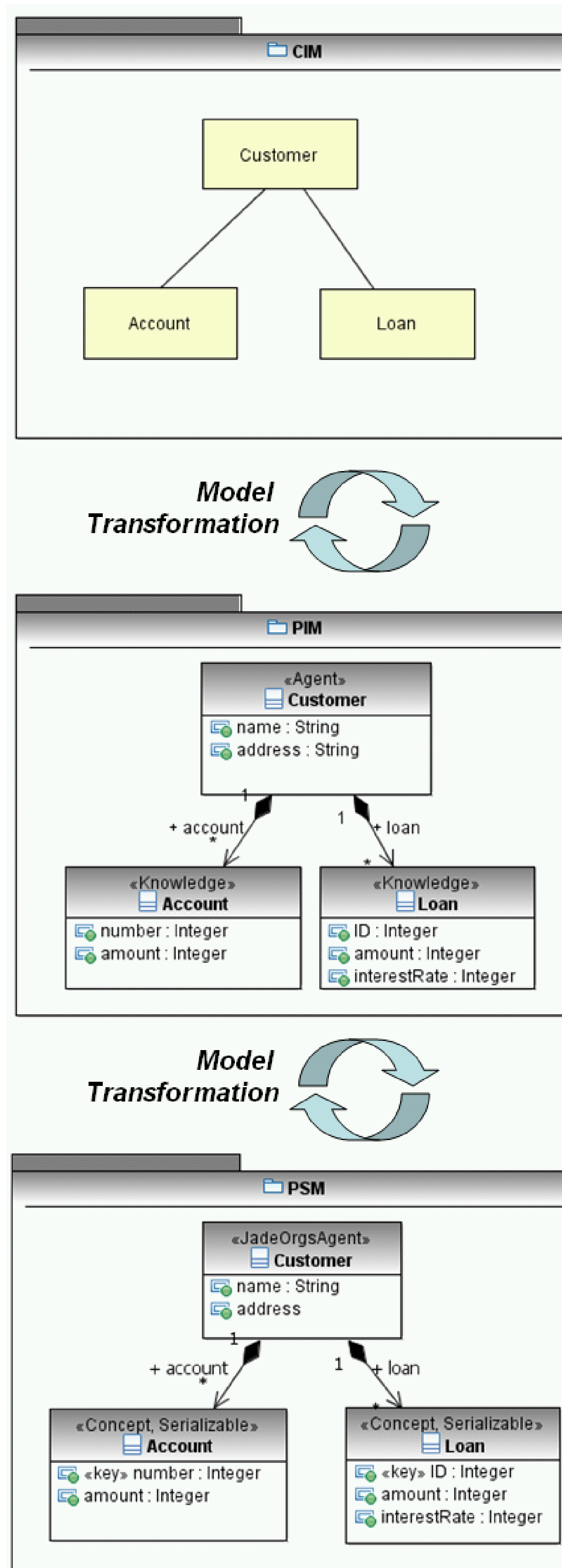


Figure 2.2: Example of a CIM, PIM and PSM

(i) establishes interoperability among various agent systems and other information technologies, and (ii) identifies a core metamodel that unifies the most common agent-oriented concepts to increase the efficiency in developing agent-based software applications.

Chapter 3

State of the Art in Agent-Oriented Software Engineering

This chapter is divided into four parts and we discuss approaches related to the Unified Modeling Language, agent metamodels, agent-oriented programming languages, and model-driven approaches in AOSE respectively.

3.1 Agents and the Unified Modeling Language

The Unified Modeling Language (UML) is the de jure standard language in industry for specifying and designing software systems. UML addresses the modeling of architecture and design aspects of software systems. It provides language constructs for describing software components, objects, data, interfaces, interactions, activities etc. However, it has a few shortcomings for modeling agent systems, in particular w.r.t. modeling agent communication. Agent oriented extensions to UML have tried to address these shortcomings and some features of these extensions have been merged into more recent versions of the UML.

Agent UML (AUML) [BMO00, BMO01] provides agent-based extensions to four UML representations: packages, templates, sequence diagrams and class diagrams.

In particular in AUML we can specify agent interaction protocols by mechanisms to define agent roles, agent lifelines, nested and interleaved protocols, and extended semantics for UML messages, such as the associated communicative act and whether messages are synchronous or not. Some of these features have been merged into the current UML 2.0. Furthermore, Bauer[Bau02] proposed a way to extend UML class diagrams to agent class diagrams.

Agent Modeling Language (AML) [TC05, CTCG04] is a semi-formal visual modeling language for specifying, modeling and documenting systems that incorporate features drawn from MAS theory. It is specified as an extension to UML 2.0 and its ultimate objective is to provide software engineers with a ready-to-use, complete and highly expressive modeling language suitable for the development of commercial software solutions based on multiagent technologies. Nevertheless, code generation facilities are not part of AML, but relies on UML compatible CASE tools for the generation of code.

3.2 MAS Metamodels and Frameworks

In the recent history of AOSE, a variety of approaches have emerged. Each approach focuses on particular issues or areas of interest. In this section we provide a short overview of the most relevant approaches, with particular focus on how they represent agent groups, organizations or teams and how the concept of a role, if present, is utilized.

Aalaadin [FG98] specifies one of the first developed metamodels for MAS. Based on the three main concepts *Agents*, *Groups* and *Roles*, it takes an organizational-driven (i.e. structural relationship between a set of agents) approach to build MAS. Agents are defined by their role they take on inside an organization and the capabilities they offer.

Tropos [BPG+04] is based on the idea of using the agent paradigm and related notions during all phases of the development of software process focusing on the concepts of actor and goal, and on early requirements. It proposes the use of AgentUML for detailed design and JACK Intelligent Agent as implementation platform. As already mentioned, the main concept in Tropos is the *Actor*, who is capable of executing *Plans* that fulfill a *Goal*, i.e. a *SoftGoal* or *HardGoal*, and uses *Resources*. The concept of an *Agent* inherits from *Actor* and may play *Roles*. The *Role* again inherits from the *Actor*.

ADELFE [BGPP02] specifies a methodology to develop adaptive MAS by concentrating on cooperative behavior. The main concept of ADELFE is the *Cooperative Agent* which has *Skills, Aptitudes, Characteristics,* and *Communications*. Furthermore, the agent observes *Cooperation Rules*.

Gaia [ZJW03, WJK00] has been designed to explicitly model the social aspects of open agent systems, with particular attention to the social goals, social tasks or organizational rules. The main concept of Gaia is the *AgentType* which is part of an *Organization*, and which collaborates with other *AgentTypes*, to provide *Services*. It plays several *Roles* like for instance 'Initiator' and 'Participant' that act in a *Communication* specifying a *Protocol*.

MOISE [HBSS00] proposes an organizational model based on three concepts: the roles, the organizational links and the groups. Under this model,

Roles constrain the individual behaviors of agents, *Organizational Links* regulate the social exchanges between the agents, and *Groups* constrain the layout of agents involved in strong interactions. MOISE+ [HSB02] is an extension to MOISE that attempts include the structural, functional and deontic aspects into MAS organizations.

INGENIAS [PGS03] provides both, a methodology and a set of tools to develop agent systems. INGENIAS distinguishes between five viewpoints: organization viewpoint, agent viewpoint, interaction viewpoint, tasks and goal viewpoint and environment viewpoint. The main concept of INGENIAS is the *Organization* that contains a *Workflow* and *Group*. A *Workflow* contains *Task* that affects and consumes *MetalEntity* and produces *Interaction*. A *Group* contains again a *Group* and belongs to *Application*, *Resource*, *Agent* and *Role*.

PASSI The Process for Agent Societies Specification and Implementation [Cos05a] is organized in three different domains. The solution domain covers the concepts *FIPA-Platform Agent*, *Service Description* and *FIPA-Platform Task*. The agency domain covers aspect like *Agent* that has a set of *Roles* that provide a *Service* and solve *Tasks* that includes a set of *Actions*. Furthermore, the *Role* is connected to *Communication* that works on *Agent Interaction Protocols* with a set of *Performatives*. The problem domain contains concepts like *Resource*, *Non-Functional Aspects* and *Requirements* that are connected with the *Agent*.

RICA (Role/Interaction/Communicative Action) specifies a metamodel [SO04] that integrates aspects of agent communication languages (ACL) and organizational models on three different layers: On the first layer, generic concepts of the system (e.g. agent, role and action types) are specified, the second includes social aspects like norms and institutions. The last layer specifies agent interactions via communication.

O-MaSE [DeL05, DGO10] is an organization-based extension to the Multiagent Systems Engineering (MaSE) [DeL91] framework. O-MaSE is composed of seven stages which involve the capture of goals, application of use cases, refinement of roles, creation of agent classes, construction of conversations, assembly of agent classes (agent/system architecture) and system

design (deployment). Once the system goals are defined, the organization structures and agent types are determined in accordance to how the system (sub-)goals are distributed. O-MaSE provides a Java-based development tool, called agentTool, to help users analyze, design and implement MAS.

MASSIVE [Lin01] is a pragmatic development model for multiagent systems based on a combination of standard software engineering techniques. It features a product model to describe the target system, a process model to construct the product model and an institutional framework that supports learning and reuse over project boundaries.

Unified MAS Metamodel Proposal A first attempt towards the development of a unified metamodel was described in [BCG+04]. This metamodel was developed by merging the metamodels of ADELFE, Gaia and PASSI and thus combines the strengths of each metamodel. For instance, the unified metamodel covers aspects like (i) cooperative behavior as described by the ADELFE metamodel, (ii) organizational behavior as specified by the Gaia metamodel, and (iii) FIPA-compliant communication structures as defined by the PASSI metamodel.

A more recent approach with respect to a unified metamodel was discussed during an AOSE Technical Forum Group meeting in Ljubljana. The attendees agreed on a smaller core compared to the first draft. In this metamodel, the *Agent* participates in a *Communication* and plays a *Role* that has the ability to solve particular *Tasks*. *Organizations* also refer to *Roles*. The *Cognitive Agent* is a specialization of *Agent* as it is represented in an *Environment*.

3.3 Agent-Oriented Programming Languages

Several agent-oriented programming languages already exist to implement MAS. In the following listing, we mainly focus on JACK[1] and JADE[2] as in our MDD approach they will be used for code generation purposes.

[1] http://www.agentsoftware.com.au/

[2] http://jade.tilab.com/

JACK Intelligent Agents [PH01] provides programming constructs and concepts for developing complex agent-oriented applications. It is based on the Beliefs, Desires and Intentions model, presented in Section 2.1.2 JACK already provides agent organization structures called **JACK Teams**. When designing systems in *Team mode*, there are no "single agents", but just complex and simple teams. The complex teams are those composed of other teams that play a **role** in the complex team, and simple teams are the actual agents that compose the leafs of the organizational hierarchy. One important trait is that all teams can and do execute their own behaviors/plans, which means that they orchestrate the actions of their sub-teams.

JADE (Java Agent DEvelopment Framework) [BPR99] provides programming concepts that simplify the development of MAS as it complies to the FIPA specification by providing the necessary communication infrastructure. In contrast to JACK, it intentionally leaves open the internal agent architecture and necessary concepts. Instead, JADE focuses on communication which is performed through message passing where each agent is equipped with an incoming message box. In addition, standard interaction protocols specified by FIPA such as FIPA-request or FIPA-query can be used as standard templates to build an agent conversation.

3.4 Model-Driven Development of MAS

Even if MDD is a relative young field, several efforts have been undertaken to bring MDD practices into the MAS development. We present a short of overview of some of this work as follows.

The Malaca Agent Model [AFV04] is an approach to agent-oriented design using MDA. The Malaca UML Profile provides the stereotypes and constraints necessary to create Malaca models on UML modeling tools. In this MDA approach, the transformation is realised from a TROPOS design model—as PIM—to a Malaca Model—as PSM.

Guessoum [Gue05] proposes a MDA-based approach for MAS to fill the gap between existing MAS tools and agent-oriented methodologies and metamodels, respectively. This approach is mainly based on the separation of the application logic (described in a PIM) from the underlying technology (described in a PSM). Based on Meta-DIMA, a MDA-based MAS development process defines the PIMs and PSMs by analysing the multiagent applications,

defines a library of metamodels by identifying the concepts used and designing the transformation rules to implement a metamodel from its description. A first step has been done by defining a PSM for the multiagent tool DIMA and PIMs from PASSI and Aalaadin/PASSI [BCG+04] metamodels.

An update to INGENIAS presented in [PGSF06] introduces the *INGENIAS Development Kit (IDK)*, as a way to provide MDD tools for MAS development. It presents the IDK MAS Model Editor, a graphical tool for MAS model creation, and a modular approach to adapt the editor and tools to new metamodels or target platforms. It also proposes that the model generation and metamodel development should be performed in parallel with periodic consistency checks to allow feedback from one activity to the other during the development.

The *Gaia2Jade Process* [MS06] shows how systems designed following the GAIA methodology, and its corresponding models, can be converted to JADE for deployment. It proposes that the implementation phase should be performed in four stages: communication protocol definition, activities refinement, JADE behaviour creation, and agent classes construction.

All the mentioned contributions make valuable points for the specification and modeling tasks in agent systems. However, interoperability between agent systems and especially among other technologies and domain-specific architectures is not addressed in these works. The *generic MAS metamodel* [BGPLHS05] or the *unified metamodel* [BCG+04] do address interoperability within agent systems, but with completely diverging approaches. On the one hand, the *generic metamodel* proposes a basic, but complete metamodel with respect to the concepts that define MAS, allowing the generation of systems in different agent platforms. On the other hand, the *unified metamodel* improves on ADELFE, Gaia and PASSI by combining their concepts, but also raises issues like the complexity of the methodology process necessary to develop a system and the construction of a tool chain for it. Even if both metamodels define the most important building blocks of a MAS, it is not really clear if executable code can be generated automatically as neither the internal behavior of an agent nor its external behavior—i.e. the agent interaction—is specified in an adequate manner.

Part II

A Model-Driven Approach for Organizations in Multiagent Systems

Chapter 4

Modeling Agent Organizations: A Methodology

In order to model and implement MAS, it is necessary to have a clear process that allows system architects to define the various components of the MAS. This chapter presents the methodology that we recommend for the creation of systems using PIM4Agents (Chapter 5) and JadeOrgs (Chapter 6). Both metamodels are defined using the the Eclipse Modeling Framework (EMF) [BSM+03] in order to take advantage of the transformation tools available for Eclipse and to fit our model-driven approach, depicted in Figure 4.1. The figure also displays how additional agent platforms, such as JADEX [PBL05], could be supported under this approach. However, it is beyond the scope of this thesis.

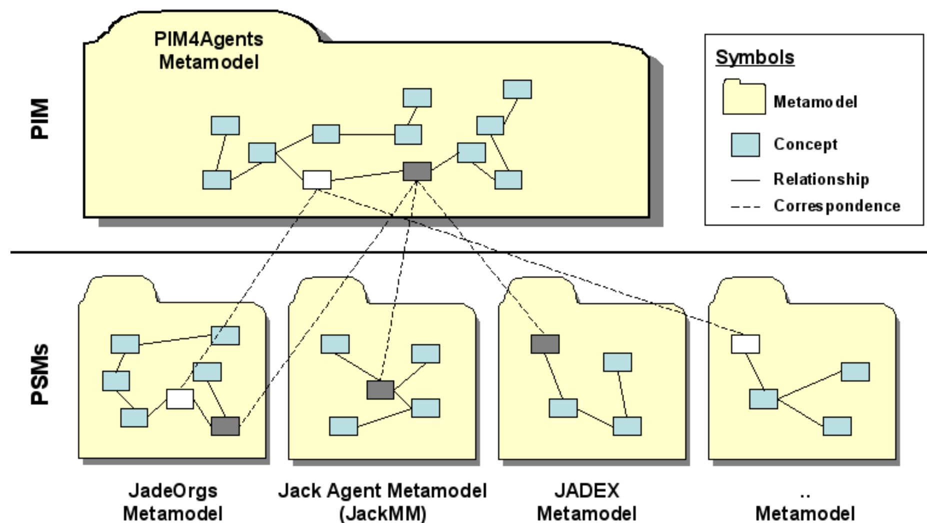


Figure 4.1: Overview of the model-driven approach with PIM4Agents

In this approach, a model defined with the PIM4Agents is transformed to a model described with the JadeOrgs metamodel, and finally the JadeOrgs model is serialized into Java source code. Transforming a PIM4Agents model to a PSM level model in JadeOrgs allows, if necessary, the refinement of the model with JADE/JadeOrgs concepts and avoids the need of introducing platform specific concepts in the PIM4Agents. The model to model transformations are specified in the Atlas Transformation Language (ATL) [ATL06] and the model to code serialization is implemented using MOFScript [SIN06].

From our point of view, the creation of a PIM4Agents model can be done in two different ways depending on the preferences of the designer and the characteristics of the system to be modeled. A PIM4Agents model can be

goal-driven or *behavior-driven*.

In a goal-driven model, the responsibilities of the different parties in the system are represented by goals that contribute to the overall system goals. On the behavior-driven model, the responsibilities and roles that the different parties play are defined by the behaviors that they possess. The goal-driven approach provides a higher abstraction level in the construction of the system as there is a clear detachment from **what** the system should do and **how** it is done. The behavior-driven approach encodes both the **what** and the **how** in the behavior's body, which might be more familiar for designers and developers that have less experience with agents and agent theory, and that have more experience with a more 'procedural' model for execution.

The methodology that we propose covers both ways to model a MAS with PIM4Agents. The biggest difference would be that the stages that have to do with goal modeling should be skipped for the creation of behavior-driven models. The methodology can be described by the following 8 stages:

1. Modeling of the abstract goals and their respective decompositions
2. Definition of the information model to be used in the system
3. Refinement of the abstract goals into concrete goals including the data dependencies between the goals
4. Modeling of the roles in the system with their corresponding responsibilities
5. Modeling of organizational structures and relations between roles
6. Modeling of the communication patterns through the definition of protocols
7. Definition of the agent plans
8. Establishment of the initial deployment configuration of the system

4.1 Modeling of the Abstract Goals

As an early requirements stage, we propose that goals should be defined in a very abstract fashion. At this point each goal is merely a label, in order to identify the desired state to be reached.

The abstract goals are linked by decomposition links. These links are of AND or OR kinds. An AND-decomposition link indicates that the supergoal is achieved when all of the subgoals are reached, while an OR-decomposition shows that the supergoal is achieved when any of the subgoals is achieved. This implies that subgoal of an OR-decomposition are actually specialization of the parent goal, in the sense that the OR-subgoal represents a state that fulfills all the conditions described in the parent.

The rationale behind this stage is to concentrate in what the system is supposed to do without dealing with other issues such as data dependencies or responsibilities. The definition of the abstract goals help define a clear picture of the issues the system should address and can be created in a iterative fashion. Each iteration should refine the decompositions and add 'leaves' to the decomposition tree.

4.2 Definition of the Information Model

The information model comprehends all the data types that are used in the MAS. These objects represent the type for the information that the system takes as input, the knowledge of the agents, the content of messages, the types used by services, the variables of goal events and the internal variables in plans.

The information model is usually created separately from the PIM4Agents model and it is also an Ecore model. Therefore the model can be created manually, by importing an XML schema or annotated Java code.

4.3 Refinement of the Abstract Goals into Concrete Goals

Once the information model and the abstract goal hierarchy is clear, the abstract goals must be refined into the concrete goal types:

Perform Goal indicates a procedural goal to execute a given action. Namely, the goal is to perform an action A, once the fact $done(A)$ is asserted, the goal has been achieved.

Achieve Goal denotes a declarative goal in which a desired state of affairs should be reached.

Query Goal describes a declarative goal that pursues a desired state of affairs in which a piece of information is available.

Maintenance Goal restricts the set of valid states of affairs. In other words, if its *maintenancecondition* is broken, the *Maintenance Goal* triggers a plan that should reestablish the broken condition.

By assigning a concrete goal types, the execution semantics for the goal are being set. Depending on the concrete goal type one or several conditions should be set to describe, for instance, the state that the goal represents or the situation under which the goal is no longer achievable. The conditions are expressed using variables which are bound when a goal event is created or as its bound plan is executed. For instance, the objective of a *Query Goal* is usually to find a binding for a given variable.

The decomposition tree for the concrete goals is usually analogous to the abstract goal tree. The tree can be further refined towards the leaves if more specific concrete goals are desired.

At this stage, the flow of information between the different goals should also be specified. For instance, a variable bound as the result of a given goal is passed as initial binding value for a variable in a goal that should be executed after.

4.4 Modeling of the Roles in the System

In order to define the roles in the system, it is necessary to consider the objectives of the system and how they related to one another with respect to the entity that should take care of each objective.

For example, if the system represented a bakery and one of the system objectives includes activities such as bake_cake, decorate_cake, attend_customers, each of these activities are performed by roles such as baker, decorator and clerk, respectively. It is important to note that the roles do not have necessarily to match one-to-one with the agent or organization types that the MAS will have. Agents and Organizations can play various roles depending on the desired system configuration. For instance, if we were modeling a bakery in a small town, it could be possible the an individual *Person* agent would have all roles assigned to it, while if we were modeling a big bakery in a shopping center, there could be several *Person* agents that only play one of the aforementioned roles.

If a goal-driven model is created, the roles will group concrete goals as responsibilities for each of the roles. In the case of a behavior-driven model, at this stage empty plans should be created as placeholders to represent the responsibilities and assigned to each role as a 'required behavior'.

4.5 Modeling of Organizational Structures and Relations between Roles

Once the roles have been determined, we can proceed to construct the organizational structures that the MAS will have. The organizations are determined by their required roles. The required roles specify the 'slots' that the member agents should fill in order for the organization to function. Also, it is important to assign the goals that will be considered as the organizational goal(s) for each organization.

In the case of a goal-driven model, the goal decomposition tree and the goals assigned to each role can provide a guide into determining the organization structure. For instance, if a given goal G is an organizational goal and its subgoals are assigned to certain roles, this is an indicator that perhaps these roles should be grouped together in an organization, given that a collaboration among the agents that will play these roles will be necessary to achieve G.

In addition to grouping the roles under the organization, it is necessary to set the cardinalities for each of these roles. The cardinalities specify the minimum and maximum amount of member agents necessary for the organization to achieve its organizational goals.

The way the roles will interact with one another is specified in a collaboration. The collaboration represents how the required roles from the application domain relate to the roles in the protocols, the actors. As will be described in the following section, protocols specify the generic communication patterns followed by the agents in the MAS. The collaboration represents how the protocols are applied in the social context of the given organization. In the collaboration, it is necessary to indicate again the cardinalities for each mapping, so that the actors are bound to the agents in a fashion that is in accordance to the protocol specification. The concrete messages and content type are also specified in the collaboration context. This is necessary because the messages used in the protocols, ACLMessages [Fou01], are abstract and

do not specify the content exchanged in them.

4.6 Modeling of the Communication Patterns

In order to specify the way agents and organizations interact with one another, protocols are defined to model the valid sequences of messages that can be exchanged. Each party that takes part in an interaction is known as an actor. Messages are always exchanged between 2 actors, and can be grouped into what is known as a scope. The scope specifies whether the group is, for instance, a sequence of messages. Alternative paths in the interaction are represented by subactors. A subactor represents the subset of agents bound to the actor that follow this alternative path.

The protocols are written in a generic fashion, given that they do not contain detailed information about the domain of application. As mentioned previously, the messages in the protocol do not specify the content used in the domain of application. This gives us the possibility to reuse protocols in different models.

In the case of a goal driven model, one way to link the goal and protocol execution, in an eventual plan, is to use a goal to represent the state of affairs in a given point of the protocol. For instance, a goal can represent the desired state before a given message is received. However, we think that in order to preserve the reuse of protocols only abstract goals should be use for this purpose. The reasoning behind this is that concrete goals and their conditions introduce specific domain knowledge reducing the reusability of the protocol, while the abstract goal is merely a label that represents the goal. Through the implementation relation between the abstract and concrete goals, the domain information can be 'reconnected' when reusing a protocol with abstract goals in another model.

4.7 Definition of the Detailed Behaviors of the Agents

The agent behaviors bring together all the other elements in the model. They represent how goals can be achieved, implement the projection of the protocols for each actor and use and specify how the information is used and manipulated by the agents.

The number of plans necessary for a given MAS can vary greatly, mostly depending on design preferences. One could have lots of small atomic plans that are linked together by others, or one can have big monolithic plans that do everything. We consider that the appropriate number lies somewhere in the middle.

Regardless of what the exact number of plans is, it is important to ensure that there is at least one plan that achieve every one of the agent/organizational goals and that the projection of every actor in a protocol is also implemented in a plan. As the body of the plans are designed these two tasks might be intertwined. For example, a plan that achieves a certain goal might require to perform the role of an actor of a protocol as part of the achievement. Depending on the design style, this may mean designing everything in the same plan or having the plan that achieves the goal invoke the plan that implements the behavior of the protocol actor.

The body of the plan is specified as a graph whose nodes are constructs such as sending a message or invoking a service, and whose edges indicate the flow of control from one activity to the next. In order to store and manipulate the information, variables can also be specified and are handled using scopes in a similar fashion to many programming languages: variables from container activities are available to their respective children activities. Therefore, we recommend that common variables are declared in the outermost common scope, so that unnecessary copying of variables is avoided.

Once the plan is fully modeled, we should make sure that plans are linked to the corresponding agent and organization types, so that the runtime instances can have the behaviors available at runtime.

4.8 Establishment of the Initial Deployment Configuration of the System

Once all the other stages of modeling have been completed, it becomes necessary to specify what the initial state of the system will be. At this stage we need to consider how many instances of each type of agent/organization type will be available, how organizations are originally established and, correspondingly, under which of the allowed roles will agents be bound to organizations.

Depending on the plans that were previously modeled and the applica-

tion domain of the MAS, the establishment of the organization could be completely dynamic, however it is often convenient that a initial configuration is provided. This initial configuration should respect the cardinalities that are specified for the roles in the collaborations and that agents are only bound under roles that they can fulfill.

Once the model is completed, the transformation steps should be able to generate the corresponding PSM model and source code. After the source code is generated, it should only be necessary to implement whatever interfaces where produced for specific platform dependent code.

The methodology presented in this chapter is only a recommendation on how to produce a model in a structured fashion. Depending on the application or development style, iterations and refinements on the different stages may be applied. While using a methodology cannot guarantee the success in the design or implementation of a MAS, this structured process will help to show possible issues with respect to the system requirements at an early stage. This issues can then be addressed in the models avoiding a full code implementation based on false understanding of the requirements.

For illustration purposes, an example that follows the methodology steps can be found in Section 5.10.

Chapter 5

The Platform Independent Metamodel for Agents: PIM4Agents

One challenge in defining a platform-independent metamodel is to decide which concepts to include and which to abstract from the target execution platforms that support the architectural style of agent-based systems. Chapter 3 discusses several metamodels for MAS, however, the only concept most metamodels have in common is the concept of an *agent*. Some of them also focus on *role* and *communication/interaction*. From this discussion, it is important to note that finding platform-independent concepts for MAS is a complex and non-trivial task. From our point of view, the minimal definition for an agent is an entity that is capable of acting in the environment. It acts in an autonomous manner, i.e. the agent has control over its own behavior and reacts in response to internal and external stimuli. Another property is the ability to communicate with other agents. Additionally, the agent is capable of perceiving its environment. In the following chapter, we present an overview of the platform-independent concepts and attributes necessary for designing agents in an adequate manner. To facilitate the presentation of our platform-independent metamodel for MAS called *PIM4Agents*, it is structured into several aspects. Each of them focuses on a specific viewpoint of a MAS.

1. *Multiagent System view* contains the main building blocks of a MAS and thus includes concepts like Agent, Capability, Interaction, Role, or Environment.

2. *Agent view* describes single autonomous entities, the capabilities they have to solve tasks and the roles they play within the MAS.

3. *Organization view* describes how single autonomous entities cooperate within the MAS and how complex organizational structures can be defined.

4. *Goal view* represents the agent's and organization's objectives in the system and how they are related to one another.

5. *Role view* covers the abstract representations of functional positions of autonomous entities within an organization or other social relationship.

6. *Behavioral view* describes how plans are composed by complex control structures and simple atomic tasks like sending a message and how information flows between those constructs.

7. *Interaction view* describes how the communication in the form of interaction protocols takes place between autonomous entities or organizations.

8. *Information Model view* represents the ontology related to the domain of application and contains any kind of resource that is dynamically created, shared, or used by the agents or organizations, respectively.

9. *Deployment view* presents how the agent and organization instances are initially configured in the system.

Grouping modeling concepts in this manner allows metamodel evolution by (i) adding new modeling concepts in the existing aspects, (ii) extending existing modeling concepts in them, or (iii) defining new modeling concepts for describing additional characteristics of agent systems. In the following sections, we explore the core aspects of the PIM4Agents in more detail. Each aspect is defined by a submetamodel focusing on the related concepts that altogether form the PIM4Agents metamodel.

5.1 Multiagent System Viewpoint

The Multiagent System Viewpoint is centered around the *MultiagentSystem* concept. The metamodel for this viewpoint is presented in Figure 5.1. As seen in the metamodel, *MultiagentSystem* is the general container of all the elements that compose a MAS such as *Agents*, *Goals*, *AgentInstances*, *Capabilities*, *Interactions*, *Roles*, *Behaviors*, and *Information Models*. The features of the contained elements are described in detail in the following sections.

5.2 Agent Viewpoint

The Agent Viewpoint is centered on the concept of an *Agent*, the autonomous entity capable of acting in the environment, and the basic building block of the MAS. Figure 5.2 depicts the metamodel that corresponds to this view.

In order to be able to take action and achieve its goals, the *Agent* must possess a series of *Behaviors* that are either assigned by the system designer or acquired at runtime. For additional modularity, *Behaviors* that have certain affinities or common purpose can be grouped into *Capabilities*.

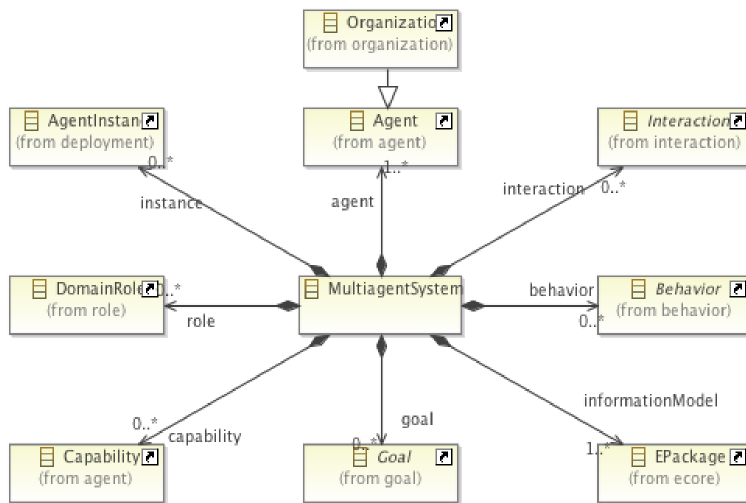


Figure 5.1: The Multiagent System View of the PIM4Agents.

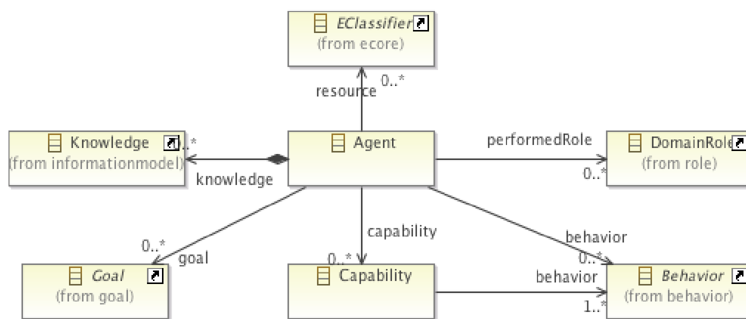


Figure 5.2: The metamodel reflecting the agent aspect of the PIM4Agents.

Through its *Behaviors*, the *Agent* utilizes a set of *Resources* from its surrounding environment. These *Resources* may include information sources to which the Agent has access and are represented using the *EClassifier* class from Ecore, bootstrapping on the meta-metamodel used to define PIM4Agents. The beliefs of the *Agent* are represented through a *Knowledge* class. These beliefs are accessed and modefied as the *Agent* executes its behaviors.

Furthermore, the *Agent* has a social aspect and can perform particular *DomainRoles* that are aligned with its own internal *Goals*. The *DomainRoles* specify the responsibilities that the *Agent* commits to fulfill in a given social context. The *DomainRole* can also provide the *Agent* with additional *Behaviors* that would facilitate the performance of the given *DomainRole*.

Even if most of the agent modeling approaches presented in Section 3.1 do not address the modeling of runtime instances, we consider that it is useful to model the initial configuration of the instances in the MAS through the *AgentInstances* concept. The *AgentInstance* is classified by the *Agent* class. This allows the system designer to model particular instances that share common features specified in the *Agent* type.

5.3 Organization Viewpoint

The Organization Viewpoint presents the concepts that represent the social aspect of the MAS. It allows the system designer to model how agents and organizations relate to one another and under what particular circumstances these relationships occur. Figure 5.3 depicts the metamodel for this aspect. It includes the concepts *Organization, Goal, DomainRole, Collaboration, DomainRoleBinding, Interaction,* and *ActorBinding*.

The *Organization* defines a social structure for *Agents* and other *Organizations*. It is formed to regulate, support, and facilitate the interaction among its members. Therefore, it provides a social context under which the *Agents* interact. At the same time, the *Organization* is a specialization the *Agent* type. This enables the *Organization* to execute its own *Behaviors* in order to achieve organizational *Goals* (cf. Section 5.4). In most cases, the responsibilities of the *DomainRoles* performed by the member agents will be subgoals of the organizational *Goals*, which will be called upon by the *Organization* as it executes the behaviors that pursue the achievement of such organizational *Goals*.

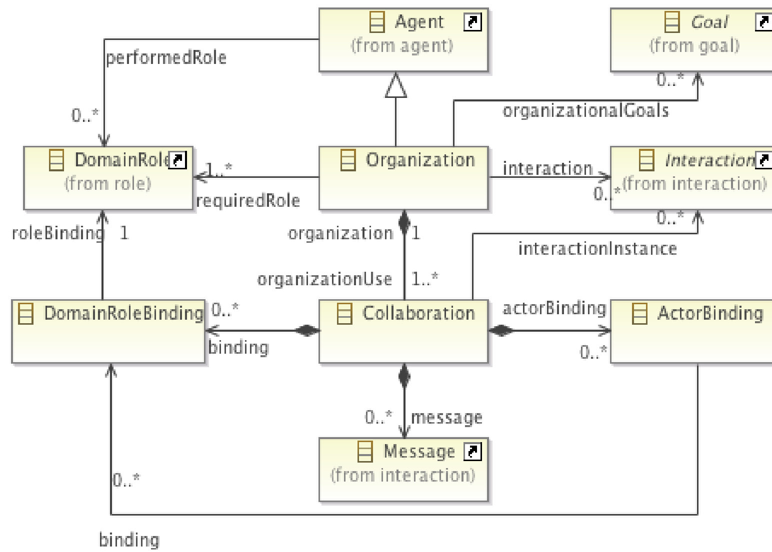


Figure 5.3: The metamodel reflecting the organizational aspect of the PIM4Agents.

A *Collaboration* is utilized to establish how the domain specific roles in the *Organization*, the *DomainRoles*, relate to the domain independent roles in an *Interaction*, the *Actors* (cf. Section 5.7). Through the use of the *DomainRoleBindings* and *ActorBindings* in the *Collaboration*, the *Interaction* specifies which *DomainRoles* interact with each other under the chosen communication pattern. The *Collaboration* also refers to the *Messages*. A *Message* is the domain specific counterpart to the domain independent *ACLMessage* that is part of a *Protocol* (cf. Section 5.7). The *Message* determines the content that will be transmitted when the *Speechact* specified in the corresponding *ACLMessage* is performed in the context specified by the *Collaboration*.

For example, given a domain of application with a **Bank** *Organization* and a **Customer** *DomainRole* that requests the creation of a new account to a **Bank Clerk** *DomainRole*, they would interact using the FIPA-Request protocol [Fou02d]. The **Customer** would be linked through the bindings to the **Requester** *Actor*, and the **Bank Clerk**, correspondingly, would be linked to the **Responder** *Actor*. A *Message AccountCreation* would then be linked to the *ACLMessage* that specifies the `request` performative and indicate that the content would be of type **AccountInfo**.

5.4 Goal Viewpoint

The goal viewpoint (cf. Figure 5.4) represents the goals that are assumed by *Agents* and *Organizations* in the MAS. A *Goal* describes a state of affairs that the agent has committed itself to bring about through its actions. When an agent assumes a goal, a given *Behavior* or *Plan* is initialized or triggered to take action towards the realization of this goal. Therefore a *Goal* is also a specialization of an abstract *Event* type that represents events which *Agents* must react to or handle. As an *Event*, it contains *variables* which are bound through the process of achieving the *Goal*. In addition, *Goals* can be related to one another through *conflictingGoals*, in which two goals cannot be achieved at the same time, or decomposition relations.

If an *ANDDecompositionLink* joins a subgoal with its parent, this implies that the parent goal is only achieved once all the subgoals have been achieved. This link prescribes a logical *AND* on the conditions specified in the subgoals $s_1, \ldots, s_n$, which entails the achievement of the parent goal p , as in a Horn clause [Hor51]: $(s_1 \wedge s_2 \wedge \ldots \wedge s_n) \Rightarrow p$.

If subgoals are joined with an *ORDecompositionLink*, the parent goal is achieved when any of the children goals are achieved. Correspondingly, this link describes a logical *OR* performed on the subgoals to determine if the parent goal is achieved. This usually entails that each subgoal is a special case of the parent goal. For example, if the parent goal is **ProcessPayment**, then subgoals **ProcessCCPayment** and **ProcessCheckPayment** would be linked to its parent through an *ORDecompositionLink*. Since credit card and checks are both acceptable methods of payment, the **ProcessPayment** goal is achieved.

Furthermore, the *Goals* are classified as *AbstractGoals* and *ConcreteGoals*. An *AbstractGoal* is meant to be used when the concrete conditions under which the goal is achieved are not clear. Such goals would be particularly used in the early requirements phase of the design of the MAS, since the concrete details of the goals and the application domain may not be known in detail.

The *ConcreteGoals* are used when the detailed information about the goal *Conditions* is known, such as in the detailed design of the MAS. Based on the Unified Goal Framework [vRDW08], the *ConcreteGoal* is further specialized into 4 types:

PerformGoal indicates a procedural goal to execute a given action. Namely,

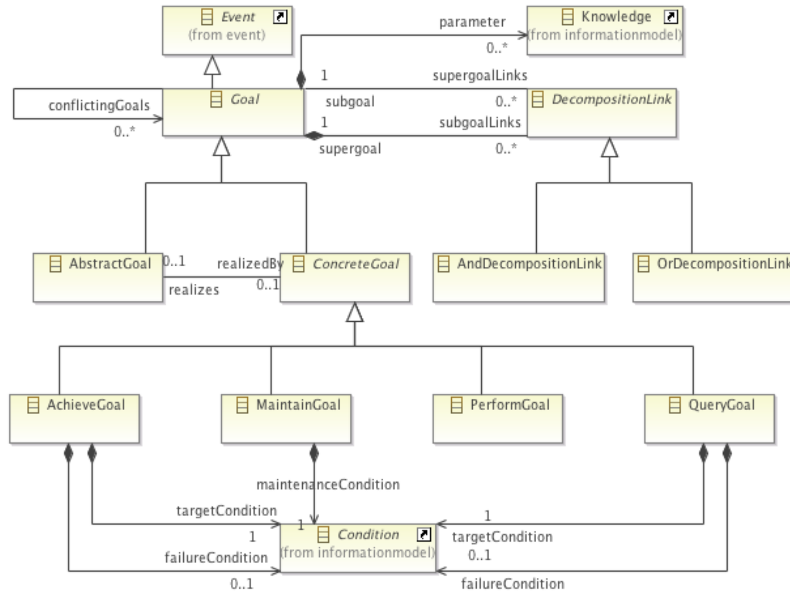


Figure 5.4: The metamodel reflecting the goal aspect of the PIM4Agents.

the goal is to perform an action A, once the fact $done(A)$ is asserted, the goal has been achieved.

AchieveGoal denotes a declarative goal in which a desired state of affairs should be reached.

QueryGoal describes a declarative goal that pursues a desired state of affairs in which a piece of information is available.

MaintenanceGoal restricts the set of valid states of affairs. In other words, if the *maintenanceCondition* is broken, the *MaintenanceGoal* triggers a plan that should reestablish the broken condition.

In the case of the *AchieveGoal* and QueryGoal, the *targetCondition* describes the target state(s) of affairs, while the optional *failureCondition* describes the state(s) where the goal can no longer be achieved. As mentioned in its definition, *MaintenanceGoal* is described through a *maintenanceCondition* which describes the state(s) of affairs that should be preserved.

The goal types and the decomposition links which join them allow the creation of a goal decomposition tree that is used to distribute the responsi-

bilities among the members of the organization in order to achieve organizational goals.

5.5 Role Viewpoint

A *Role* is an abstraction of the social behavior of the *Agent* in a given social context, usually an *Organization*. The *Role* specifies the responsibilities and the functional position of the *Agent* in that social context. It defines what the "role player" is expected to do. It refers to (i) a set of *Goals* that specify the responsibilities to be fulfilled, (ii) a set of *Capabilities* that define the *Behaviors* that the "role player" is required to have or will be provided with when the *Role* is granted, and (iii) a set of resources which are required by the *Role* or provided in order to play the *Role*.

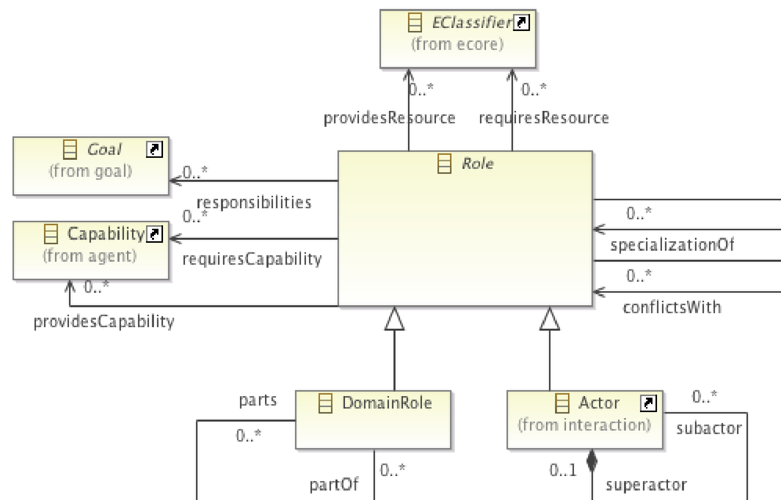


Figure 5.5: The metamodel reflecting the role aspect of the PIM4Agents.

The *Role* aspect covers the *Role*, its specializations and how they can be related to each other. The metamodel for this aspect is depicted in Figure 5.5. It includes the concepts *Role*, *Actor*, and *DomainRole*, as well as *Goal* (form the goal aspect), *Capability* (from the agent aspect), and Ecore's *EClassifier* to represent resources.

The main *Role* concept is abstract and its two main specializations *DomainRole* and *Actor* describe the agent positions in the domain-specific *Organizations* and the domain-independent *Interactions* respectively. *Roles* can

be defined in a hierarchy through the *specializationOf* relation. For instance. the **CEO** *Role* is a specialization of the **Manager** *Role*.

Additionally, *DomainRoles* can be composed with the *parts* relation, aggregating the features of all the *parts* into the *partsOf DomainRole*. An example of this aggregation is an **OfficeAssistant** *DomainRole* that is composed of **Receptionist** and **Typist** *DomainRoles*.

On the interaction side, *Actors* can be partitioned into *subactors*. The idea behind this partitioning is to represent the different paths available during the execution of an interaction or protocol and how the different instances bound to each actor are grouped into different subgroups depending on the trace of the execution of the interaction. The *Actor* concept is discussed in more detail in Section 5.7.

Besides discussing the features and properties that define the role type, we should address the assignment of "role players" to these roles. The roles can be assigned endogenously through self-organization or exogenously by the system designer [OPF03]. The approach taken in PIM4Agents leans towards the latter approach, by defining the static assignment of roles at design time, while JadeOrgs (cf. Chapter 6) supports both a static initial role assignment as well as dynamic role assignment at run time. The language constructs used to model the static role assignment will be presented in Section 5.9.

5.6 Behavior Viewpoint

The behavior aspect describes (i) how Plans are composed by complex control structures and simple atomic tasks like sending a message and (ii) how information flows between those constructs. A partial metamodel of the behavior aspect is depicted in Figure 5.6 and includes the concepts *Behavior*, *Plan*, *Flow*, *ControlFlow*, *InformationFlow*, *Activity*, *StructuredActivity*, and *Task*.

A *Behavior* represents the super class connecting the agent aspect with the behavior aspect, where a *Plan* can be considered a specialization of the abstract *Behavior* to specify an agent's internal processes. An *Activity* contains a set of *Flows* and *Activities*. *Activities* are classified into *StructuredActivities* and *Tasks*. *StructuredActivities* are composed of other *Activities* and present complex control structures, while *Tasks* are atomic activities that perform simple actions. It is important to note that the *Plan* is also a specialization of *StructuredActivity* that possesses the *preconditionObject* and

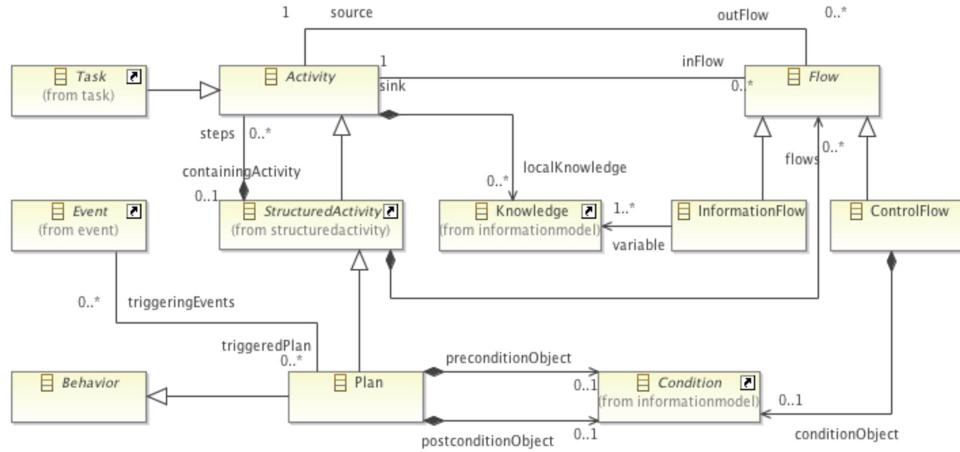


Figure 5.6: The partial metamodel reflecting the behavior aspect of the PIM4Agents.

postconditionObject associations to represent the circumstances under which the plan can execute and the effects of its execution, respectively. In addition, the *Plan* may be triggered by an *Event*, such as one of the *Goals* shown in Section 5.4.

The *Activities* are linked to each other via *Flows* which are either of the type *InformationFlow* or *ControlFlow*. *InformationFlows* are only necessary when an activity requires access to a variable and this variable is not within the scope of the activity, in other words it is not declared in any of the *StructuredActivities* that contains the *Activity* in question. *ControlFlows* are the links that express how the controls changes from one activity to the next and may contain a guard, the *conditionObject*, that determines if the following activity may be executed.

StructuredActivities and *Tasks* are specialized into various types, as depicted in Figures 5.7 and 5.8, respectively. We will briefly review what each of these specializations represent.

The hierarchy of *StructuredActivities* include the following:

Sequence represents a set of *Activities* under a total order.

Loop indicates a set of *Activities* whose execution is repeated until a certain condition is met.

Split represents a fork in the flow of control. The different paths can be

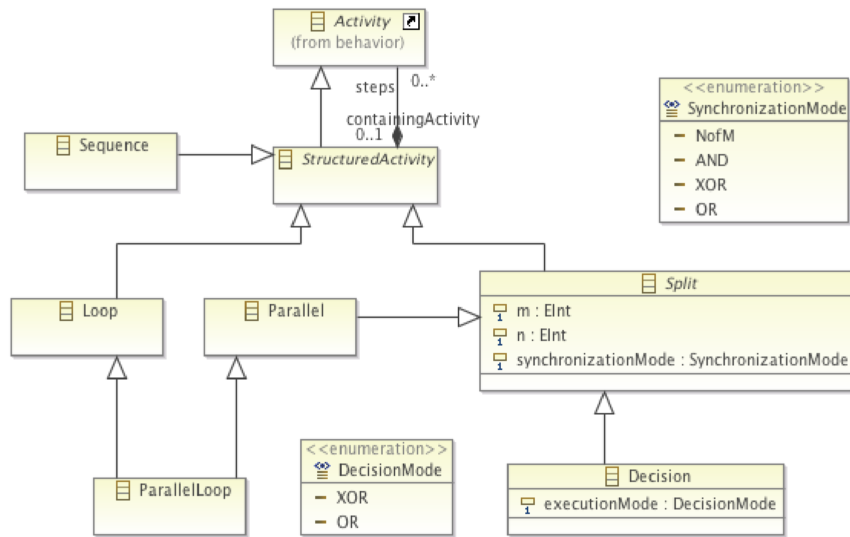


Figure 5.7: The hierarchy of StructuredActivities.

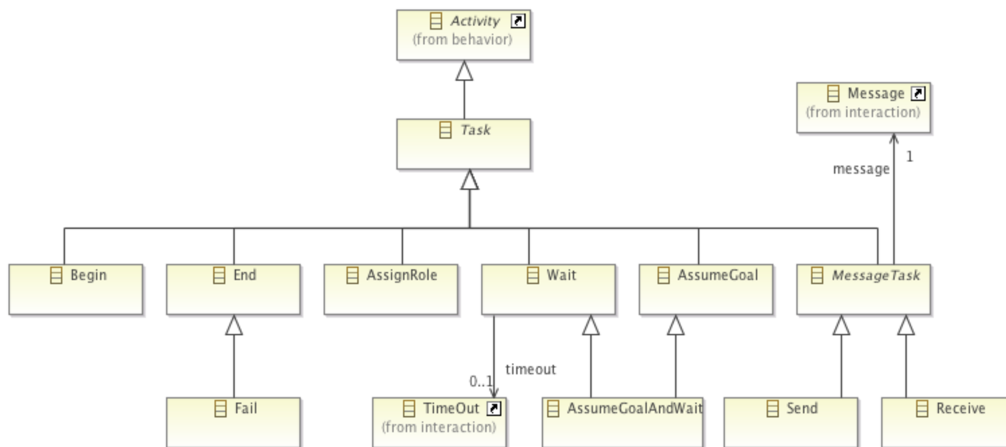


Figure 5.8: The hierarchy of Tasks.

synchronized in accordance to the *SynchronizationMode* enumeration: after all paths terminate (*AND*), after one or more paths terminate (*OR*), after exactly one path terminates (*XOR*) or after a determined number of paths have terminated.

Decision is a specialization of *Split* that is restricted to only two *Execution-Modes*: one or more paths may execute (*OR*) or exactly one path may execute (*XOR*). The execution of each path depends on the evaluation of the guard in the *ControlFlow* that links the *Begin* with the first *Activity* of each execution path.

Parallel is a *Split* in which each execution path is executed in parallel to the others.

ParallelLoop is a combination of *Parallel* and *Loop*. It represents the parallel execution of several instances of the process described by the *Activities* contained. The number of parallel threads executed is only known at run time. A common use case of this construct is to handle communication with all the agent instances bound to a given *Actor*.

As previously mentioned, to represent atomic *Activities* we use the concept of a *Task*. *Tasks* include actions such as sending and receiving messages or assuming a goal. The *Tasks* presented in Figure 5.8 are described as follows:

Begin represents the starting point of an *Activity*.

End represents the expected/successful end of an *Activity*.

Fail represents an exceptional end or the failure to execute the *Activity*.

AssignRole asserts the fact that a given agent is playing a certain *Role*. It is used to keep track of the partitioning of *Actors* into *subactors*.

Wait causes the process to wait for a given period of time and may be linked to a *TimeOut* specified in an *Protocol*.

AssumeGoal represents the addition of a given *Goal* to the set of currently pursued *Goals* prior to continuing the execution.

AssumeGoalAndWait is similar to *AssumeGoal* but causes the execution to wait for the achievement or failure of the goal event before resuming the execution. A timeout period may also specified to avoid waiting indefinitely.

MessageTask is an abstract class that classifies all tasks that deal with messages and, therefore, refers to the message to be sent or received.

Send is a specialization of *MessageTask* that represents the action of sending a *Message*.

Receive is a specialization of *MessageTask* that represents the action of receiving a *Message*.

Although the number of *Activities* indicates that specifying a *Behavior* is a complex task, the example in Section 5.10 will show that with the appropriate tool support this complexity can be managed.

5.7 Interaction Viewpoint

Figure 5.9 depicts the partial interaction aspect of the PIM4Agents. The ability to communicate is one of the core characteristics of agents and groups of agents in MAS. In the PIM4Agents, a *Protocol* refers (i) to a set of *Actors* (e.g. **Buyer** and **Seller**) that interact within the *Protocol* and (ii) to a set of *MessageFlows* that specify how the exchange of messages occurs. The *Actor* can again refer to a set of *Actors* as *subactors*, meaning that the set of agents performing the *superactor* is partitioned into the *subactor* sets. In general, the *subactors* are determined at design time, but filled with the particular instances that perform this kind of *Role* at run time.

A good example for distinguishing between *superactor* and *subactor* is the Contract Net Protocol (CNP) [DS83] . In the CNP, after the proposals have been collected, the **Initiator** may send either an `accept-proposal` or a `reject-proposal` to the **Participant**. The decision about which message is sent depends on if a **Participant** is considered as the best bidder with respect to some pre-defined criterion. If this is the case, this **Participant** gets an `accept-proposal`, otherwise it receives a `reject-proposal`. This implicit distinction between best bidder and remaining bidders can be made explicitly in the PIM4Agents. The **Participant** (as a *superactor*) would have

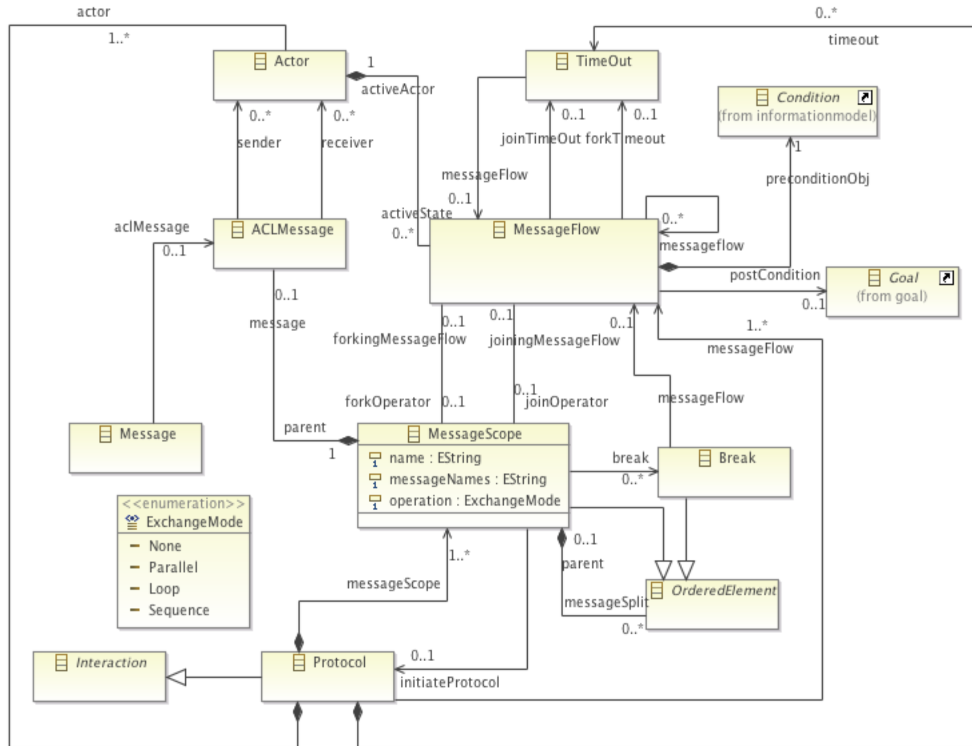


Figure 5.9: The partial metamodel reflecting the partial interaction aspect of the PIM4Agents.

two *subactors*, i.e. **BestBidder** and **RemainingBidders**, that are filled at run time.

The *MessageFlows* represent the states in the execution of the protocol linked to a set of *Actors* that are active in the current state, i.e. those *Roles* that send the specified *ACLMessages*. Through the *messageflow* relation, a *MessageFlow* can refer to another *MessageFlow* to indicate the transition from an *superactor* to a *subactor*.

Furthermore, the *MessageFlow* is associated with the *MessageScope* as *join* or *forkOperator*. An incoming transition is a *joinOperator*, while an outgoing one is a *forkOperator*. Another way to trigger a transition is through a *TimeOut*. The *TimeOut* is a time constraint for some section of the *Protocol*. In a way analogous to the relation with *MessageScope*, *TimeOuts* are either *fork* or *joinTimeouts* depending on whether the transition is outgoing

or incoming respectively.

As a state in the dialogue, the *MessageFlow* can also specify under what conditions a transition to the state can take place and the effects of that state. The *preconditions* are modeled with a *Condition* object, while the *postCondition* is modeled with a *Goal*. Using a *Goal* as *postcondition* links the interaction view with the behavior view. It permits the system designer to model how a state in a *Protocol* can cause a certain *Plan* to be triggered via the *Goal*.

A *MessageScope* defines the communication transitions between the states of the *Protocol*, contains the *ACLMessages*, and determines the order in which they are exchanged. The *MessageScope* can contain *OrderedElements* in its *messageSplit* relation. The *OrderedElements* are specialized in *MessageScopes* or *Breaks*. The *Break* specifies if a given *Trigger*, such as a expired *TimeOut*, causes the message exchange to be transition to be cancelled. Whether or not a MessageScope contains only one *ACLMessage* or a set of *OrderedElements* is determined by the *ExchangeModes* as follows:

None indicates that only one *ACLMessage* is exchanged in this transition.

Sequence prescribes that the *ACLMessages* are exchanged in a sequence.

Parallel the *ACLMessages* in this transition may be exchanged in in parallel. In other words, the order of the exchanged messages is not determined.

Loop the *ACLMessages* may be exchanged again and again until the *trigger* of a *Break* holds or the following *MessageFlow*'s *precondition* evaluates to true.

Finally, *Messages* are the domain specific counterpart to *ACLMessages*. The *Message* specifies the message content used for a particular *ACLMessage* when the *Protocol* is used in a certain context, the *Collaboration* (cf. Section 5.3).

5.8 Information Model Viewpoint

In order to communicate with one another and to have a common domain of discourse, it proves necessary to define what common concepts the *Agents* will be discussing. The information model aspect represents the domain

ontology for the MAS. It contains all concepts that classify objects created, shared, or used by the *Agents* and *Organizations*.

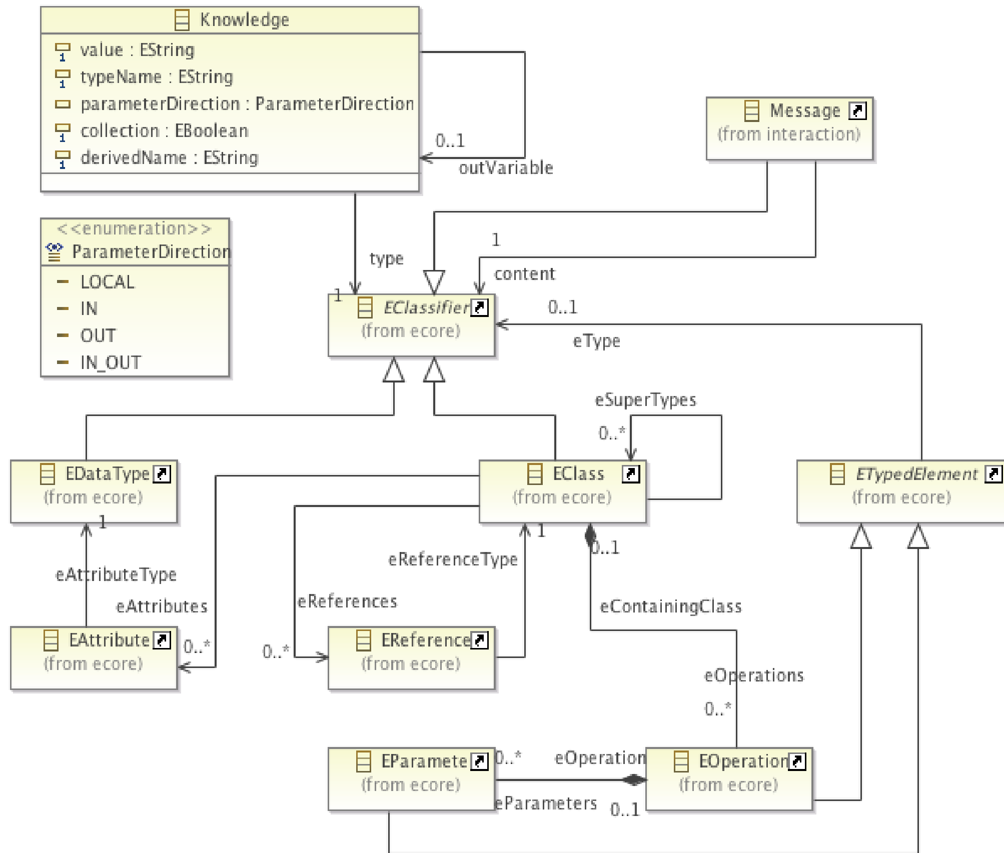


Figure 5.10: The partial metamodel reflecting the information model view of the PIM4Agents.

The *Agent's* beliefs, the content of the messages exchanged and the local variables and parameters inside the *Activities* are typed by the concepts defined in this viewpoint. Depicted in Figure 5.10, the *Knowledge* concept is applied to represent all these references to the ontology classes. When used as a parameter for an *Activity*, the *Knowledge*'s *parameterDirection* indicates the direction of the parameter. If it were an incoming parameter, the *value* property indicates how the variable is initialized upon entering the *Activity*. If it were an outgoing parameter, the *outVariable* relation could indicate to where the value of the parameter would be stored at the end of the *Activity*.

For the definition of the ontology classes themselves, we make use of the Ecore metamodel itself [BSM+03]. Using Ecore as the ontology representation allows us to take advantage of the ample tool support that the Eclipse framework provides. Tools such as graphical editors, model repositories and transformation tools for Ecore models are widely available in the Eclipse tool ecosystem.

In particular, we make use of the abstract *EClassifier* class. *EClassifier* represents the class that classifies all objects, be it objects classified by a primitive type (*EDataType*) or a class (*EClass*). *EClasses* are composed of *EAttributes*, *EReferences* and *EOperations*. *EAttributes* refer to properties typed by primitive types, *EReferences* represent relations between the classes, and *EOperations* specify operations or methods that *EClasses* possess, along with their corresponding *EParameters*.

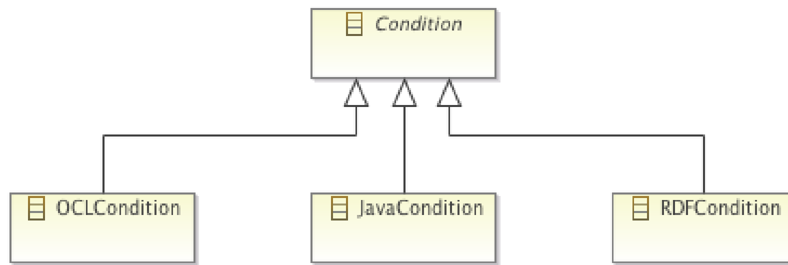


Figure 5.11: The Condition hierarchy.

In addition to the concepts that are part of the ontology, the information model view includes the *Condition* concept, as shown in Figure 5.11. A *Condition* is an expression that is evaluated in a given context to obtain its truth value. Among its uses we find the description of target states in *Goals* (cf. Section 5.4) and the guards in *ControlFlows* (cf. Section 5.6) and *MessageFlows* (cf. Section 5.7). The abstract *Condition* concept is specialized into *JavaConditions*, *OCLConditions*, and *RDFConditions*. Each specialization indicates the language used to encode the condition expression.

5.9 Deployment Viewpoint

In order to model the initial configuration of the MAS when the system starts, the deployment view is represented by the metamodel depicted in Figure

5.12. This view is centered on the *AgentInstance* concept. The *AgentInstance* represents an instance for any *Agent* type. Given that *Organizations* are also *Agents*, *AgentInstance* also models instance of *Organizations*.

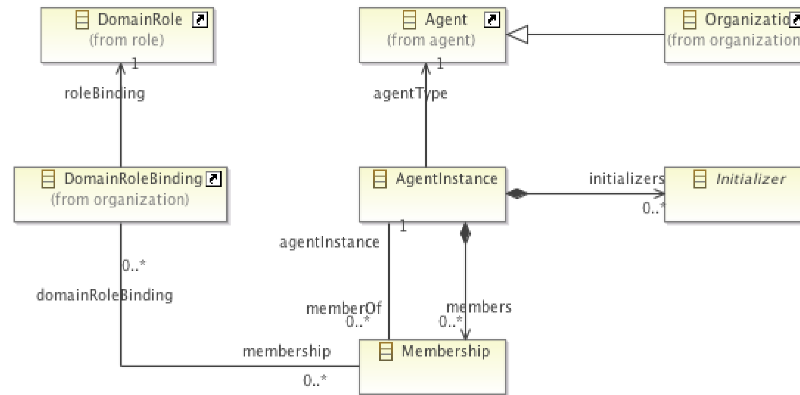


Figure 5.12: The metamodel for the Deployment View of PIM4Agents

The *Initializers* bound to each instance describe the initial state of the instance. For example, they may initialize its belief base or the *Goals* it assumes since the system start.

Instances of all *Agent* types may have a *memberOf* association with various *Membership* objects, while only instances of *Organizations* may have a *members* relation with *Membership* objects. The *Membership* concept indicates how an instance may be bound to another instance and under what *DomainRole*, as indicated by the corresponding *DomainRoleBinding*. In other words, the *Membership* concept models the initial establishment of the *Organizations*.

5.10 Case Study: The Conference Management System

In order to illustrate how the PIM4Agents can be applied to model a MAS, we have chosen a well studied example: The Conference Management System (CMS). This scenario has been discussed by other authors [DeL02, ZJW01] and has been modeled with other AOSE approaches such as O-MaSE [DeL07], Tropos [MNP^{+}07] and Prometheus [PTW07]. Therefore we can analyze the

analyse how PIM4Agents models address modeling issues with respect to the other approaches.

The CMS is a MAS that supports the management of conferences that requires the coordination of individuals in order to perform the selection of papers for publication at the conference. The process includes activities such as paper submission, paper review, paper selection, author notification, final paper collection and printing of the conference proceedings [PL07]. Authors may submit a paper until a given deadline. Once the deadline passes, the members of the Program Committee (PC) may forward the paper to referees for review or review the papers themselves. Once all the reviews are collected, a decision is made about whether to accept or reject each paper. Each author is notified of the decisions and the authors of accepted papers are asked to provide a camera-ready version of the paper. Once all camera-ready versions of the paper are collected, they are sent to the printer for publication.

In this section, we will present how this scenario is modeled with PIM4Agents using the concrete, graphical syntax of the language [WH08] and following the software process described in Chapter 4.

5.10.1 The CMS Goal Model

We start by identifying the purposes or goals of the systems. In the early requirement stages, we define a set of abstract goals. At this point, the only important issue to address is how goals relate to one another in terms of composition, i.e,. how complex goals decompose into simpler goals. Figure 5.13 presents the abstract goal tree for the *ManagePaperSubmission* goal.

Following the ordering labels that can be seen in the diagram, *ManagePaperSubmission* starts with the subgoal of *GetPapers*. This subgoal involves retrieving the submissions from the data bank or file system where they are stored. The following subgoal, *AssignPaper* involves matching reviewers and the submission(s) they should review. *ReviewPaper* describes the process of performing the actual review of the paper and producing a review report. Once the papers have been reviewed, the *SelectPapers* goal describes the process of choosing the papers that will be accepted for publication. The selection process is correspondingly subdivided into the goals of collecting the review reports (*CollectReviews*), and analyzing each review report, and deciding if the paper is to be accepted (*MakeDecision*). Once the decision has been reached for all submissions, the authors are notified of the result.

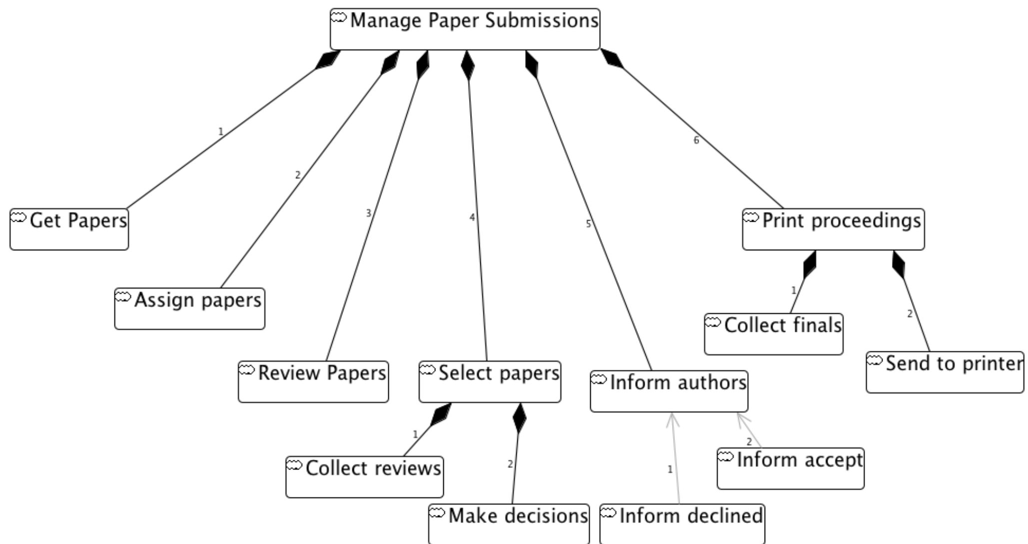


Figure 5.13: The abstract goal decomposition for ManagePaperSubmission

For this purpose, *InformAuthors* is OR-decomposed into *InformDeclined* and *InformAccepted*, which represent the transmission of the corresponding messages to each of the papers' authors. Finally, the *PrintProceedings* goal represents the stage where the proceedings are put together by collecting the camera-ready versions from the authors (*CollectFinals*) and sending these to the printer (*SendToPrinter*).

Once the abstract goals and the data types in the information model (see Section 5.10.2) have been defined, the abstract goals are refined into a concrete goal tree as presented in Figure 5.14. This refinement involves defining the concrete variables that are to be bound during the processing of the goal events, specifying which of the four concrete goal types should be used for each of the abstract goals, as well as describing the state to be reached by each goal. This state is described with a condition expression. For instance when we look at the goal *AssignPapers* in Figure 5.14, a variable *papers* is specified as input. This means that the variable will be previously bound by another goal, in this case, by the *GetPapers* goal. As an output variable, the mapping *paperAssignment* is specified. This variable will be bound as a result of the achievement of the goal. The goal will be achieved when the target condition is met. Therefore, *AssignPapers* will be achieved when all papers have been assigned to *PCMembers* in the *paperAssignment*

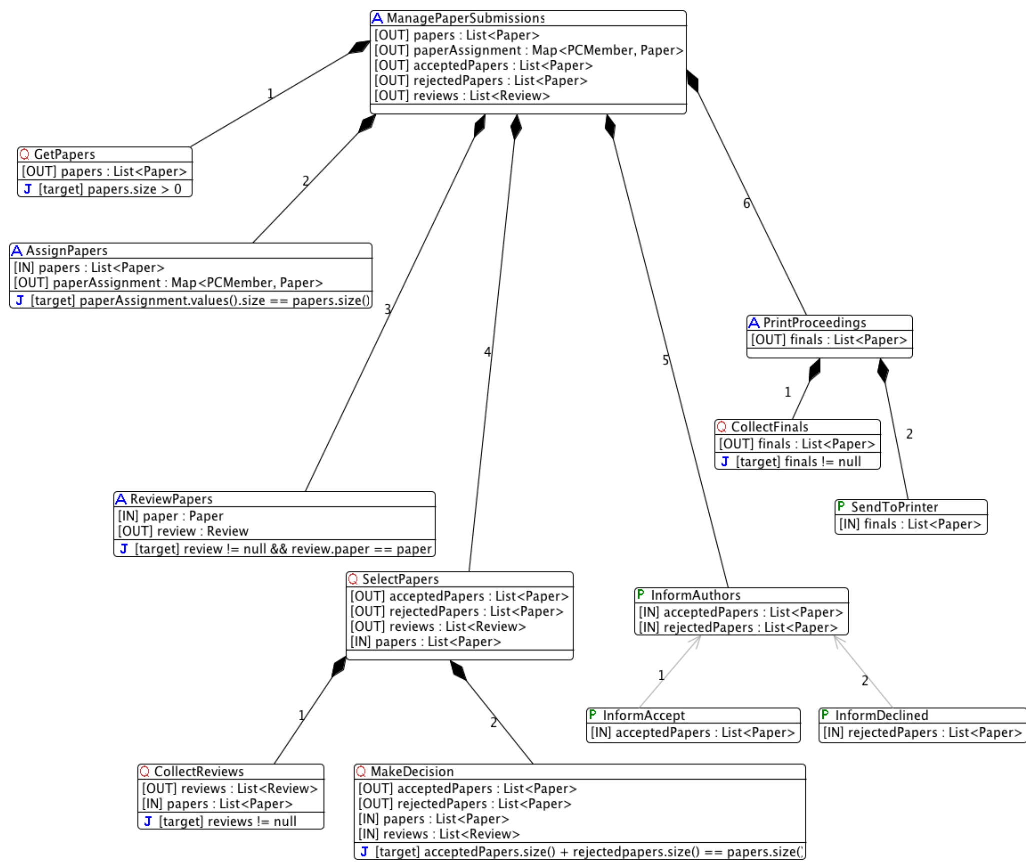


Figure 5.14: The concrete goal decomposition for ManagePaperSubmission

map.

As can be seen in the diagram, the target conditions of goals that are decomposed is not explicitly provided. The reasoning behind this is that the composed goals have their state description provided by the AND- or OR-conjunction of their subgoals. Therefore, only the leaves of decomposition tree have a explicit target condition.

Other AOSE approaches also start by modeling the system goals but with slight differences in process and purpose. For example, in O-MaSE [DeL07] goals are first defined roughly, and then they are refined by aggregating the parameters and precedence links between the goals. Even though the methodology steps are similar to the ones in PIM4Agents, one key difference is that the 'rough' goals and the refined ones are the same goals, just with a more information added to them in the refinement process step, while the abstract goals and concrete goals in PIM4Agents are separate concepts with a slight difference in abstraction. The abstract goals—as the protocols—are generic, since they do not really contain any domain specific information, but merely represent a concept that through its relations binds other domain specific concepts. This is what allows us to link abstract goals as postconditions in the protocol states and eventually link the plans that implement the protocol to the corresponding concrete goal. We consider that the precedence links in the O-MaSE goal diagram provide a clear picture of the dependencies between goals, but when a high number of goals is to be modeled the diagram can become overloaded. Therefore, PIM4Agents models use the ordering labels on the decomposition links to establish precedence and the variable initialization fields to represent the data dependencies instead.

Tropos [MNP+07] introduces the concept of a *soft goal*: an abstract notion that represents non-functional requirements, such as 'conference quality' in the CMS context. While we agree that they do make the specification more complete, since they are hard to quantify/implement, we have decided to leave such notions out of our model for the time being.

5.10.2 The CMS Information Model

The information model for this scenario consists of the basic set of classes necessary to handle the submission of papers. Depicted in Figure 5.15, this simple model introduces the classes: *Paper*, *Person*, *ReviewRequest* and *Review*.

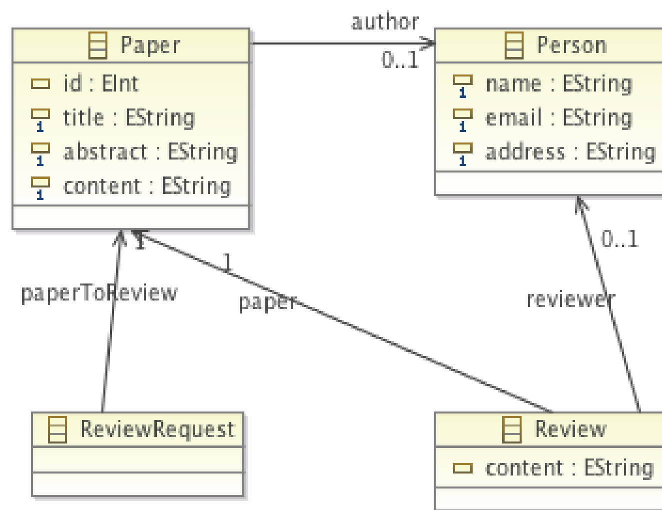


Figure 5.15: The Information Model for the CMS

The *ReviewRequest* represents content of the request to a *PCMember* to provide a review, therefore it refers to the paper to be reviewed. The *Paper* represents the article entity with properties such as its title, abstract and content. In addition an internal ID is assigned to it for management purposes. The information about authors and reviewers is represented by the class *Person*, containing mainly the contact information of the individual. The result of the interaction is the *Review* containing the assessment of the reviewer with respect to the article.

As previously mentioned, the information model is an Ecore model on its own, therefore a variety of tools can be used to create and manipulate it, as well as transform and generate code from it. This provides added flexibility in comparison to other approaches without leaving the definition of data layer partially or completely to the code level. For example, Prometheus simply abstracts the information model into databases (PapersDB, ReviewerDB, ReviewDB) and links the corresponding goals to the databases[PTW07].

5.10.3 The CMS Role Model

The Role Model is created by grouping goals that should be performed by an agent or organization that plays each given role. In our scenario, this view

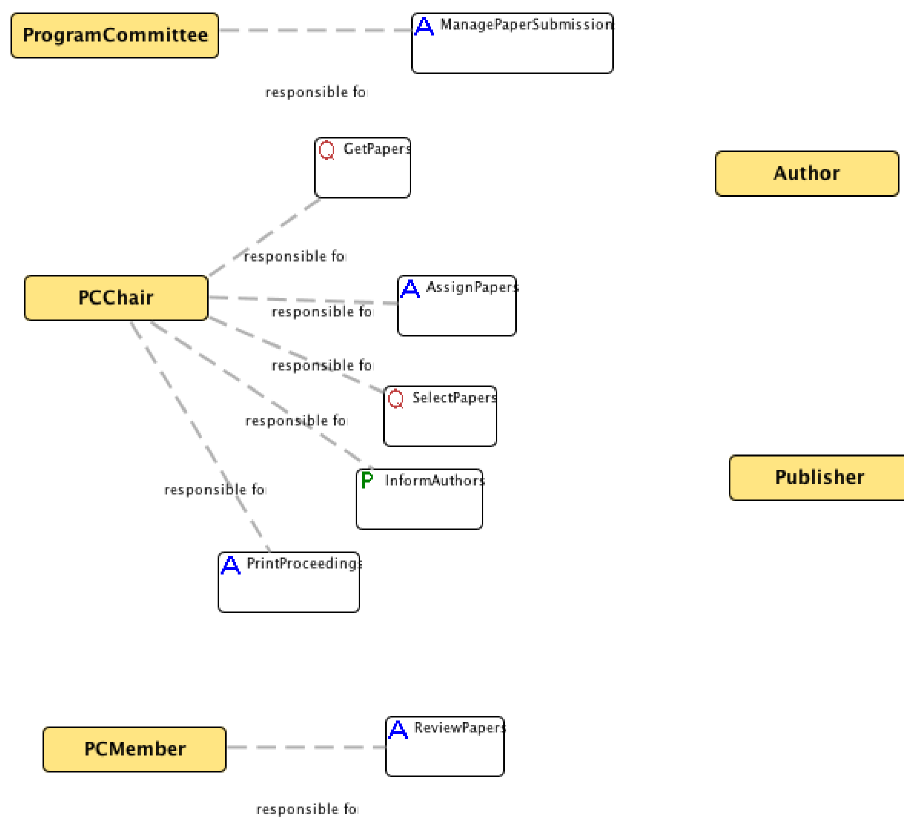


Figure 5.16: The Role View for the CMS example

is presented in Figure 5.16. Since we are concentrated in the goals internal to the Conference Management System, the responsibilities *Publisher* and *Author* roles are not presented.

The *ProgramCommittee* domain role is responsible for managing all of the paper submissions, which is the root goal of the goal hierarchy presented in Section 5.10.1. The subgoals of *ManagePaperSubmissions* are distributed between two domain roles: *PCMember* and *PCChair*. The *PCMember* contributes by reviewing papers, while the *PCChair* takes care of the other responsibilities, such as assigning papers to reviewers, selecting the accepted papers, and informing the authors about whether their paper was accepted or rejected.

This assignment of responsibilities to roles does not restrict that an agent that plays the *PCChair* may also review papers, but it does indicate that if that is desired such an agent should be playing both the *PCChair* and *PCMember* roles.

In contrast to other AOSE approaches, the PIM4Agents Role View is quite simple, since it concentrates on linking the DomainRoles with their goals–in the case of a goal-driven model, such as the CMS scenario—or their required behaviors—in the case of a behavior-driven model. Prometheus [PTW07] adds perceptions and actions to their role/actor diagrams while Tropos [MNP+07] takes a similar approach to PIM4Agents and groups the goals under Actors (Tropos equivalent of a DomainRole). The difference with respect to the presentation of perceptions and actions in the role view is merely a design decision, since these concerns can be addressed in other model views, such as the behavior model. In contrast, the role model in O-MaSE for this scenario [DeL07] distinguishes between Actors and Roles. Roles are smaller and more abstract such as Assigner and DecisionMaker and they are linked to Actors such as PCChair in a closer fashion to a DomainRole. This distinction can bring some compartmentation, but we consider that the aggregation of DomainRoles is functionally equivalent without requiring an additional concept.

5.10.4 The CMS Organization Model

The Organizational Model presents how organizations are related to roles. Roles can indicate the configuration of the organization ('requires' relation) as well as which roles it can play ('permitted to' relation).

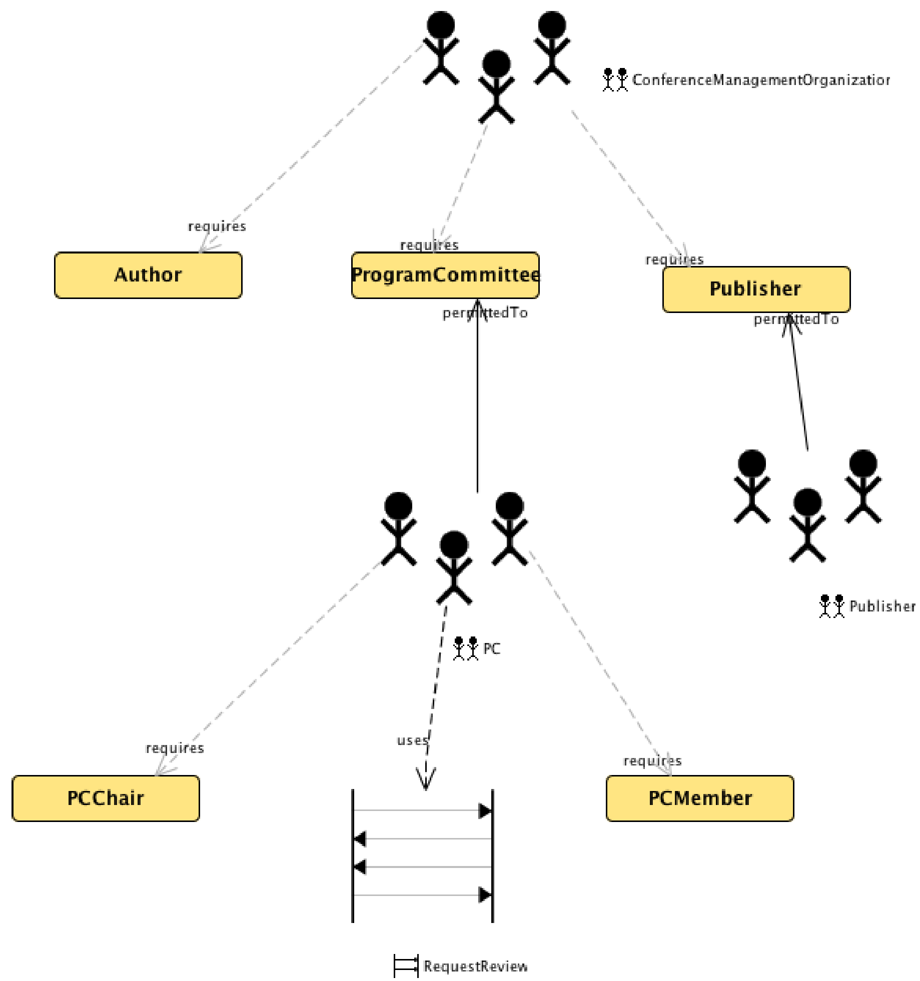


Figure 5.17: The Organization View for the CMS example

In our case study, three organizations are present: *ConferenceManagementOrganization*, *PC*, and *Publisher*. The *ConferenceManagementOrganization* includes all the parties involved in the case study. It is composed by the *Author*, *ProgramCommittee* and *Publisher* roles. The *Publisher* and *PC* organizations play the *Publisher* and *ProgramCommittee*, respectively.

The *PC* organization is composed by the *PCChair* and *PCMember* roles that have been previously presented. In addition, the *RequestReview* protocol is used by the *PC* to coordinate the review process. The details of the protocol will be presented in Section 5.10.6.

At this stage, it is still not specified exactly how many agents or organizations will fulfill each role, this information will be provided in the Deployment Model (Section 5.10.8).

Representing the organization as an independent concept is a feature that O-MaSE and PIM4Agents share, while Prometheus projects the organization into the agents beliefs [1] and Tropos sees the CMS system itself as an actor that decomposes into subactors. While all manage to represent the organizational structures in a somewhat equivalent fashion, the organization-as-an-agent approach possesses the advantage that it both agent and organizations interact in exactly the same way as role players in organizations.

5.10.5 The CMS Agent Model

The responsibilities and activities assigned for each agent type are modeled in the view presented in Figure 5.18. Our CMS scenario presents two Agent types: *Researcher* and *SeniorResearcher*. A *SeniorResearcher* is considered an experienced scientist that can evaluate work from his/her colleagues in a given area of expertise, while a *Researcher* is, simply put in this context, a scientist that is able to produce a scientific article. In reality, one could conceive that the *SeniorResearcher* is actually a specialization of *Researcher*. We have decided to leave out that association between the agent types for the sake of simplicity in this example.

As mentioned, a *Researcher* performs one main activity in this scenario: writing papers. Therefore it uses the corresponding *WritePaper* plan and it is assigned the role of *Author*.

[1] in [PTW07] an integration of the ISLANDER [EdlCS02] organisation design phase is promised in the future work

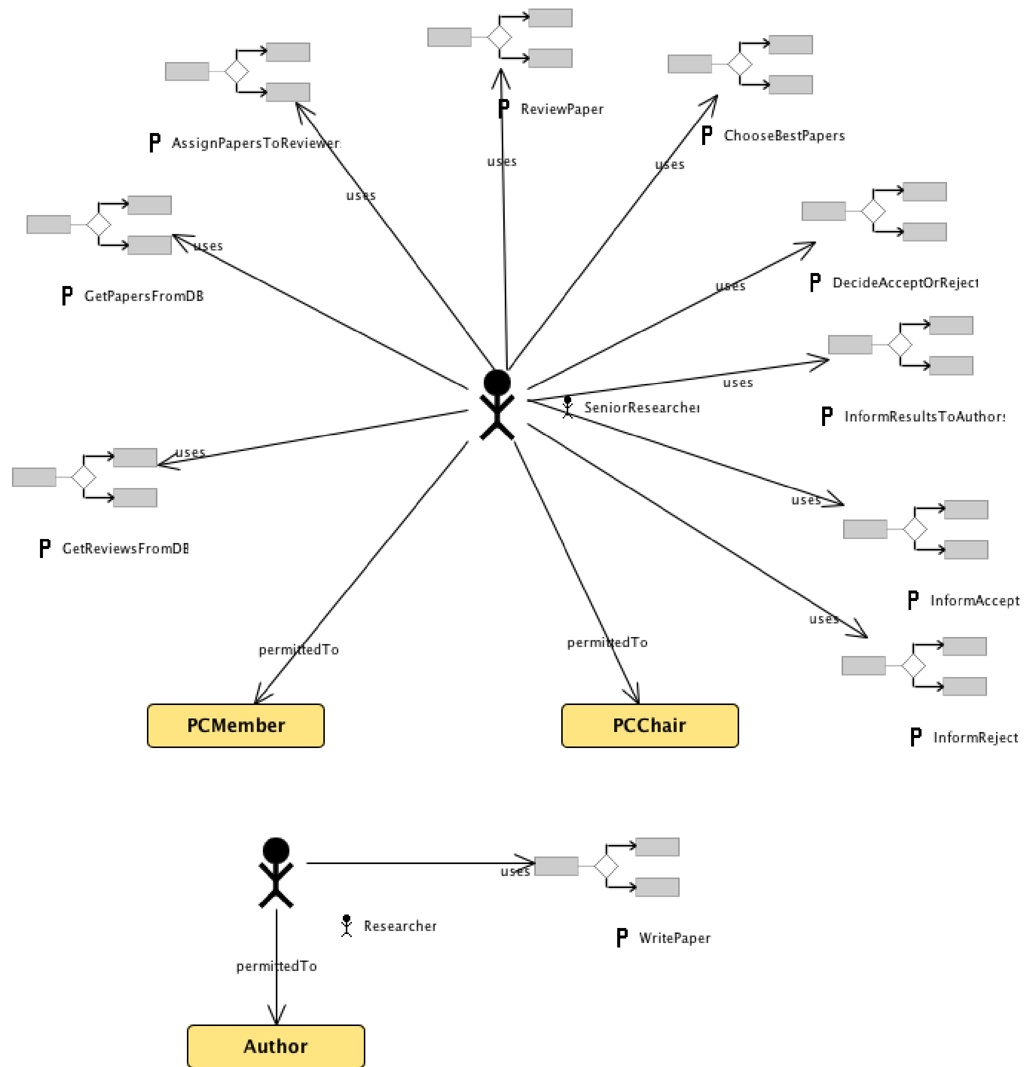


Figure 5.18: The Agent View for the CMS example

Correspondingly, the *SeniorResearcher* can play the *PCMember* and *PC-Chair* roles. In order to be able to play both roles, *SeniorResearcher* uses a number of behaviors that allow him to achieve the goals prescribed by the roles it has been assigned, such as *ReviewPaper* or *ChooseBestPapers*.

With respect to the agent view, the competing approaches and PIM4Agents are very similar. They all intend to link the agent types with the goals they should achieve in various ways: through the DomainRoles they play (PIM4Agents), as derivation from Actors (Tropos) or by a direct link (o-MaSE and Prometheus).

5.10.6 The CMS Interaction Model

Once the *Organizations* are structured with their corresponding *Domain-Roles*, it is necessary to specify how these roles interact within the boundaries of the organization. This specification is two-fold: a protocol specifies the message exchange between *Actors* and a collaboration diagram specifies how the *DomainRoles* are mapped to the protocol's *Actors*.

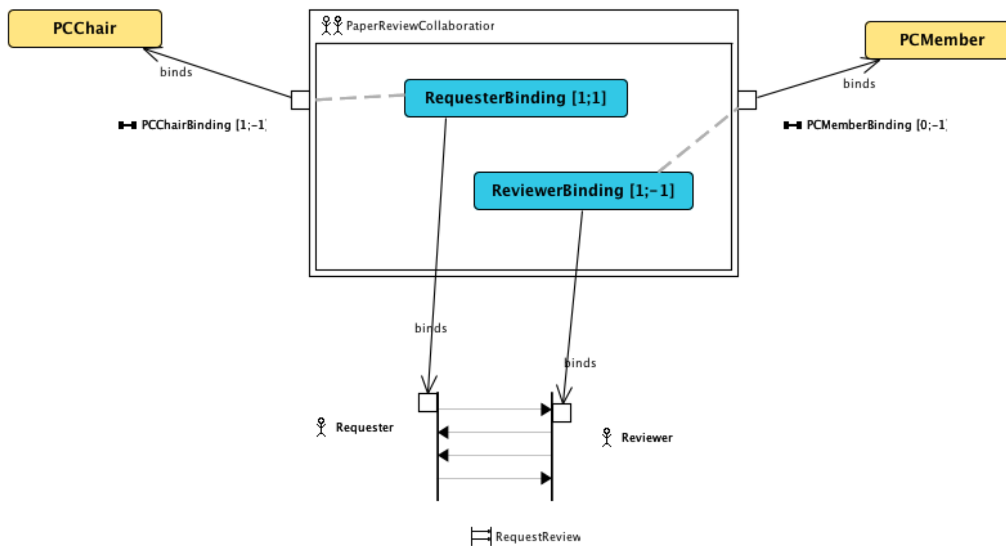


Figure 5.19: The PaperReviewCollaboration for the PC organization

In the case of the *PC* organization, we have defined a collaboration known as *PaperReviewCollaboration*. This collaboration is presented in Figure 5.19.

As the diagram presents, the *PCChair* role is bound with the *PCChairBinding* to the collaboration and this binding specifies a minimum cardinality of 1 and a maximum cardinality of -1 (no upper bound). Furthermore, the *PCChairBinding* is linked to the *RequesterBinding*, which specifies the cardinalities, $[1, 1]$, to be used inside the *RequestReview* protocol for the *Requester* actor. Likewise, the *PCMember* role is linked with the with the *ReviewerBinding* through the *PCMemberBinding*, and the corresponding cardinalities at each stage are specified.

These bindings will allow that eventual runtime instances of this organizations may be assigned dynamically to the *Actors* in the protocol, while still respecting the cardinalities specified.

Although it is not explicit in the graphical representation, the *Collaboration* also contains the *Messages*. Messages are the domain dependent construct that corresponds to an *ACLMessage*, the domain independent concepts used to models protocols. Since the *Messages* have a direct link to their corresponding *ACLMessage*, the collection of messages in the *Collaboration* provides the necessary information to instantiate the protocol specification for the application domain.

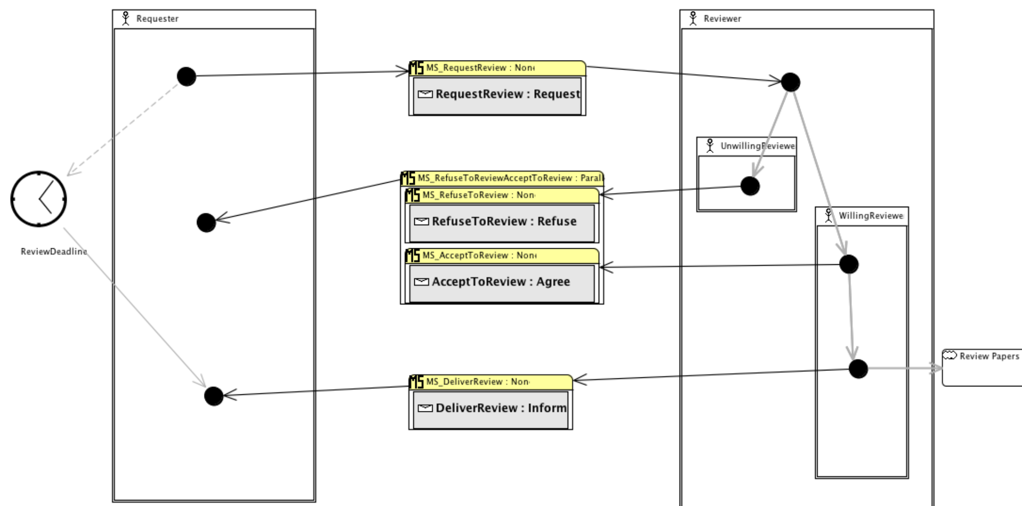


Figure 5.20: The RequestReview protocol

As mentioned before, the exchange of messages is modeled as a protocol. Figure 5.20 presents the protocol that models the *RequestReview* message exchange. In this example, the *Requester* sends a *RequestReview* ACL message

to each *Reviewer*. The *Reviewer* must decide whether he is willing to provide a review. *Reviewers* that accept to review the paper send the corresponding *AcceptToReview* message and therefore are considered *WillingReviewers*. Correspondingly, *UnwillingReviewers* represents the users that refuse to review the paper and send the corresponding *RefuseToReview* ACL message. After the acceptance to produce a review, the *Requester* will be expecting a *DeliverReview* message from each reviewer before the deadline specified in *ReviewDeadline*.

In this example, an explicit link between the protocol and goal models is presented. In the state that represents the sending action of the *DeliverReview* message by the *WillingReviewer*, the abstract goal *ReviewPapers* is used to represent the state's postcondition. This direct connection will enable to better integrate the behaviors that handle communication—generated in accordance to the specified protocols—and other behaviors that handle the goals specified in the goal model in the specification of the behavior model that will follow.

The use of the abstract goals as postconditions also allows to keep the protocol domain independent. It should be possible to reuse a protocol specification in various situations, where only the content of the messages specified would be domain dependent. By using abstract goals, we preserve this principle, while the goal model and its links between abstract and concrete goals provide the domain information needed to execute the protocol.

Other approaches [PTW07, MNP+07, DeL07] use variations of UML and AgentUML diagrams to represent message exchange and interactions for the CMS scenario. While these are well known within the MAS community, we consider that the PIM4Agents protocol and collaboration views provides advantages over these for dealing with situations such as dealing with partitioning the entities taking part in the interaction (subactors) or the explicit mapping of DomainRoles to protocol's Actors (Actor- and RoleBindings).

5.10.7 The CMS Behavior Model

The Behavior model in the CMS scenario is started by the *ManagePaperSubmissions* plan executed by the *PC* organization playing the *ProgramCommittee* role. This plan is triggered by a corresponding *ManagePaperSubmissions* goal event, represented by a knowledge labeled trigger, depicted in Figure 5.21. We take this plan to exemplify he modeling the goal delegation behav-

ior, additional behavior examples will be provided in Section 9.

Following the concrete goal decomposition presented in Figure 5.14, a chain of goal delegations is performed following the sequence prescribed in the decomposition. For every goal delegation, there is an internal task that prepares the corresponding goal event (as a task output), before it is delegated. This preparation step may also include retrieving required data (as task input) from the previous goal delegation.

The first delegated goal is the *GetPapers* goal. After the goal is achieved, its event contains the list of submitted papers. This list is then passed along to the *AssignPapers* goal, which is delegated and, if achieved, provides a mapping between the *PCMembers* and the papers to be reviewed.

This mapping is used in a *Loop* structured activity to iterate through the *PCMembers* and delegate the corresponding *ReviewPaper* goal events. Once all reviews are in or the achieve timeout expires, the list of reviews is passed along to the *SelectPapers* goal. When the goal is achieved, the papers are split into lists of accepted and reject papers, which are then delegated to the *InformAuthors* goal. Once the authors have been informed, the accepted papers are sent to the publishers for publication by the delegation of the *PrintProceedings* goal.

As with the interactions, the most competing AOSE approaches use UML-like diagrams to model the activities, which allows an easy comprehension by new users, but can fall short in situations such as dealing with dynamic number of execution traces (how many trace instances follow which execution path) which is usually closely linked to the partitioning of interaction Actors (subactors).

5.10.8 The CMS Deployment Model

The final view on the MAS system modeling process represents the system initial configuration or deployment. Since a MAS is a dynamic system, this merely represents how the system will be configured when it is started.

Following our scenario we have one instance of the *ConferenceManagementSystem* organization named AAMAS. AAMAS has a series of agents or organizations playing the required roles: *Authors* (Author1 to AuthorN), a *Publisher* (Springer) and a *ProgrammCommittee* (PC). The PC organization is correspondingly composed by a *PCChair* (Jörg) and several *PCMembers* (Klaus, Stefan, Esteban, and Cristián).

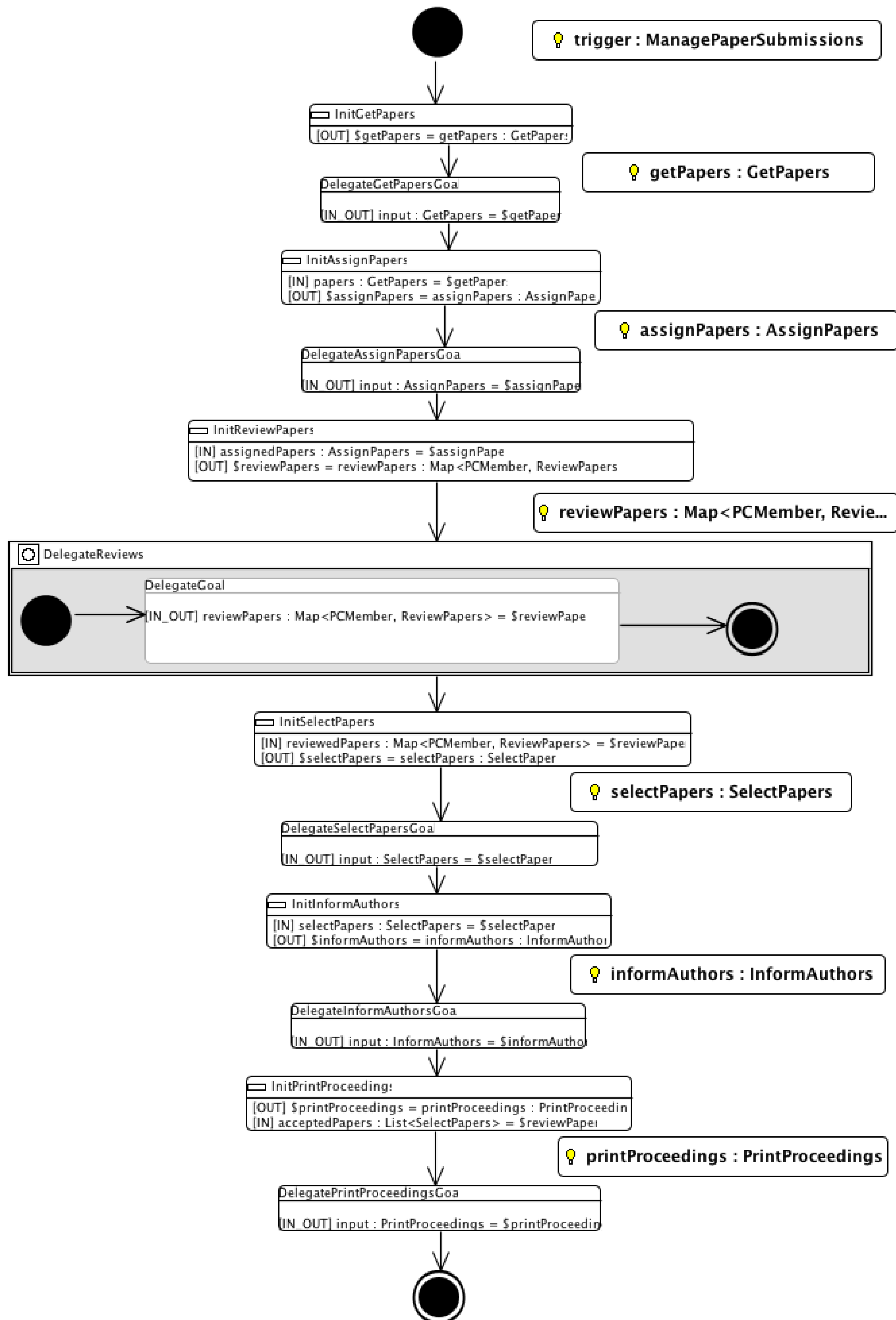


Figure 5.21: The ManagePaperSubmissions plan

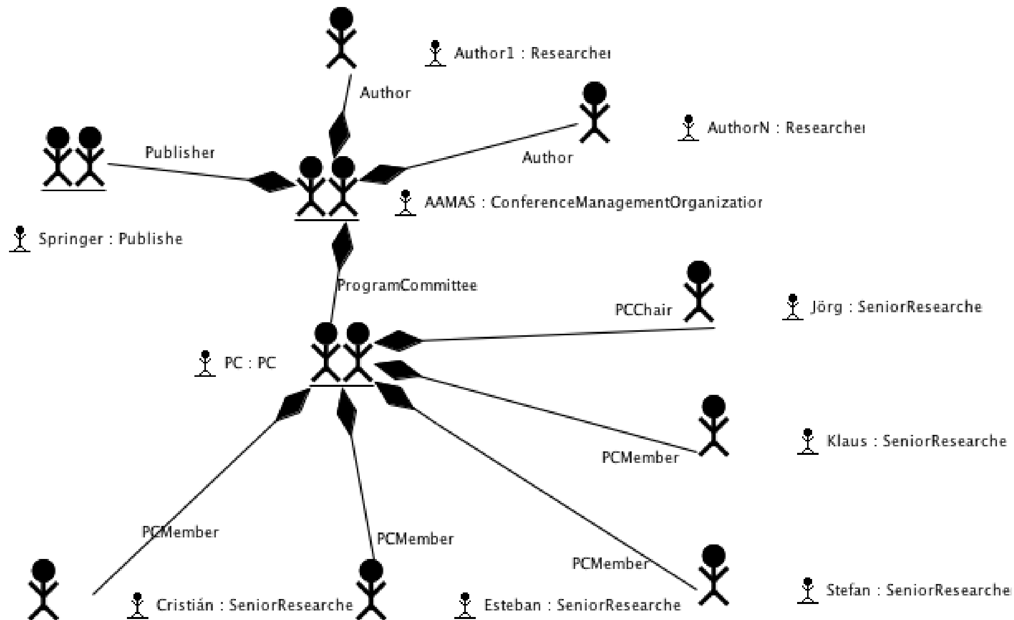


Figure 5.22: The CMS deployment view

The *Membership* links between the instances model the role under which every agent/organization instance is bound to the organization of which it is a member. Each *Membership* is linked to a *DomainRoleBindings*, such as the ones from the Interaction View (see Figure 5.9).

None of the other approaches that modeled the CMS scenario present a specific initialization view for the CMS scenario. Although this can be specified with various degrees of difficulty—depending on the agent platform used—at the code level, having a PIM-level view of the system initialization provides a clear understanding of the complexity and size of the running system.

5.11 Summary

In order to model a MAS, various dimensions of the scenario must be modeled and brought together. In the PIM4Agents, we have addressed each of these dimensions in a separate viewpoint.

Each viewpoint comprehends a subset of concepts from the PIM4Agents

metamodel and a corresponding diagram to ease the creation of the models and their presentation to colleagues for discussion as part of model development.

In this chapter, we have presented the PIM layer of our model-driven approach. The metamodel and its graphical concrete notation enable the creation of valid models that will be used in later of the model-driven process. The following chapters will present one of the PSMs we address along with the transformation between PIM and PSM.

Chapter 6

Organizations and Roles in JADE: JadeOrgs

Since the overall computation in Multiagent Systems (MAS) is obtained by the combination of the autonomous computation of every agent in the system and the communication among them [SF03], the coordination and communication among agents is essential. However a flat structure where every agent communicates with every other agent is usually too expensive and inefficient, designing agents to act within an organizational structure can provide additional encapsulation, thus simplifying representation and design. Modularization, code reuse and incremental deployment are further advantages. However, these coordination or organizational structures are not always explicitly supported by agent platforms, even when some agent meta-models and methodologies do present them. We consider that organizations and their corresponding role structure can reduce interoperability problems since they help specify the scope of interactions within and outside the organization. Additionally, the evaluation of organization members against a set of requirements, namely role descriptions, reduces the possibility that unfit parties/agents can join and also potentially enables them to look for ways to comply with these requirements in order to take part of the interactions inside the organization.

This chapter presents JadeOrgs, an organization-oriented extension for the JADE agent platform [BPR99]. First, we introduce a formal specification of our approach to organizations in JADE. Second, we present the JadeOrgs metamodel, along with the runtime library that provides the implementation of the agent behaviors and classes introduced in the metamodel. Finally, we compare JadeOrgs against other AOSE metamodels and against a competing implementational approach of runtime organizations in JADE.

6.1 Formal Specification of Organizations in JADE

Given the social capabilities and the finite resources that agents possess, it is apparent that they need to organize themselves in order to coordinate their actions and improve their utility. In the literature, we find different ways in which agents organize themselves. These societies of agents are denominated as groups, teams, coalitions, or, simply, organizations. At the same time, agent societies must describe the roles, norms and goals of the society instead of just individual agent states [DD01].

In this thesis, we propose a general model-driven framework in which organizational concepts are represented at the different abstraction levels from the PIM to the PSM to the running code. Most of the works presented in Chapter 3 introduce organizational concepts, such as Organization or Role, at the modeling level, but these are internalized in the agent implementation and, therefore, 'disappear' in the running code. We argue that preserving these concepts at runtime can allow the agent to reason about the roles it can perform and communication can be simplified by having a single point of contact when communication with an organization is desired.

In this section, we present the formal specification of the abstractions that we will use to represent organizational concepts at the PSM and runtime levels. The specification is presented at a high level of abstraction and, consequently, additional details of some of the operations as well as the specialization of some of the types presented will be introduced in their corresponding representations in JadeOrgs. The specification is described in Object-Z [Smi00, DR00], an extension of the Z language [Spi89] to facilitate specification in an object-oriented style.

6.1.1 Basic types

We must first introduce some basic types used by the concepts that we will use:

LITERAL is any expression that can be assigned to a variable or condition,

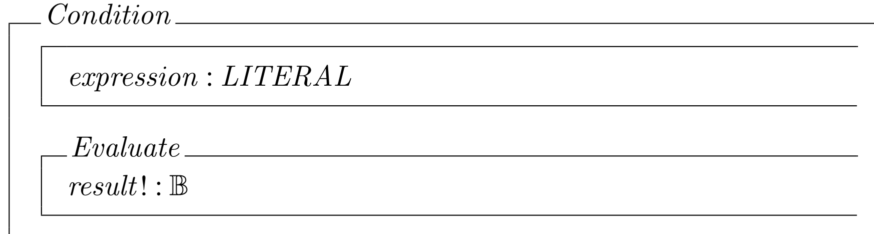
NAME is the identifier of a concept,

PLANBODY represents the description of the steps executed inside a plan,

PERFORMATIVE represents the speech act conveyed by a message,

TYPE represents the type of a field or variable such as primitive types and classes, and

ONTOLOGY is the formal representation of the knowledge the agents possess. It consists of the set of objects and the relations between them that are present in the application domain.



Schema 1: Class schema for Condition

6.1.2 Condition

Now we introduce the *Condition* in Schema 1 .

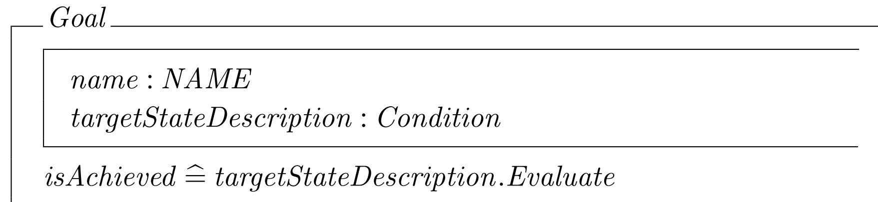
Definition 6.1.1 *A* ***condition*** *is defined as a singleton* $C = \langle expression \rangle$ *where expression represents the description of a state of affairs in a given language.*

Conditions can be evaluated in order to obtain their truth value, this operation is represented by the operation Evaluate. The Condition provides us with the foundation in order to describe *Goals*.

6.1.3 Goal

A goal is a mental attitude representing preferred progressions of a particular multiagent system that the agent has chosen to bring about [vRDW08]. In this specification, we only describe a simple, abstract type of goal, while in JadeOrgs and PIM4Agents we refer to various concrete goal types (perform, achieve, query, and maintain). Presented in Schema 2, the Goal also defines an operation that indicates whether the state of affairs prescribed by the goal has been reached by evaluating the condition that describes its target state.

Definition 6.1.2 *A* ***goal*** *is a pair* $G = \langle name, targetStateDescription \rangle$*, where name identifies the goal and targetStateDescription represents a desired state of affairs to be reached or maintained.*

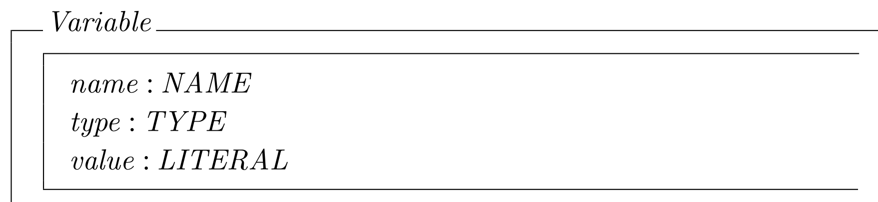


Schema 2: Class schema for Goal

6.1.4 Variable

In similar fashion to programming languages, in order to represent local data we define *Variables* (cf. Schema 3).

Definition 6.1.3 *A* ***variable*** *is given as a triple* $V = \langle name, type, value \rangle$ *where name identifies the variable, type indicates its type, and value describes the variables initial value.*



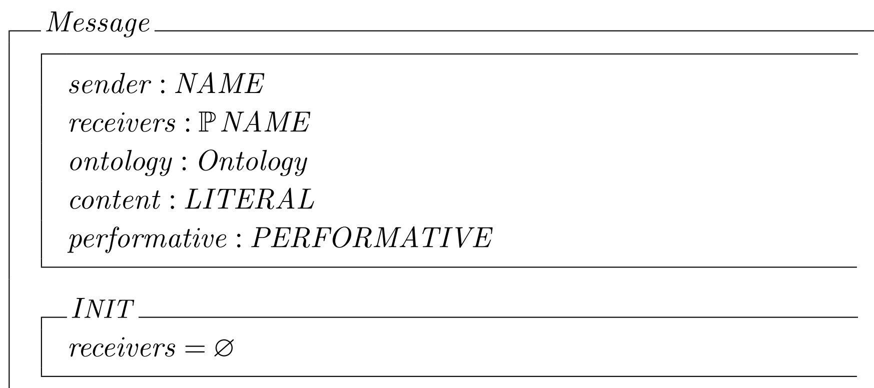
Schema 3: Class schema for Variable

Variables also contain all the locally stored information that will be used in the plans that represent the agent's behavior and they are also used to represent the agent's knowledge.

6.1.5 Message

Messages represent the communication exchanges among the agents and they are modeled as presented in Schema 4.

Definition 6.1.4 *A* ***message*** *is given as a 4-tuple* $M = \langle sender, receivers, ontology, content, performative \rangle$ *where:*



Schema 4: Class schema for Message

sender *represents the identity of the agent that sends the message,*

receivers *lists the agents that should receive the message,*

ontology *indicates the ontology that contains the concept represented in the content expression,*

content *represents the payload of the message, and*

performative *indicates the communicative action intended by the sender agent.*

There are no additional communication structures provided given that in JADE, and respectively JadeOrgs, there is no representation of protocol structures as such. Instead they are only projected into behaviors that implement such protocols.

6.1.6 Plan

Agent behavior is what enables the agent to take action and bring about the changes in its environment in accordance to its intentions and purpose. One way to model agent behavior is through the concept of a *Plan*. *Plans* provide a way to react or take action in a given situation. We use this notion of plan as an abstraction for the complex behavior hierarchy present in JADE (cf. 6.2.3). We model this abstract concept of *Plan* in Schema 5.

Schema 5: Class schema for Plan

Definition 6.1.5 *A **plan** is given as a 5-tuple* $P = \langle name, achieves, precondition, localVariables, planBody\rangle$ *where:*

name *identifies the plan,*

achieves *lists the goals that may be achieved by the plan or the goal events to which the agent can react by executing this plan,*

precondition *indicates an expression that must evaluate to **true** in order for the plan to be executed,*

localVariables *represent the local data used in the execution of the plan, and*

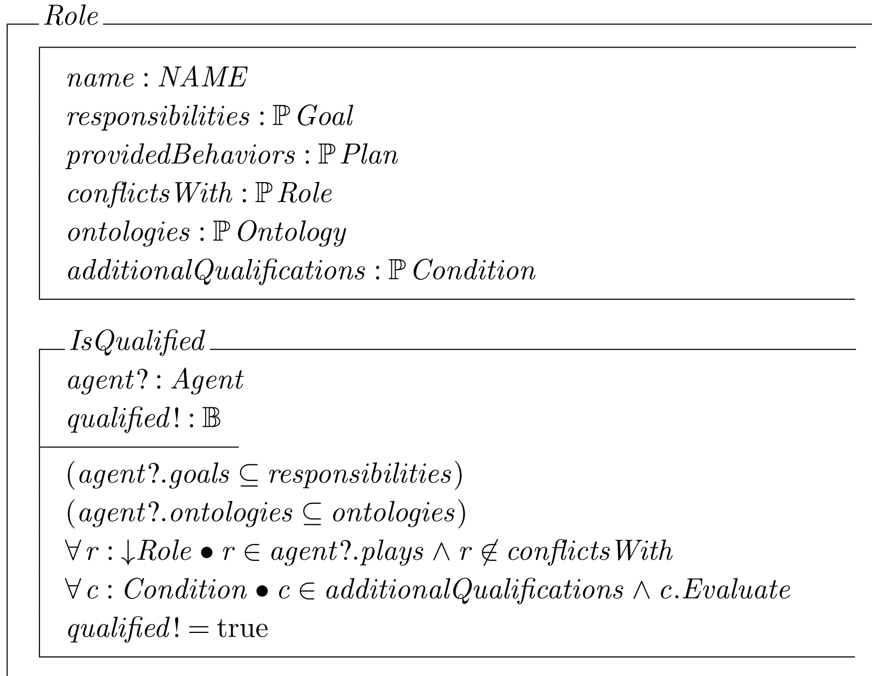
planBody *represents the process that is executed in this plan.*

The *Plan* is represented in JadeOrgs as an *FSMBehaviour* (cf. Section 6.2.3) whose states are derived from the process steps in planBody. Each process activity is implemented in a behavior and the state transitions are determined depending on the connections and dependencies between the different activities.

6.1.7 Role

Just as in the theater where an actor is assigned to play a part or role, in a MAS, the actor is an agent. Roles specify the expectation/requirement of behavior and other features for agents. They provide both major building blocks for agent social systems and the requirements by which agents interact.

Each agent is linked to other agents by the roles it plays by virtue of the system's functional requirements—which are based on the expectations that the system has of the agent. To specify how this these requirements are specified, we use the concept of a *Role* presented in Schema 6.



Role

$name : NAME$
$responsibilities : \mathbb{P}\, Goal$
$providedBehaviors : \mathbb{P}\, Plan$
$conflictsWith : \mathbb{P}\, Role$
$ontologies : \mathbb{P}\, Ontology$
$additionalQualifications : \mathbb{P}\, Condition$

IsQualified

$agent? : Agent$
$qualified! : \mathbb{B}$

$(agent?.goals \subseteq responsibilities)$
$(agent?.ontologies \subseteq ontologies)$
$\forall r : \downarrow Role \bullet r \in agent?.plays \land r \notin conflictsWith$
$\forall c : Condition \bullet c \in additionalQualifications \land c.Evaluate$
$qualified! = \mathrm{true}$

Schema 6: Class schema for Role

Definition 6.1.6 *A* ***role*** *is given as a 6-tuple* $M = \langle name, responsibilities, providedBehaviors, conflictsWith, ontologies, additionalQualifications \rangle$ *where:*

name *identifies the role,*

responsibilities *represent the goals that the agent commits to achieve for the organization when playing this role,*

providedBehaviors *lists the set of behaviors that the agent gains when playing this role,*

conflictsWith *specifies the other roles that conflict with this role,*

ontologies *identifies the ontologies that agents that play this role must know, and*

additionalQualifications *specify additional constraints that the role player must fulfill in other to be allowed to play this role.*

The Role Schema also specifies the IsQualified operation. This operation returns the value "true" only if all responsibilities can be fulfilled by the agent, all required ontologies are known by the agent, all roles the agent plays do not conflict with the given role and that any additional qualifications specified are fulfilled. These additional qualifications are encoded into the behaviors that handle the Establishment protocol (cf. Section 6.3.2).

6.1.8 Agent

The *Agent* represents the core of the MAS. As reviewed in Section 2.1, it is the autonomous entity that is able to react to conditions in its environment, pursue its own goals and communicate with other agents in order to achieve tasks. Therefore, we define the concept of an *Agent* and specify its properties in the context of multiagent organizations in Schema 7.

Definition 6.1.7 *An* ***agent*** *is given as a 9-tuple* $M = \langle name, plays, behaviors, memberOf, goals, knowledgemessageQueue, ontologies\rangle$ *where:*

name *identifies the agent,*

canPlay *specifies the roles that the agent type can take,*

plays *lists the set of role that the agent has committed to play,*

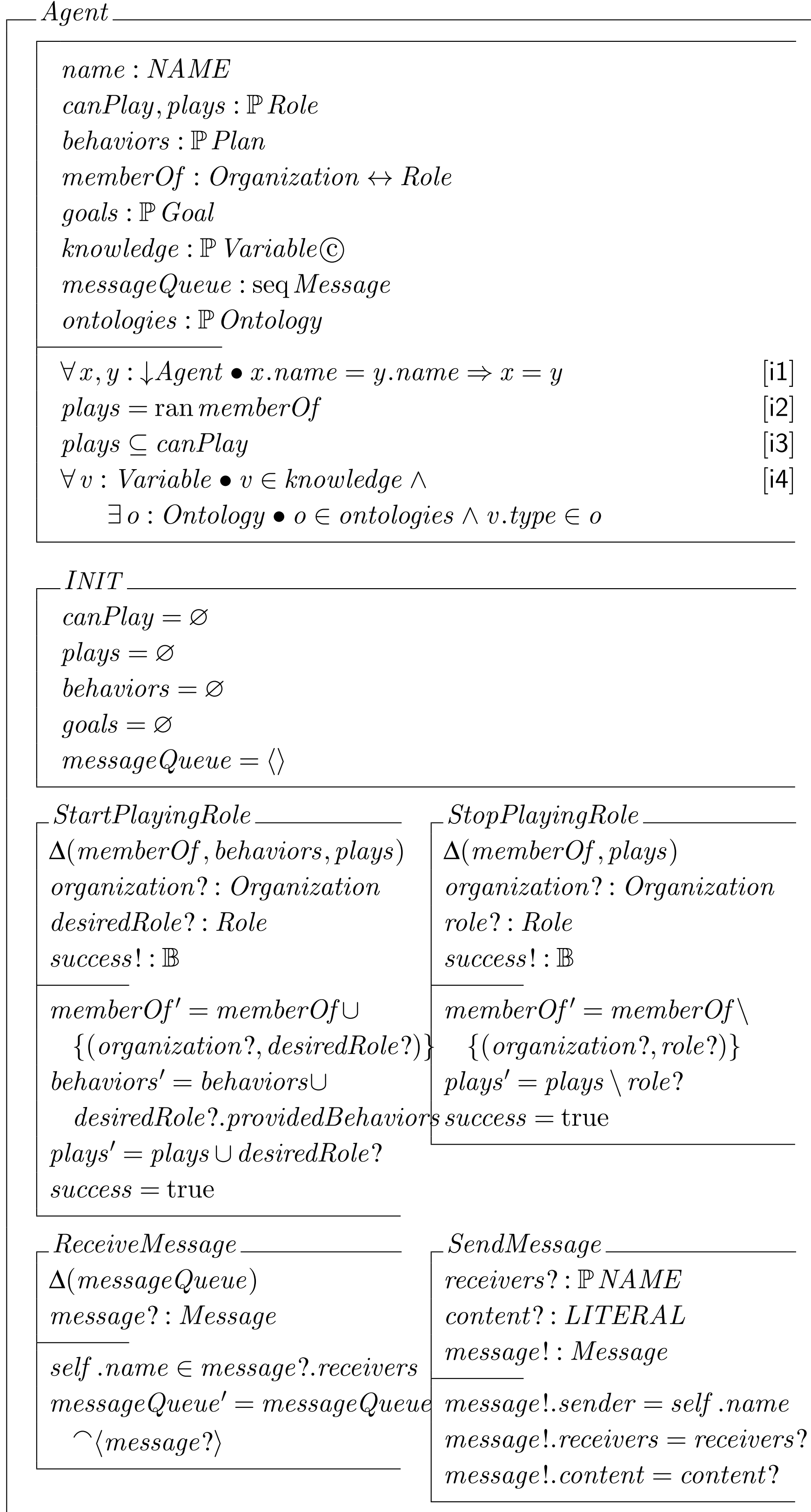
memberOf *represents the way the agent is associated with an organization and under what role,*

goals *presents the goals that the agent pursues*

knowledge *provides the data that represents the agent's beliefs,*

messageQueue *contains the messages that have been sent to this agent, and*

ontologies *specify the ontologies known by the agent.*



Agent

$name : NAME$
$canPlay, plays : \mathbb{P}\, Role$
$behaviors : \mathbb{P}\, Plan$
$memberOf : Organization \leftrightarrow Role$
$goals : \mathbb{P}\, Goal$
$knowledge : \mathbb{P}\, Variable©$
$messageQueue : \mathrm{seq}\, Message$
$ontologies : \mathbb{P}\, Ontology$

$\forall x, y : \downarrow Agent \bullet x.name = y.name \Rightarrow x = y$ [i1]
$plays = \mathrm{ran}\, memberOf$ [i2]
$plays \subseteq canPlay$ [i3]
$\forall v : Variable \bullet v \in knowledge \land$ [i4]
$\quad \exists o : Ontology \bullet o \in ontologies \land v.type \in o$

INIT

$canPlay = \varnothing$
$plays = \varnothing$
$behaviors = \varnothing$
$goals = \varnothing$
$messageQueue = \langle\rangle$

StartPlayingRole

$\Delta(memberOf, behaviors, plays)$
$organization? : Organization$
$desiredRole? : Role$
$success! : \mathbb{B}$

$memberOf' = memberOf \cup$
$\quad \{(organization?, desiredRole?)\}$
$behaviors' = behaviors \cup$
$\quad desiredRole?.providedBehaviors$
$plays' = plays \cup desiredRole?$
$success = \mathrm{true}$

StopPlayingRole

$\Delta(memberOf, plays)$
$organization? : Organization$
$role? : Role$
$success! : \mathbb{B}$

$memberOf' = memberOf \setminus$
$\quad \{(organization?, role?)\}$
$plays' = plays \setminus role?$
$success = \mathrm{true}$

ReceiveMessage

$\Delta(messageQueue)$
$message? : Message$

$self.name \in message?.receivers$
$messageQueue' = messageQueue$
$\quad \frown \langle message? \rangle$

SendMessage

$receivers? : \mathbb{P}\, NAME$
$content? : LITERAL$
$message! : Message$

$message!.sender = self.name$
$message!.receivers = receivers?$
$message!.content = content?$

Schema 7: Class schema for Agent

Also in Schema 7, a series of invariants are defined in order to preserve the consistency of the agent properties:

i1 makes agent names unique,

i2 ensures that the agent can only play roles inside a given organization,

i3 guarantees that the agent can only play roles that are contained in the *canPlay* relation, and

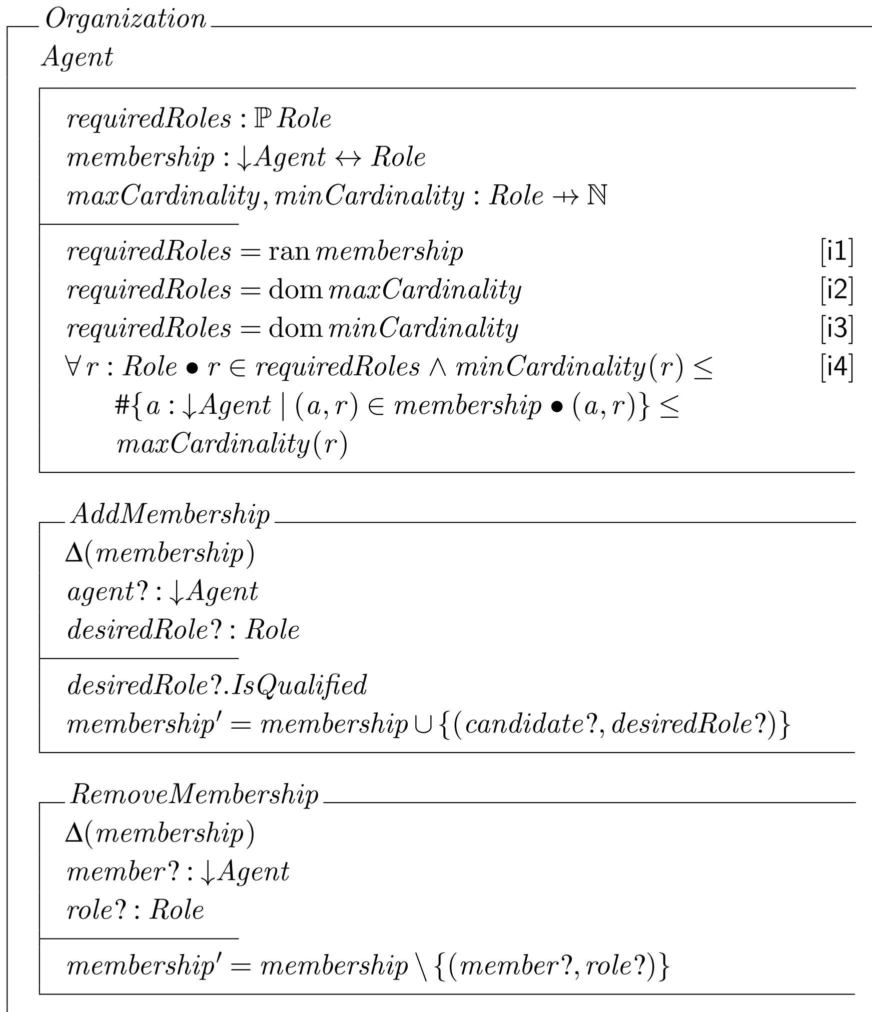
i4 ensures that the type of all knowledge variables is described in one of the known ontologies.

In addition, a series of operations are defined for the agent. Two of these operations specify the communication. *ReceiveMessage* presents how received messages are included into the agent's *messageQueue*, while *SendMessage* produces a message to be delivered by the *AgentPlatform*.

The activation of the roles from the agent's perspective is achieved with the *StartPlayingRole* operation. It updates the *memberOf* relation, adds provided behaviors to agent's behavior set and also activates the role by adding it to the *plays* relation. In the case that an agent leaves the organization and stops playing the role, the *StopPlayingRole* is executed. the *memberOf* and *plays* relations are updated accordingly. The behaviors that may have been added in *StartPlayingRole* cannot be taken away from the agent, since these behaviors represent the process that the agent has learned from playing this role. The counterpart operations of this organization joining/leaving process will be presented in the next subsection.

6.1.9 Organization

As agents work together, they establish groupings that are more stable and structured than just a random set of agents interacting. We represent these structured groupings as *Organizations*. The *Organization* defines what agents are necessary to achieve certain organizational goals and under what conditions they may do it. In our approach, the *Organization* is also an *Agent* in its own right. It has it own (organizational) goals and it may posses it own behavior to coordinate with the organization members and/or orchestrate organizational processes. Schema 8 presents the abstract specification for the *Organization* concept and its definition is presented in Definition 6.1.8.



Organization

Agent

$requiredRoles : \mathbb{P}\, Role$

$membership : \downarrow Agent \leftrightarrow Role$

$maxCardinality, minCardinality : Role \nrightarrow \mathbb{N}$

$requiredRoles = \operatorname{ran} membership$ [i1]

$requiredRoles = \operatorname{dom} maxCardinality$ [i2]

$requiredRoles = \operatorname{dom} minCardinality$ [i3]

$\forall\, r : Role \bullet r \in requiredRoles \land minCardinality(r) \leq \#\{a : \downarrow Agent \mid (a, r) \in membership \bullet (a, r)\} \leq maxCardinality(r)$ [i4]

AddMembership

$\Delta(membership)$

$agent? : \downarrow Agent$

$desiredRole? : Role$

$desiredRole?.IsQualified$

$membership' = membership \cup \{(candidate?, desiredRole?)\}$

RemoveMembership

$\Delta(membership)$

$member? : \downarrow Agent$

$role? : Role$

$membership' = membership \setminus \{(member?, role?)\}$

Schema 8: Class schema for Organization

Definition 6.1.8 *An **organization** is given as a 14-tuple* $M = \langle name, plays,$ $behaviors, memberOf, goals, knowledgemessageQueue, ontologies, requiredRoles,$ $membership, maxCardinality, minCardinality\rangle$ *where:*

name *identifies the agent,*

canPlay *specifies the roles that the organization type can take,*

plays *lists the set of role that the organization has committed to play,*

memberOf *represents the way this organization is associated with another organization and under what role,*

goals *presents the goals that the organization pursues*

knowledge *provides the data that represents the organization's knowledge,*

messageQueue *contains the messages that have been sent to this organization,*

ontologies *specify the ontologies known by the organization,*

requiredRoles *specifies with roles the organization needs to operate,*

membership *relates the members of the organizations with the roles that they play within it,*

maxCardinality *specifies the maximum number of agents that can play a given role,*

minCardinality *specifies the minimum number of agents that can play a given role,*

Just as with *Agents*, we define invariants for the additional properties of an *Organization*:

i1 guarantees that the range of the *membership* relation is equal to the set of *requiredRoles*,

i2 ensures all roles in *requiredRoles* have a corresponding value in the maxCardinality relation,

i3 ensures all roles in *requiredRoles* have a corresponding value in the minCardinality relation, and

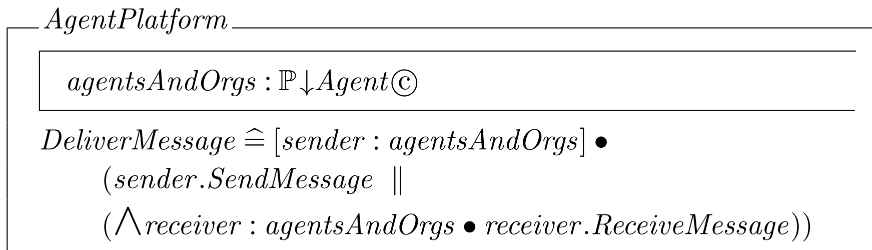
i4 verifies the number of agents that play a given role inside the organization is between the values specified in the *maxCardinality* and *minCaridinality* functions.

The basic operations that the *Organization* must perform involve the addition and removal of members. *AddMembership* adds a new $\langle Agent, Role \rangle$ pair to the *membership* relation only if the role's *isQualified* operation evaluates to true. This implies that the agent fulfills the basic requirements established in the role. Correspondingly, *RemoveMembership* removes the specified $\langle Agent, Role \rangle$ pair from the *membership* relation.

Even though it is not directly specified, there is an implied communication between the *Organization* and the member *Agent* which provokes the invocation of the corresponding operations for *membership* and *memberOf* respectively. These communication protocols will be further detailed in Section 6.3.

6.1.10 AgentPlatform

In order to support the execution of a MAS, a series of basic services must be provided for the agents. These services can include storage, execution control, search facilities and, of course, message passing/delivery. Because communication is such a critical aspect, we have decided to focus on the message delivery aspect of the *AgentPlatform* and specify it as described in Definition 6.1.9 and Schema 9.



Schema 9: Class schema for AgentPlatform

Definition 6.1.9 *An **AgentPlatform** is defined as a singleton* $C = \langle agentsAndOrgs \rangle$ *where agentsAndOrgs represents the set of* Agents *and* Organizations *that populate the platform at a given time.*

The message delivery service of the *AgentPlatform* is specified in the *DeliverMessage* operation. This operation combines the *SendMessage* operation of the sender *Agent* with the *ReceiveMessage* operation of every receiver *Agent* respectively in the given communication.

With this formal view of the concepts involved in the execution of agent organizations, we introduce the JadeOrgs metamodel in the next section. The metamodel provides a further detailed representation of the abstract concepts presented in this section.

6.2 The JadeOrgs metamodel

The JadeOrgs metamodel is organized in packages and each of the packages represents a viewpoint of the MAS modeled. These viewpoints are:

1. *Project view* contains the main building blocks of a MAS and thus includes concepts like Agent, Behavior, Role, or Environment.

2. *Core view* describes single autonomous entities, the roles they play within the MAS and the organizations they build.

3. *Behavioral view* presents the variety of behaviours available to agents and organizations.

4. *Process view* describes how plans are composed by complex control structures and simple atomic tasks like sending a message and how information flows between those constructs.

5. *Ontology view* is a formal representation of the knowledge the agents possess. It consists of a set of concepts within a domain and the relationships between those concepts.

6. *Deployment view* shows how the agent and organization instances are initially configured in the system.

6.2.1 The JadeOrgs Project View

The Project View is the most general of the metamodel viewpoints. All the components of the MAS, namely the *ProjectElements*, are contained by the *Project* concept. As depicted in Figure 6.1, the basic component classes from all the other viewpoints are specializations of ProjectElement.

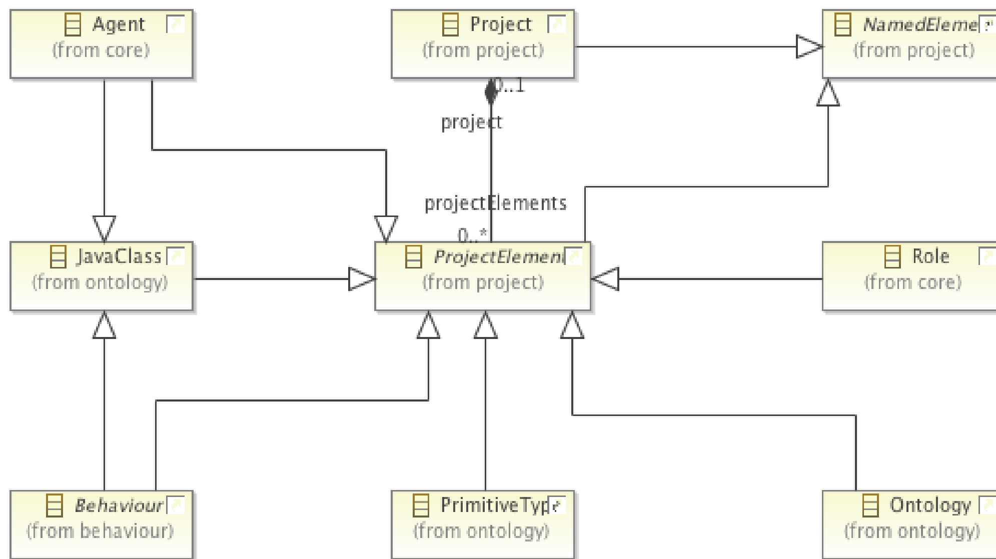


Figure 6.1: View of the Project package of the JadeOrgs metamodel

6.2.2 The JadeOrgs Core View

The definition of *Organization* that we propose in JADE is the *Agentified Group* from [ONL04]: a group of agents that, as a unit, has the same features and interaction abilities as a single agent. For example, just as an agent, it can send and receive messages directly and take on roles. For this purpose, an *Organization* is a specialization of *Agent*. Therefore, the JadeOrgs metamodel is centered around three concepts/classes: *JadeOrgsAgent*, *Organization* and *Role* (Figure 6.2).

The *JadeOrgsAgent* is a extension of JADE's *Agent* class that provides the data structures and methods necessary to manage the agent's membership information in whatever organizations it has joined, as well the list possible

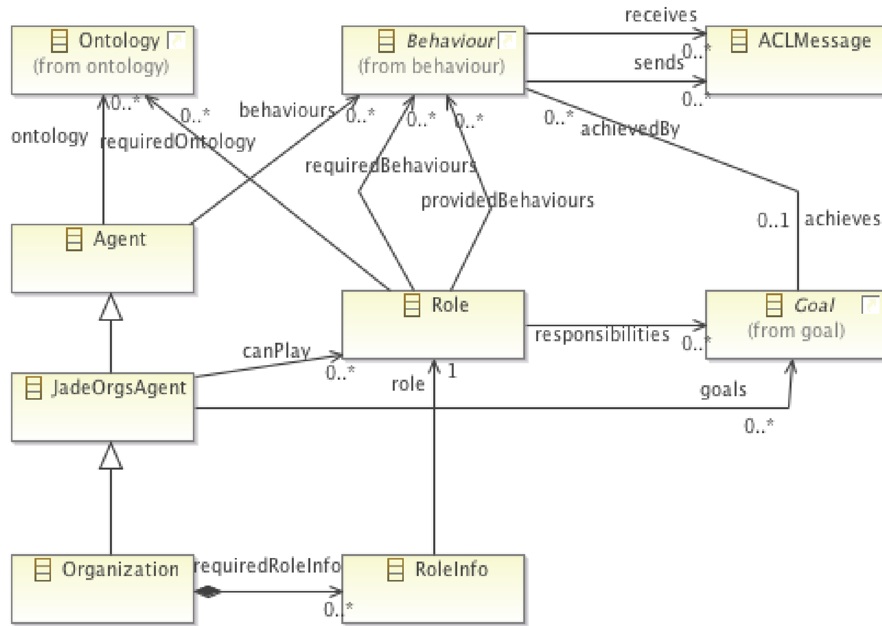


Figure 6.2: The Core of the JadeOrgs metamodel

roles the *JadeOrgsAgent* can play. In the first implementation of JadeOrgs [MMLSF08], we had intended to use the *Agent* class of JADE directly and just provide some auxiliary classes to manage this membership data. The intention being to allow existing system implementations to add this functionality without changing their agent types hierarchy. However, it proved most practical to just provide this functionality as a specialization of *Agent*.

The *Organization* class contains the information about its members, as well as the Roles under which the membership relation is stated. The information about the required roles is represented by the association class *RoleInfo*, which provides the cardinality information with respect to the amount of role filler agents required. The *Organization* class extends the *Agent* class, given that we want it to be able to perform tasks and communicate with its members and other agents. As such, the *Organization* is itself an *Agent* and possesses its own set of behaviors. Additionally, *Organization* also provides the functionality of registration and deregistration of members as the Organization changes over time. These tasks are performed using communication protocols and will be described in more detail further on.

The *Role* class is implemented as an *Ontology Concept*, part of the *OrganizationOntology*. Among the properties of the *Role* class, we find responsibilities, required and provided behaviors, and required ontologies. Each of these properties allow different requirements to be checked and depending on the evaluation strategy desired only some of them may be specified. The responsibilities represent the goals that the agent should achieve when performing the *Role*. In the case that goals are not used to model the responsibilities, the list of required behaviors can be used to verify that the agent is actually capable of performing the Role's tasks. The provided behaviors allow the Agent to acquire additional *Behaviors* required to fulfill the role by adding them to its known behaviors. In a similar fashion, the required ontologies allow an evaluation of the knowledge available to the agent.

6.2.3 The JadeOrgs Behavioral View

The *Behaviour* class[1], previously shown in Figure 6.2, represents the root element of the *Behavior* hierarchy. The behaviors are divided in two groups: simple and complex behaviors. The simple ones perform smaller, atomic tasks, while the complex behaviors allow the nesting of behaviors and permit different ways of executing them. In Figure 6.3, a partial view of this behavior hierarchy is presented. The behaviors marked with a darker color are provided by JADE, while the lighter color ones are part of the JadeOrgs library.

Under the group of complex behaviors, the *FSMBehaviour* and its specializations are very relevant for our purposes. The *FSMBehaviour* is an implementation of a Finite State Machine (FSM). In this FSM, the states and actions are represented by behaviors and the transition function between the states is represented by a list of *Transition* objects, as illustrated in Figure 6.4. The triggering event for the transition is obtained represented by an integer value and the transition is fired when the source behavior returns the same value on its onEnd() method upon completion of the execution of the action. The *FSMBehaviour* enables the modeling of complex behaviours such as the implementation of protocols. JADE already takes advantage of this implementing the Initiator-Responder pattern for FIPA interaction protocols using *FSMBehaviours*. For example, the *AchieveREInitiator* and

[1]The names of the behavior concepts in the metamodel use the UK spelling of the word behaviour, since that is the convention in JADE.

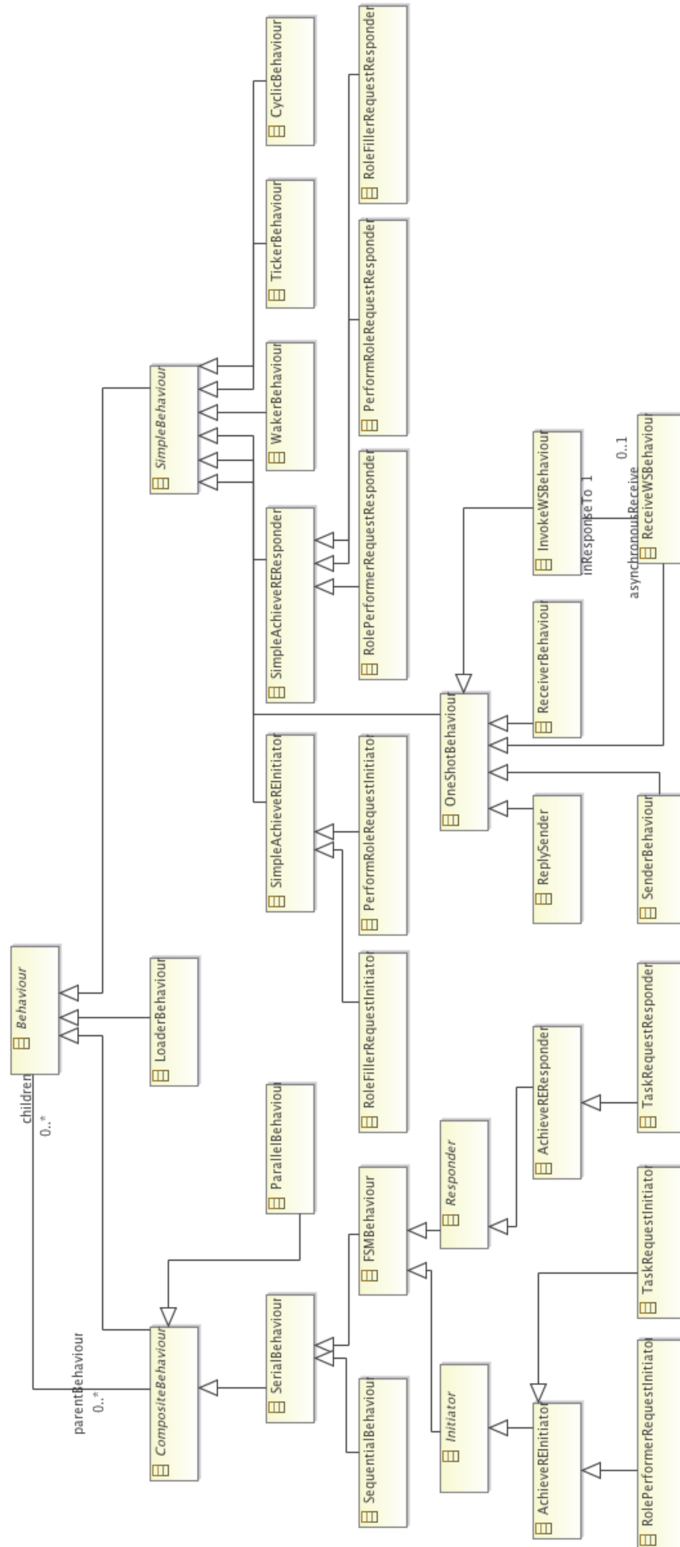


Figure 6.3: Partial view of the Behavior class hierarchy

AchieveREResponder are provided to implement all the FIPA-Request-like interaction protocols defined by FIPA [Fou02d] in which the initiator sends a single message to a responder in order to verify if the rational effect (RE) of the communicative act has been achieved or not. As shown in the Figure 6.4, we have extended some of these behaviors for the organization establishment and task delegation processes, presented in Section 6.3.

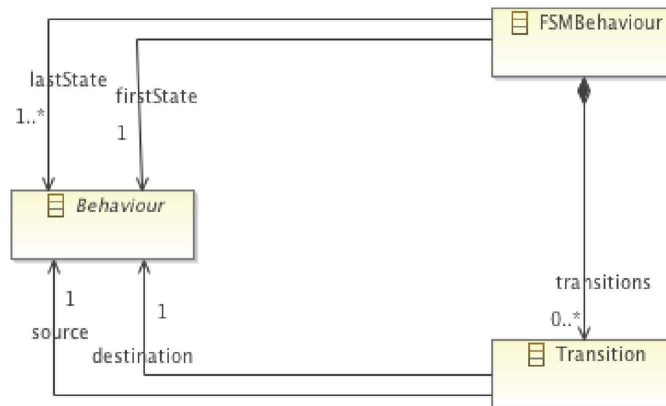


Figure 6.4: Representation of the FSMBehaviour

6.2.4 The JadeOrgs Process View

In order to model the processes that occur inside behaviors and methods, a basic set of constructs was chosen for the Process package. The set of constructs chosen is small given that it was not our intention at this time to reproduce the structure of the complete target language, Java, but still provide some process modeling capabilities.

Shown in Figure 6.5, the Process package is composed by a set of statements and a *CodeBlock* concept. The *CodeBlock* is a sequence of statements to be executed in the block. The statements available in this view are:

Decision: a basic if-then-else statement. When the condition evaluates to true, the mandatory *then* CodeBlock is executed; otherwise the optional *else* CodeBlock executes.

WhileLoop: a basic loop statement. While the *condition* holds, the *codeBlock* attribute is executed.

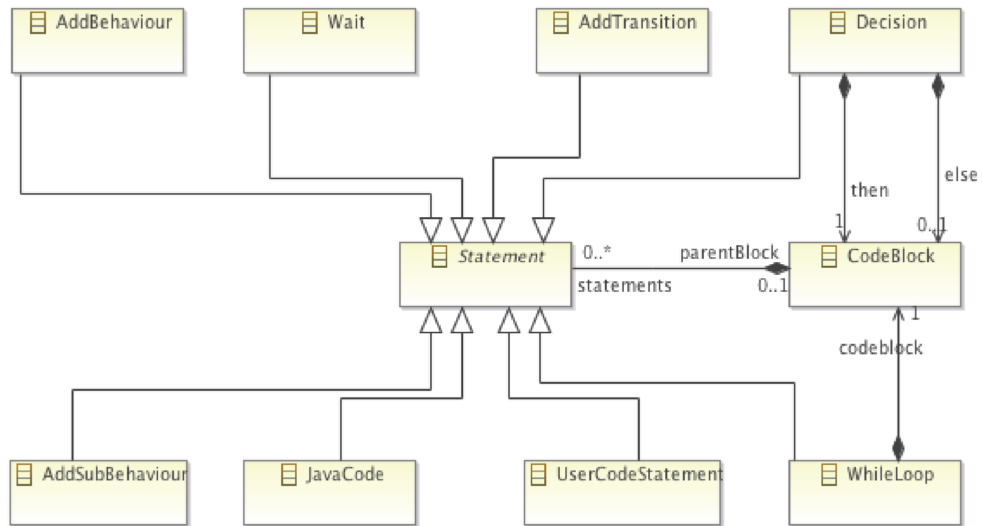


Figure 6.5: The JadeOrgs Process package

Wait: a statement that causes the process to wait for a fixed amount of time.

AddBehavior: a statement which adds a given behavior to the agent's scheduler.

AddTransition: adds a transition object to a given FSMBehaviour.

UserCodeStatement: an auxilary construct that is serialized as a "to do" comment in the target code to indicate that the programmer needs to provide further implementation.

JavaCode: a block of Java code that is serialized "as is" when the model is serialized in the target code.

6.2.5 The JadeOrgs Ontology View

The JadeOrgs Ontology View enables the specification of the language used by the agents in the content of the *ACLMessages* and the representation of its internal beliefs. In addition, primitive and class datatypes can be specified. Figure 6.6 presents an overview of the concepts related to this viewpoint.

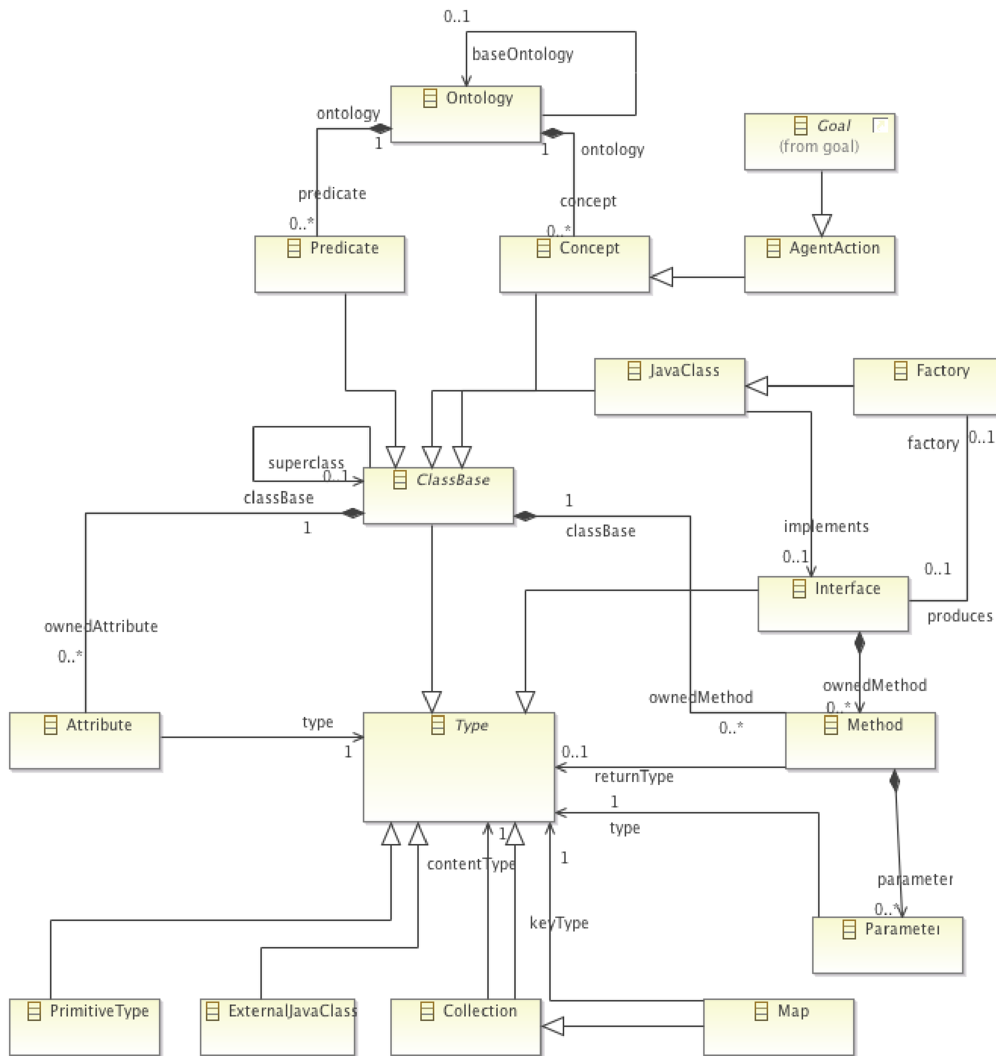


Figure 6.6: The JadeOrgs Ontology

An *Ontology* in JadeOrgs is composed by *Concepts* and *Predicates*, and it describes the knowledge that the agent can posses about a given domain. *Concepts* are entities with complex structures that are defined in terms of slots or attributes. *Predicates* are expressions that say something about the status of the world and can have a true or false value. *AgentActions* are concepts that represent activities that the agent may perform or that an agent may request others to perform.

Concepts are also a specialization of the abstract concept *ClassBase*. The *ClassBase* classifies entities that possess a set of methods and attributes with their corresponding parameters and types. The *ClassBase* structure also enables the introduction of external java classes into the agents knowledge through the use of the JavaClass concept. Java *Interfaces* and the *Factory* design pattern are also directly supported in the metamodel.

In order to support groups of items, the *Collection* concept is used. A *Collection* is a list of items of a given *contentType*. In order to grant indexed access to a *Collection*, we introduced the *Map* concept. It denotes a list of content items indexed by *keyType*.

As previously mentioned, the items modeled in this view build up the formal representation of the knowledge the agents possess. As such, they also represent the content of the messages to be exchanged among the agents. It is also important that the concepts used as message content follow the semantics of ACL speech acts [Fou02b]. For instance, concepts that will be used as content for a message using the *request* performative should be of the *AgentAction* type.

Following the semantics of *request*, a *Goal* is introduced as a specialization of *AgentAction*. *Goals* are declared through the use of conditions that describe the desired state of affairs that the goal represents. Following the goal framework presented in [vRDW08], we specialize goals into four types as depicted in Figure 6.7:

PerformGoal indicates a procedural goal to execute a given action,

AchieveGoal denotes a declarative goal in which a desired state of affairs that should be reached,

QueryGoal describes a declarative goal that pursues a desired state of affairs in which a piece of information is available,

MaintenanceGoal restricts the set of valid states of affairs. In other words, if the maintenance condition is broken, the *MaintenanceGoal* triggers a behavior or behaviors that should reestablish the broken condition.

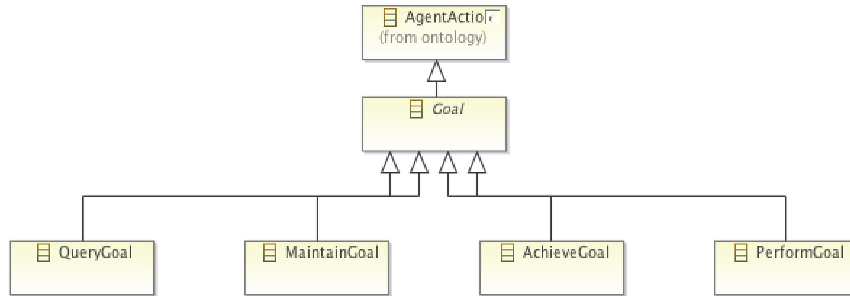


Figure 6.7: The JadeOrgs Goals

In [vRDW08], the authors argue that these type of goals are "individual goals" and not organizational goals. Nevertheless, we take this goal hierarchy and apply it also to organizational goals in order to have the same execution semantics once the "achieve goal" request is received by a member and, respecting its autonomy, it accepts to achieve this goal for the organization. Since the goals that can be requested to be achieved by an organization member are specified in the role's *responsibilities* and they should match the member's own goals, it is assumed the member should be able to achieve the requested goal.

It is not within the scope of this work to provide JADE with a full goal-driven execution. Therefore we only implement *Goals* as means to represent a desired action in a declarative way. Frameworks such as JADEX [PBL05] already provide a way to integrate BDI goal execution with JADE.

6.2.6 The JadeOrgs Deployment View

In addition to the definition of the agents and organizational types, it is necessary to represent how instances of these will be related to one another when the MAS is initialized. We therefore introduced the Deployment View, depicted in Figure 6.8, to allow the modeling of an organization composition at design time.

The *AID* class represents JADE's agent identifier. We extend the *AID* class into 2 types: *AgentInstance* to represent regular JADE agent instances

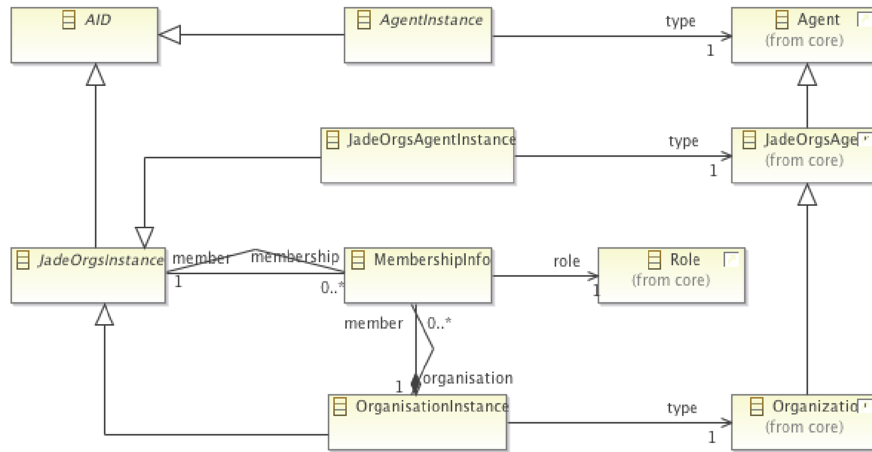


Figure 6.8: The Deployment View in JadeOrgs

and *JadeOrgsInstance* to represent instances of *JadeOrgsAgents* and *Organizations*. In this view, *MembershipInfo* is the association class that binds the types together to express that a *JadeOrgsInstance* is member of an Organization under a given *Role*.

6.3 JadeOrgs protocols and interactions

As part of the JadeOrgs run time querying, a set of protocols to support the organization's activities have been implemented into JADE agent behaviors. These protocols are described in the following.

6.3.1 Publishing to the Directory Facilitator

The structure of the *Organization* can be established at design time or at run time. For those which are setup at design time, the initialization of the organizational structure is already set; however, for those that are not determined until run time, a set of role fillers has to be selected. JADE already provides a directory service called the Directory Facilitator (DF). Through the DF, an agent/organization can search for other agents/organizations that possess a given set of features, such as the protocols supported or the ontologies it can access.

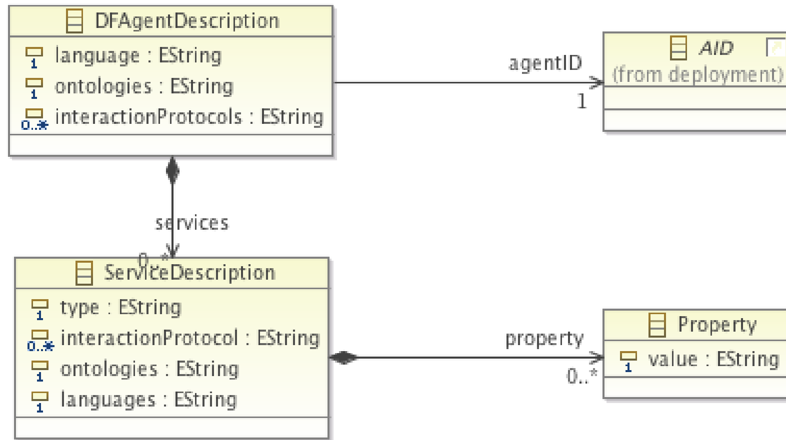


Figure 6.9: Agent Description in JADE

In order to take advantage of the DF Service, we use the *DFAgentDescription* class, an agent descriptor which is part of JADE's *FIPAAgentManagementOntology*. The structure of the DFAgentDescription is depicted in Figure 6.9. In order to take advantage of the directory infrastructure that JADE provides, we describe the roles as services using JADE's ServiceDescription. These "role services" are typed as either `requiredRole` and `playedRole`. Additional role attributes that do not match directly with attributes of the ServiceDescription, such as required behaviors, are mapped to Property elements.

6.3.2 Establishment of the Organization

Once the description of an *Organization* and members is published to the DF, the establishment of the organization can take place at run time. As a first step, a search for suitable agents/organizations is performed by quering the *DF Service*. When the list of prospective *DFOrganizationMemberDescriptions* or *DFOrganizationDescriptions* is retrieved, the agent/organization initiates the RoleFillerRequest protocol with the organization it wants to join. The protocol is implemented by the *RoleFillerRequestInitiator* and *RoleFillerRequestResponder* behaviors. As described in Figure 6.10, the organization takes the *Responder* role and a *RoleRequest* object is sent by the *Requester* as content of the ACL_request message. Once this request is received by the *Responder*, an ACL_refuse message is produced if the request

is denied, or an ACL_inform message is produced if the request has been accepted. As can be expected, the decision process for accepting/denying these requests is left to other internal behaviors of the agent/organization. An analogous protocol can be applied for the organization that wants to recruit a new member.

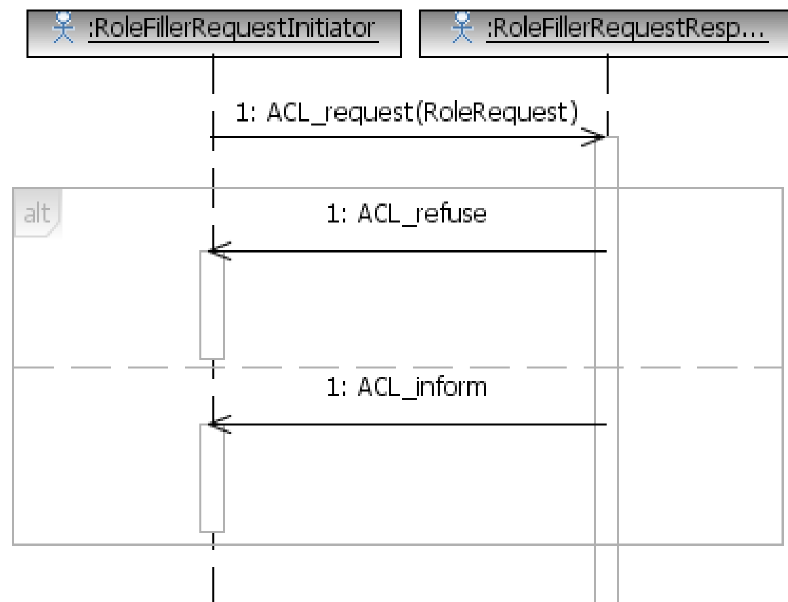


Figure 6.10: The RoleFillerRequest protocol

Depending on the design policies, the decision process may include, for example, a verification of the features of an agent with respect to the requirements specified in the Role. The Role description allows various evaluation options that can be extended and customized:

Type compatibility: the canPlay association between JadeOrgsAgent and Role permits a simple check through the type/class definition,

Responsibilities: the goals association between JadeOrgsAgent and Goal allows to check if the agent's interests fit the responsibilities in the role.

Required Behaviors: if goals are not used to represent responsibilities, the set of required behaviors can be used to ensure that the agent "knows *how* to perform a given task", and

Additional Qualifications: additional conditions can be checked by simply extending the provided behavior classes.

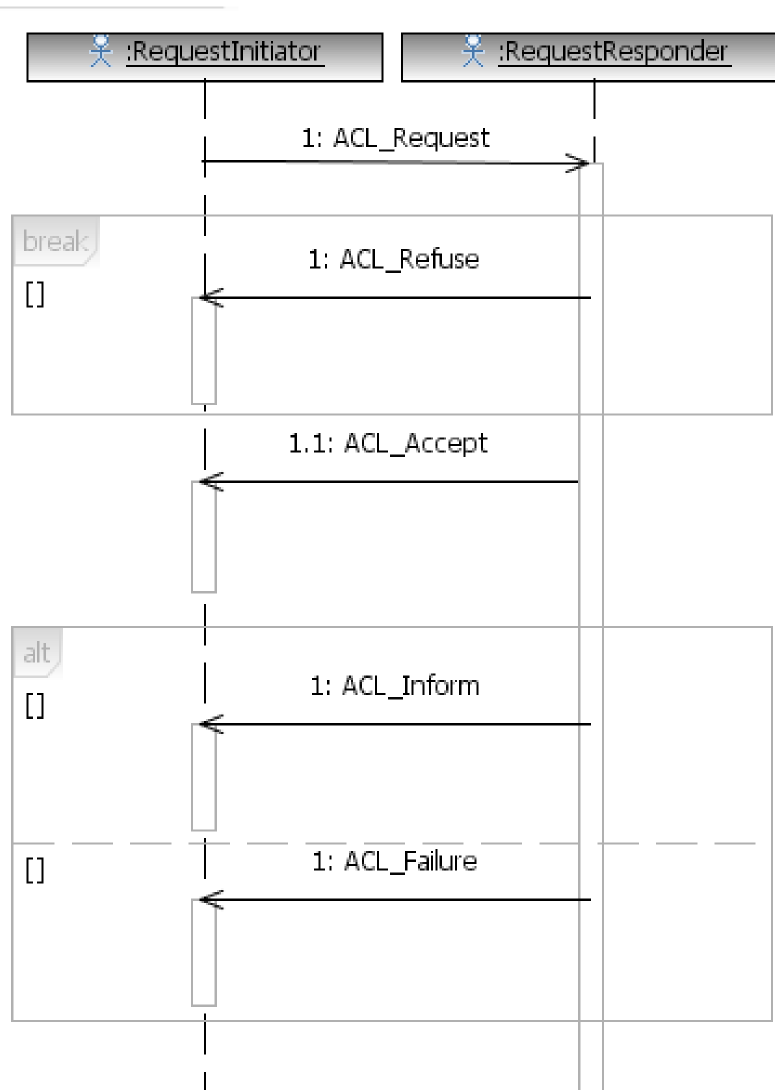


Figure 6.11: The TaskRequest protocol

6.3.3 Task Request and Goal Achieve

In order to give the members of the organization the right to manage their own work load, the distribution of tasks is performed through the simple protocol presented in Figure 6.11. This protocol is a simplified version of the FIPA Request Protocol [Fou02d] which provides the *RequestResponder* with the option of refusing in case it is already busy.

The protocol is implemented through the *TaskRequestInitiator* and *TaskRe-*

questResponder behaviors previously shown in Figure 6.3. As part of the *TaskRequestInitiator* behavior, the behavior must implement a mechanism for choosing the desired role fillers for the task out of the set of available members under the role that should perform the task. For the coding of the mechanism, an interface called *RoleFillerChooser* has been provided.

6.4 Small Example: Product Sale with Loan

As a concrete example on how Organizations can help to define the interaction context, we present a Product Sale scenario. The basic interaction in this scenario takes place between a *Buyer* and a *Seller* and it is depicted in Figure 6.12. The interaction is initiated by the Buyer making a query about a certain product. If the product is not in stock, the Seller sends an OutOfStock message and the interaction terminates. If the product is in stock, the Seller replies with the product price. The Buyer receives the price and considers if it has enough money to pay for it. If it does not, the Buyer usually cancels the transaction. If it does have the money, it sends the payment to the Seller and, correspondingly, the Seller ships the product.

We can extend this behavior by saying that if the Buyer does not have enough money, it has to find the means to get the necessary money. One solution, would be to get a loan from a Bank. This situation could be modeled in JadeOrgs with the organization, agent and roles types as depicted in Figure 6.13. On the right hand side of the image, the *Store* organization is the one that contains the ProductSale interaction previously described. It has two roles *StoreCustomer* and *Cashier*, and they require the behaviors that implement the Buyer and Seller described in the protocol respectively. The Cashier role can be played by the *StoreClerk* agent type and the StoreCustomer role can be played by the *MyAgent* agent type.

On the left hand side of Figure 6.13, we find the *Bank* organization, with its two roles: *BankRepresentative* and *BankCustomer*. The BankRepresentative role can be played by agents of the *BankClerk* type and requires the LoanApplicationLoaner behavior. Correspondingly, the BankCustomer role requires the LoanApplicationCustomer behavior and can be played by the MyAgent agent type. As can be deduced from the required behaviors, one of the possible interactions inside the Bank organization is the LoanApplication protocol (Figure 6.14). In this protocol, a Customer sends a loan application to Loaner; the Loaner evaluates the application and determines if the

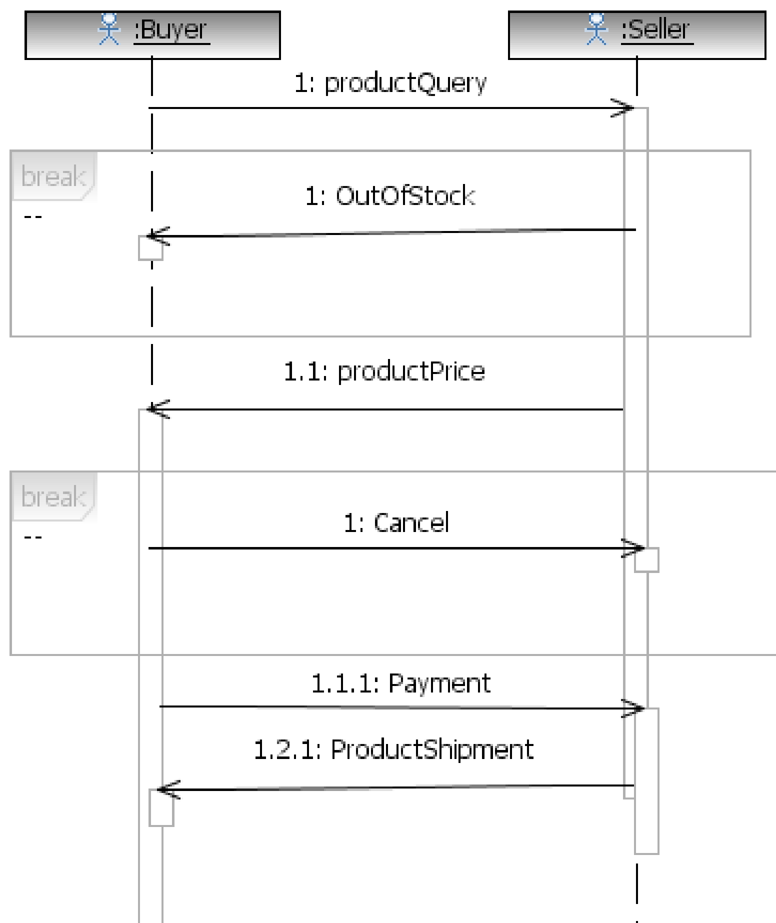


Figure 6.12: The Product Sale Protocol

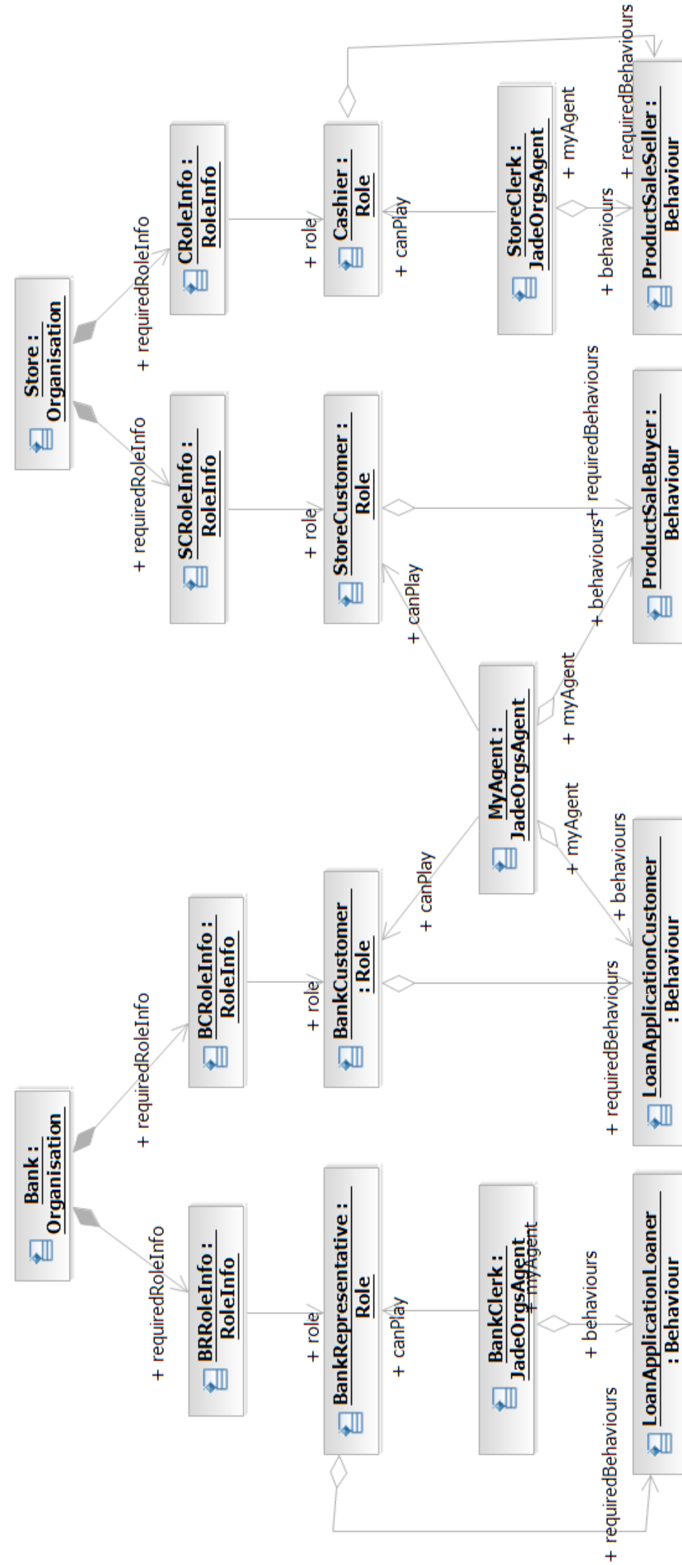


Figure 6.13: Organizational structures for the Product Sale scenario

Customer qualifies for a loan. If he does not, the interaction is over with a rejection message. But if he does, he receives a confirmation of the acceptance of his application and he must then sign the loan contract, after which the Loaner deposits the loaned amount in the Customers account.

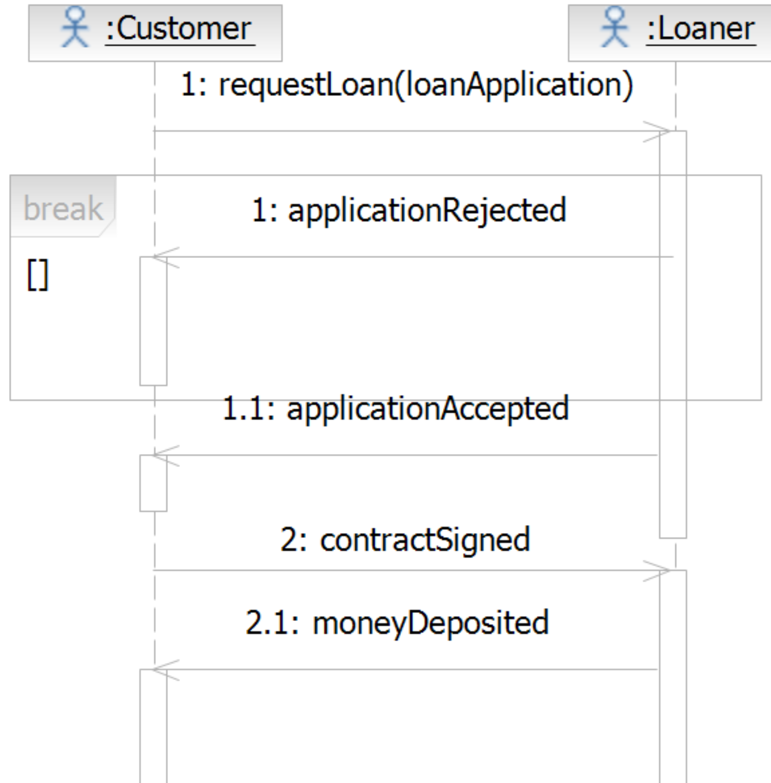


Figure 6.14: Loan Application protocol

Once the structures and interactions have been established, we have to define the instances. In Figure 6.15, we can see the initial state of the scenario. The agent instance *John* is member of the *BargainElectronicStore* organization instance under the StoreCustomer role. The BargainElectronicStore also has the agent instance *Marie* as a member playing the Cashier role. In this context, the previously described situation occurs: John wants to buy a product but does not have enough money to pay Marie the required amount.

At this point, let us assume that agent John was provided with a behavior that specifies ways to obtain money and determines that if there is a Bank organization in the environment, it could apply for a loan since its canPlay

property indicates that it can play the BankCustomer role. Therefore, John queries the DF to find any Bank organizations in the system and retrieves the description for *Bank123*, since Bank123's DFOrganizationDescription indicates that it requires the BankCustomer role. With Bank123's identifier, John initiates the RoleFillerRequest protocol and joins the organization since it meets the requirements for role type and required behaviors.

As a BankCustomer of Bank123, it can initiate the LoanApplication protocol with a BankClerk. Bank123 assigns this task to BankClerk instance *Peter*. Since John provides a good credit history in the application, Peter approves the loan. Once John has received the credited money, it activates its StoreCustomer role again in BargainElectronicStore and initiates the ProductSale protocol with Marie. This time John succeeds and obtains the product.

6.5 Related works

In this section, we first compare JadeOrgs (and JADE) to the approaches of some known agent-oriented methodologies.

6.5.1 Metamodel comparison

In order to evaluate the concepts and properties of the JadeOrgs metamodel, we compare it using a subset of the features of the AOSE Methodology evaluation questionnaire from the Agentlink III AOSE TFG [Cos05b]. In this questionnaire, a set of methodologies was evaluated with respect to concepts/properties, notation, modeling and lifecycle coverage. At this stage, we have chosen to evaluate only the coverage of the concepts and properties. JadeOrgs only provides a metamodel with runtime library and not a complete methodology, so most of the questions of the notation, modeling and lifecycle sections did not apply to our approach. The evaluation results are presented in Table 6.1, the two rightmost columns present our answers to the questionnaire for JadeOrgs and JADE in order to provide a baseline and show how JadeOrgs has extended JADE. We do not present the result of all the methodologies presented in the AOSE TFG results presentation [Age05], but only the ones that presented society structures and role concepts, namely Gaia [WJK00], Ingenias [PGS03], PASSI [Cos05a] and TROPOS [BPG+04]. We are aware that a questionnaire provides very subjective results, as was

Figure 6.15: Instance distribution of the scenario (initial state)

Concept/ Property	Gaia	Ingenias	PASSI	TROPOS	JADE	JadeOrgs
Autonomy	H	H	H	L	H	H
Mental attitudes	N	H	L	M	L	L
Proactiveness	L	H	H	NN	L	M
Reactiveness	L	H	H	NN	H	H
Concurrency	M	H	H	L	H	H
Teamwork and roles	H	H	M	M	L	H
Cooperation model	Teamwork	ALL	Task del., Teamwork	Negotiation	NN	Task del.
Protocols support	H	H	H	NN	NN (projected behaviors)	NN
Communication modes	Async	Direct, Indirect, Synch., Asynch.	Direct		Direct, Synch., Asynch.	same as JADE
Communication language	ACL like	Speech acts, Signals	Speech acts		ACL	ACL
Situatedness	H	H	H	H	L	L
(Main) Supported agents	-	BDI (mainly)	Mainly: State-based, rational, reactive	BDI, Rational	Reactive, Rational*	same as JADE
Society of agents modeling	SA	SA	A/-/-	A	D	SA
Society structure	-	Organizations, Groups	p2p, simple hierarchies, holons	Broker, Mediated, Matchmaker	-	Organizations

NN: None, M: Medium, L: Low, NA: Not Applicable

SD: Strongly Disagree, D: Disagree, N: Neutral, A: Agree, SA: Strongly Agree

Table 6.1: Concept/Property Comparison (based on [Age05])

also noted in [Age05]. Nevertheless, this comparison can provide an intuition where JadeOrgs stands with respect to other agent languages.

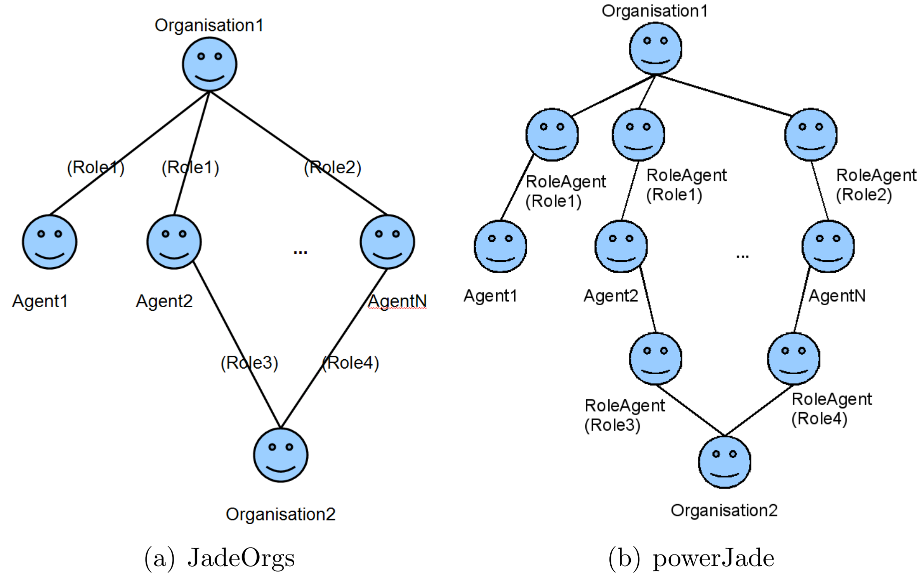
As apparent from the comparison table that JadeOrgs has inherited certain weaknesses from JADE with regard to situatedness and protocol support. These properties can, of course, be implemented in the agent code, but they should definitely be considered as additional “first order citizens” in JadeOrgs.

JadeOrgs improves on JADE with an explicit cooperation model (task/goal delegation), proactiveness (through organizational goals) and a society structure. Our approach to organizations with roles is general enough to be able to express the society structures that other approaches use: hierarchies, holons, groups. For example, organizations in Ingenias are composed by groups that perform certain tasks or achieve certain goals. The same structure can be represented in JadeOrgs by creating a suborganization in place of the Ingenias subgroup.

6.5.2 Other approaches to runtime organizations in JADE

The only approach that directly addresses the issue of implementing organizations as entities in JADE is powerJade [BBG+08]. The powerJade approach has various similarities with JadeOrgs with the main difference in the implementation of roles. On the one hand, powerJade Organizations are implemented as an extension to the JADE agent class and possess all the information with regard to the members of the organization, analogous to JadeOrgs. On the other hand, roles in powerJade are implemented as agents since they are the ones in charge of performing powers and requesting requirements from the agents that play the roles: the players. Powers are the actions that a given role can perform in the system, while requirements represent the information that a given power needs in order to be performed. When a player fails to meet a requirement, its role is “deacted”. Therefore, powerJade does not evaluate the role requirements when the players joins the organization, but deacts the role if one of the requirements is not met during the execution of a Power.

The potential drawback of having runtime organizations is the increase in computational resources necessary for the overall MAS, since the number of agents in the system will increase. This increase can depend directly on how



(a) JadeOrgs (b) powerJade

Figure 6.16: Example of organization structure instances: Two Organizations bound to N Agents through 4 roles

the relation between organizations and role players are implemented. To illustrate this, let us take the example depicted in Figure 6.16 to compare how this relation is implemented in JadeOrgs and powerJade. As previously mentioned, roles in JadeOrgs are a piece of knowledge that the agent/organization describing the requirements for the role players. In powerJade, the roles are implemented as a role agent that executes the powers for the role player for each role that the role player assumes. Therefore, the total number of agent instances in the system A for JadeOrgs is $A_{JadeOrgs} = m + n$ where n is the number of role player agent instances and m is the number of organization instances. For powerJade, $A_{powerJade} = m + n + \sum_{i=0}^{n} r_i$ where r_i is the number of "role agent instances" linked to each role player. The powerJade implementation of roles can be advantageous in the sense that the role player has less concerns about interacting with the organization, for instance it is not required to know the organization's ontology/language since the "role agent" can serve as a proxy. In JadeOrgs, the coordination of the role players is done by the organization directly and if such a proxy was necessary, it could be implemented as an additional role or a suborganization could be created to group the proxy with the represented agent.

6.6 Summary

The explicit representation of an Organization as a first level entity is usually missing in multiagent system platforms. Most of the time it is left as a result of the emergent behavior of interacting agents. This is also the case for JADE, one of the most frequently used multiagent system platforms. But in general this is unsatisfactory as some structuring is essential for conceptual reasons but also for efficiency reasons. Therefore the concept of an organization is proposed for this platform as a specific kind of agent. The fact that it is represented by an agent and not left as a virtual manifestation result of individual behaviors opens new options for modeling collaborations. Interaction protocols can be more easily modularized and, by scoping the aspects in complex interactions, the predictability, reliability, and scalability of such distributed systems are increased.

Having a concrete representation entity for an organization also facilitates the definition of the policies, by making them explicit instead of implicit. Organizations provide not only advantage for design time, but also for enabling dynamic establishment of organizations at run time. For this dynamic establishment, we have presented evaluation methods for candidate members as well as the protocols that support the establishment process.

When compared to other systems, JadeOrgs reaches an adequate coverage of the design and implementation spectrum. It allows to model organizational structures in rich, detailed fashion, while also providing a run time extension that is easy to use for developers working with JADE.

Chapter 7

Transforming PIM4Agents into JadeOrgs

This chapter introduces the mapping from the PIM4Agents concepts (Chapter 5) to the JadeOrgs concepts presented in Chapter 6 through a series of mapping rules. The list presented does not comprehend all the necessary model mappings, but only the most relevant for a better understanding of how they are applied in order to produce a JadeOrgs model. We will also present briefly how the case study model presented in Section 5.10 is mapped into a JadeOrgs model.

7.1 The Mapping Rules

Transformation 1:

Head: $PIM4Agents : Agent \rightarrow JadeOrgs : Agent$

Body: Every Agent in the PIM4Agents is mapped to a *JadeOrgs:Agent*.

The Mapping Rule 1 is fairly straight forward, given that the concepts correspond to one another in the use of behaviours, to carry actions; Roles, to represent responsibilities or compromises; and Organizations, to collaborate with other Agents.

Transformation 2:

Head: $PIM4Agents : Organization \rightarrow JadeOrgs : Organization$

Body: *JadeOrgs:Organization*, an extension to the JADE API, allows to transform *PIM4Agents:Organization* in the straightforward fashion.

The concept of an Organization in the PIM4Agents is mapped directly to JadeOrgs:Organization, since the concept in JadeOrgs is a custom made extension to the JADE API. Therefore, its properties are mainly mapped in a one-to-one fashion.

Transformation 3:

> **Head:** $PIM4Agents : Goal \rightarrow JadeOrgs : Goal$
>
> **Body:** Each of the four types of *PIM4Agents:ConcreteGoal* can be transformed into a *JadeOrgs:Goal* in the straightforward fashion.

The four types of concrete goals in PIM4Agents have a direct one-to-one correspondent in JadeOrgs. Therefore all the properties can be mapped directly and the goal conditions are mapped as evaluation methods in order to, for instance, determine if the goal has been achieved. In addition, when goals are present the model, the Agents will possess the goal handling and delegation behaviors that they require.

Transformation 4:

> **Head:** $PIM4Agents : Protocol \rightarrow JadeOrgs : FSMBehaviours$
>
> **Body:** The *PIM4Agents:Protocol* is decomposed into *n FSMBehaviours* types–one for each Actor in the Protocol—whose execution order is determined by the *PIM4Agents.MessageFlow* for the corresponding Role.

Transformation Rule 4 is a much more complex mapping than the ones presented so far. It basically does a collapse of the 'MessageFlow graph' and links the *PIM4Agents:MessageScopes* that correspond to each *PIM4Agents:MessageFlow* in the PIM4Agents into a set of *JadeOrgs:FSMBehaviours* in the JadeOrgs, whose transitions depends on the graph's links. The *PIM4Agents:MessageScopes* should go into the each of the *JadeOrgs:FSMBehaviours* depends on the *Actor* in the PIM4Agents to which they belong.

Transformation 5:

> **Head:** $PIM4Agents : Actor, PIM4Agents : DomainRole \rightarrow JadeOrgs : Role$
>
> **Body:** *Actors* and *DomainRoles* are mapped to *JadeOrgs:Role.*

The Actor transformation (Mapping Rule 5) also performs a collapse of the 'MessageFlow graph', but in this case, it groups the incoming and outgoing Messages found in the graph with respect to the Actor. Additionally, the Actors are unified with the DomainRoles through the DomainRole.binding property. Therefore, there is only one Role concept in JadeOrgs which models the Actor and DomainRole concepts.

Transformation 6:

> **Head:** $PIM4Agents : Activity \rightarrow JadeOrgs : Behaviour$
>
> **Body:** *PIM4Agents:Activity* and *JadeOrgs:Behaviour* are abstract, therefore the actual source and target for the transformation of the activities depends on their corresponding specializations.

Mapping Rule 6 represents the general rule for mapping behaviours and activities. In practice, there are several mapping rules for each particular specialization of *PIM4Agents:Behaviour* and *PIM4Agents:Activity.* Transformation rules 8 and 9 show how the different specializations for Activity are mapped into *JadeOrgs:Behaviours.*

Transformation 7:

> **Head:** $PIM4Agents : Capability \rightarrow JadeOrgs : Behaviour$
>
> **Body:** For every Behaviour referenced by a Capability in the PIM4Agents, a Behaviour in JadeOrgs will be added to the available behaviours of the Agent.

Transformation 8:

> **Head:** $PIM4Agents : StructuredActivity \rightarrow JadeOrgs : FSMBehaviour$

Body: *PIM4Agents.Behaviour:StructuredActivity* is transformed to a finite state machine (FSM). The structure of the FSM depends on, for example, whether a Sequence, Split or Loop is required.

In similar fashion to Mapping Rule 6, Mapping Rule 8 represents a series of specific rules for transforming particular specialized types of StructuredActivities. For example a Sequence in the PIM4Agents is transformed in SequentialBehaviour or ParallelBehaviour in JadeOrgs.

Transformation 9:

Head: $PIM4Agents : Task \rightarrow JadeOrgs : OneShotBehaviour$,- $JadeOrgs$:$SimpleAchieveREInitiator$,- $JadeOrgs : SimpleAchieveREResponder$

Body: Most subclasses of the Task concept are mapped into OneShotBehaviours in JadeOrgs with different Java calls in their body corresponding to the task required. In the concrete cases of the tasks ReceiveMessage and SendMessage, they will be mapped to a SimpleAchieveREResponder and a SimpleAchieveREInitiator correspondingly.

Transformation 10:

Head: $PIM4Agents : Message \rightarrow JadeOrgs : ACLMessage$

Body: *PIM4Agents:Message* is transformed to a ACLMessage in JadeOrgs with an INFORM performative as default. Depending on specific message types, other performatives may be used.

Transformation 11:

Head: $Ecore : EClass \rightarrow JadeOrgs : Concept, JadeOrgs : JavaClass$

Body: Each *Ecore:EClass* used in a message is transformed into a Concept with the corresponding slots depending on the EClass and the ones not used for communication are mapped as regular Java classes.

Transformation 12:

Head: $PIM4Agents : AgentInstance \rightarrow JadeOrgs$:- *JadeOrgsInstance*

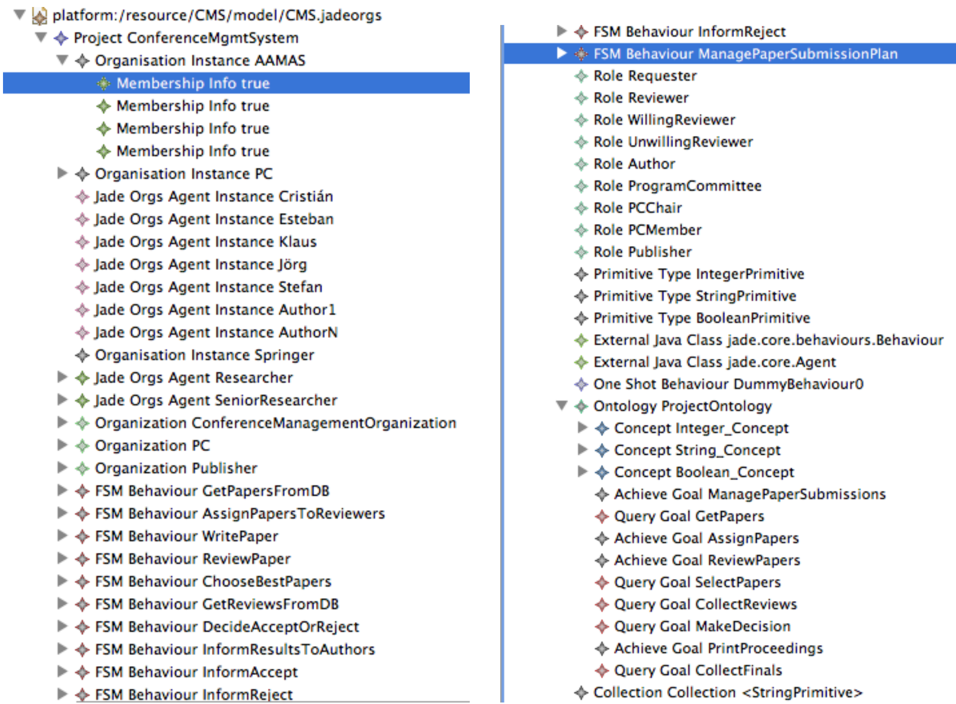
Body: Each *PIM4Agents:AgentInstance* is mapped to a corresponding *JadeOrgs:JadeOrgsInstance*, namely a *JadeOrgs:-JadeOrgsAgentInstance* when it is typed by an agent or a *JadeOrgs:-JadeOrgsOrganizationInstance* when typed by an organization.

7.2 Generated JadeOrgs models

This section presents an overview of the model produced by the transformation to JadeOrgs. At this stage, we do not count with a graphical editor for JadeOrgs to present the models in graphical form. Therefore, the models in this section are presented using the Ecore Tree Editor.

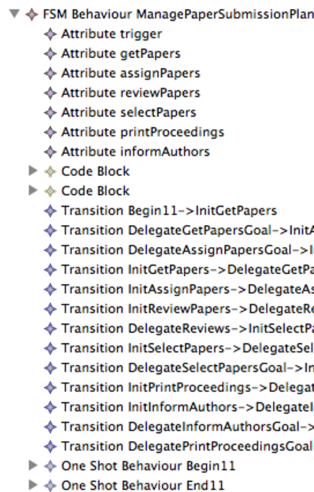
A partial view of the target JadeOrgs model is provided in Figure 7.1(a). It shows the concepts presented in Figures 5.18 and 5.17 after applying Transformations Rules 1 and 2 was applied to the *PIM4Agents:Agents* Researcher and SeniorResearcher and to the *PIM4Agents:Organizations* ConferenceManagementOrganization, PC and Publisher correspondingly. It is also visible that the instance information from Figure 5.22 has been transferred to the JadeOrgs model after the execution of Transformation Rule 12.

In the same view, we find a series of *JadeOrgs:FSMBehaviors* derived from the *PIMAgents:Plans* via Transformation Rule 8. Each of the states of the finite state machine in the *JadeOrgs:FSMBehaviour* is represented by another *JadeOrgs:Behaviour*. The transitions between each of the states are also generated following the *Flows* from the PIM4Agents *Plan* or *StructuredActivity*.



(a) Agents, Organizations, Behaviours and Instances

(b) Roles and Ontology



(c) The ManagePaperSubmissionPlan FSM Behaviour

Figure 7.1: Partial views of the JadeOrgs CMS model

Figure 7.1(c) illustrates the output of transforming the ManagePaperSubmission Plan, previously presented in Section 5.10.7.

The transformation to our target JadeOrgs model also preserves the goal and role information provided in the PIM4Agents model (cf. Figure 7.1(b)). Transformation Rule 3 generates the corresponding goals in JadeOrgs and adds them to the *ProjectOntology*. This ontology contains all the concepts that the agents in the system use for communicating or representing their beliefs. The roles generated via Transformation Rule 5 are also shown in Figure 7.1(b) and they are linked to the organizations and agents through the *MembershipInfo* objects visible under each organization in Figure 7.1(a).

7.3 Code Serialization

Once the model transformation between PIM4Agents and JadeOrgs has been executed. It is necessary to generate the Java code that will make use of the JADE agent platform and the JadeOrgs runtime library.

```
texttransformation Method2Java (in jade:"http://www.dfki.de/pim4agents/2009/jadeorgs") {
  jade.Method::generateMethodCode(){
        var is_abstract : Boolean = self._getFeature("abstract");
        if(is_abstract){
            self.generateAbstractMethodCode();
        }
        else{
            self.generateNotAbstractMethodCode();
        }
  }

  jade.Method::generateNotAbstractMethodCode(){
        var return_type_string : String;
        if(self.returnType != null)
          return_type_string = self.getTypeStringofParameterType(self.returnType)
        else
          return_type_string = "void";

        if(self.visibility = 'package'){
            tab(1) return_type_string <% %> self.name <%(%>
        }
        else{
            tab(1) self.visibility <% %> return_type_string <% %> self.name <%(%>
        }
                self.parameter->forEach(p : jade.Parameter){
                    self.getTypeStringofParameterType(p.type) <% %> p.name
                    if(self.parameter.last() != p){
                        <%, %>
                    }
                }
                <%){\n%>
                self.body.statements->forEach(s : jade.Statement){
                    tab(2) s.map2JavaCode()
                    newline(1);
                }
        tab(1) <%}%>
  }
```

Figure 7.2: Extract from the Method2Java text transformation

In order to serialize our JadeOrgs models, we created a series of code gen-

eration templates using MOFScript [SIN06]. MOFScript enables to navigate a MOF-based model and fill the values into our code template and therefore generate the corresponding Java class code. As shown in Figure 7.2, the text transformation rules allow the generation of the target code and even manipulate format aspects such as the code tabulation.

7.4 Summary

The transformation of models while preserving the greatest amount of information possible is crucial in a model-driven approach.

Besides the technical task of coding the transformation rules themselves, issues such as model validation and information loss have also been addressed in the implementation process. Sometimes it is not possible to preserve all the information when performing such as a transformation. The key factor is ensuring that whatever information is lost only because the target metamodel does not provide a natural way to represent the concept or it is hard to provide an equivalent modeling structure that preserves the originally intended semantics.

Finally, we presented briefly the technical issue of model serialization which is addressed through the application of a text transformation.

Part III

Applications and Conclusions

The novel concepts presented in this work were developed in the context of the European Research Projects ATHENA and SHAPE. As part of these projects, we applied the new technologies developed to scenarios from industrial partners. The following two chapters present the applications:

1. Chapter 8 presents an proof-of-concept application of JadeOrgs as PSM with a non-agent-oriented PIM. The e-Procurement scenario presented comes from the ATHENA project.

2. Chapter 9 describes the main application scenario in this thesis: the model-driven specification of a steel production process. The scenario in this chapter deals with the production chain of Saarstahl AG, one of the industrial partners in the SHAPE project. In Saarstahl's case, the complete model-driven framework was applied and evaluated based on the feedback provided by the development team at Saarstahl.

Chapter 8

Proof of Concept: Modeling e-Procurement with PIM4SOA and JadeOrgs

Systems interoperability is a growing interest area due to the continuously growing need to integrate new, legacy and evolving systems, in particular in the context of networked businesses and eGovernment [BEF+07]. This need led to the inception of the ATHENA Integrated Project[1] and its main result: the ATHENA Interoperability Framework (AIF).

The main objective of the project was to conceive a multidisciplinary approach to the interoperability of enterprise applications and software. The approach combines the following three areas[BEF+07]: i) enterprise modeling to define interoperability requirements and support solution implementation, ii) architectures and platforms to provide the implementation frameworks, and iii) ontologies to identify interoperability semantics in the enterprise.

The AIF is structured in three parts:

Conceptual integration which focuses on concepts, metamodels, languages and model relationships. It provides a modeling foundation in order to analyze various aspects of interoperability.

Applicative integration focuses on methodologies, standards and domain models. It provides guidelines, principles and patterns that can be used to solve interoperability issues.

Technical integration which focuses on technical development and Information and communication technologies (ICT) environments. It provides ICT tools and platforms to develop and run enterprise application and software systems.

Our contribution to this project presented in this chapter is a component of the Technical Integration. As Figure 8.1 illustrates, the Technical Integration describes an architecture centered on a set of tools and infrastructure services to support the following: collaborative product design and development, cross-organizational business process, service composition and execution, and information interoperability.

In order to carry the cross-organizational process information into the service layer, the ATHENA SOA Framework [VBE+10] is applied. The concepts are represented at the PIM level using the PIM4SOA[2] metamodel, and at the PSM level metamodels for regular web services and MAS are used. The MAS component defines extensions that allow the SOA to take advantage of

[1] http://www.modelbased.net/aif/index.html
[2] http://pim4soa.sourceforge.net/

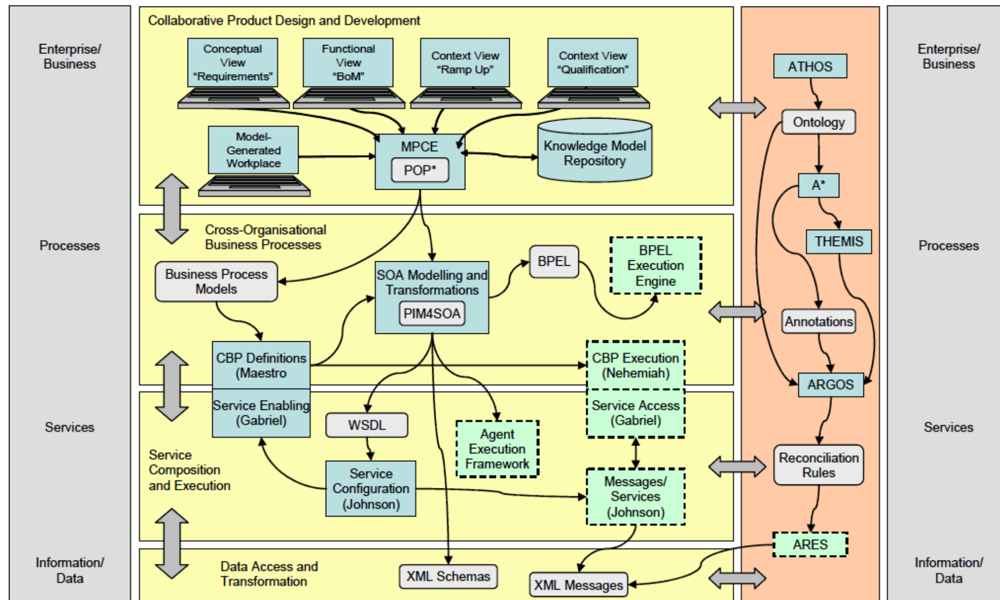


Figure 8.1: Overview of the AIF technical framework [BEF+07]

the brokering, mediation and negotiation capabilities that agents offer. Two MAS metamodels were offered as part of the framework: one based on the BDI agent framework JACK [AOS06], and an earlier version of the JadeOrgs called JadeMM[3]. We will concentrate on the application of JadeMM in this context.

The complete AIF was evaluated successfully in four selected scenarios covering Supply Chain Management (SCM), Collaborative Product Development (CPD), Electronic Procurement (e-Procurement) and Product Portfolio Management (PPM). We will present the application of the AIF to an electronic procurement scenario in this chapter: first, we give an overview of the PIM4SOA metamodel, then present the transformation to the JadeMM PSM, and finally describe the proof-of-concept scenario from ATHENA that demonstrates the application of the transformation.

[3]The metamodel name and some of the concepts names have been preserved throughout this chapter to keep it consistent with its publication [FHMM07]

8.1 Metamodel for Service-Oriented Architectures

The platform-independent model for service-oriented architectures (PIM4SOA) covers four important aspects. *Information* represents, in the context of virtual enterprises, one of the most important elements that need to be described. In fact, the other aspects manage or are based on the information elements. *Services* are an abstraction and an encapsulation of the functionality provided by an autonomous entity. In general, SOAs are formed by components provided by a system or a set of systems to achieve a shared goal. *Processes* describe a set of interactions among services in terms of messages exchange. Another suitable feature is the description and the modeling of the *Quality of Service* aspect related with the described services. In the following, we discuss the service, information and process aspects as we relate to these aspect in the model transformations.

8.1.1 Service Metamodel

This subsection describes the elements of the service-oriented metamodel that have the objective of describing service architectures. These architectures represent the functionalities provided by a system or a set of systems to achieve a shared goal. Functionalities could be represented as a service or as a set of services. In this work we emphasise the concept of collaborations to address the different levels of service description. In this subsection, we sketch out the main components of the service oriented metamodel. The service aspect of the PIM4SOA presents services modeled as collaborations that specify a pattern of interaction between the participating roles. A subset of the metamodel for this aspect is presented in Figure 8.2.

Collaboration represents a pattern of interaction between participating roles. A binary collaboration specifies a service. A Collaboration definition contains a set of roles (provider, requester) and a set of collaboration uses. Eventually it could be related with non-functional aspects. A Collaboration is related with a registry where it is specified the endpoints. Basically the attributes are:

- Subcollaborations: represents the usage of other collaborations

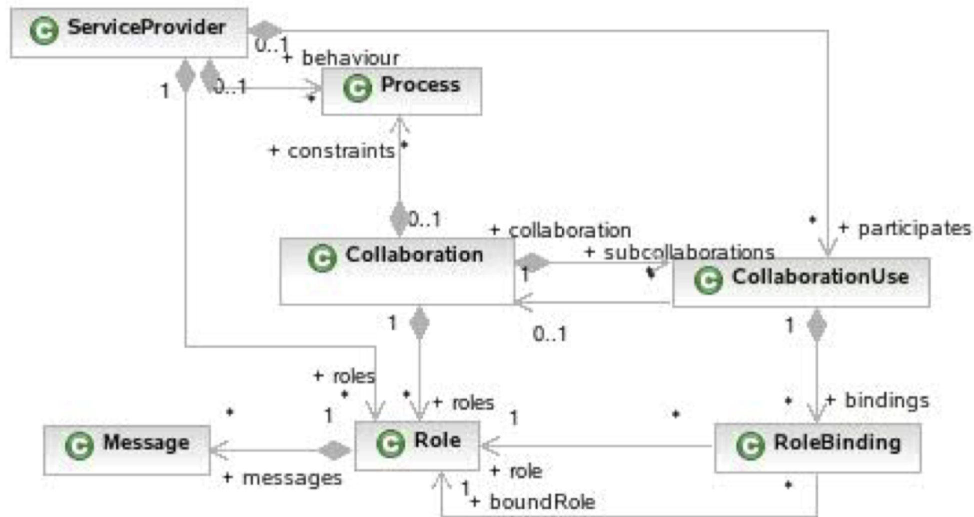


Figure 8.2: The service metamodel of the PIM4SOA.

- Constraints: constrains a collaboration by the specification of a process.
- Roles: involved within the collaboration
- Nfa: this element sets up a link to quality of service model definition
- Endpoint: is specified at design time
- RegistryItem: specifies the registry item associated with the collaboration

CollaborationUse represents the usage of a collaboration. In other words, a CollaborationUse is the model element to represent a usage of a service. The CollaborationUse contains a reference to the endpoint pointing out the address. Its attributes are:

- Provides: specifies the item provided
- Messages: specifies the messages related with this role

- RoleType: specifies the type of the Role. Basically a Role can be a requester or a provider. If it is not none of them we can specify it as other and in the property Other we specify the name.
- Other: used for the special case where the role is neither a requester nor a provider.

RoleBinding relates a role with a usage of a service. When we specify a collaboration use we need to identify which are the roles involved. This relationship is made between two Roles: one inside the collaborationUse and other inside a collaboration definition. Its attributes are:

- Role: represents a link to specific role within the collaboration definition of the current collaboration use
- BoundRole: represents a link to specific role within the current collaboration

Behaviour is an abstract class for the specification of message sequences within a service. This element represents a parent class connecting a service aspect with process aspect.

ServiceProvider specify an entity describing and specifying in its turn services, roles and constraints. ServiceProvider represents a service specification containing the specification of other services. Non functional aspects could also be added to specify quality aspects. Its attributes are:

- Behaviour: represents the process
- Participates: contains a set of the collaboration uses
- Roles: defines the roles involved at this level
- Nfa: establishes the link to the quality of service model
- QosCategory: defines the category in terms of quality of service
- Type: refers to the type of provider: Abstract or Executable

Message defines a unit of information sent from one role to other role in a collaboration. A message is owned by a specific role. Its attributes are:

- Contains: defines a set of items related with the message

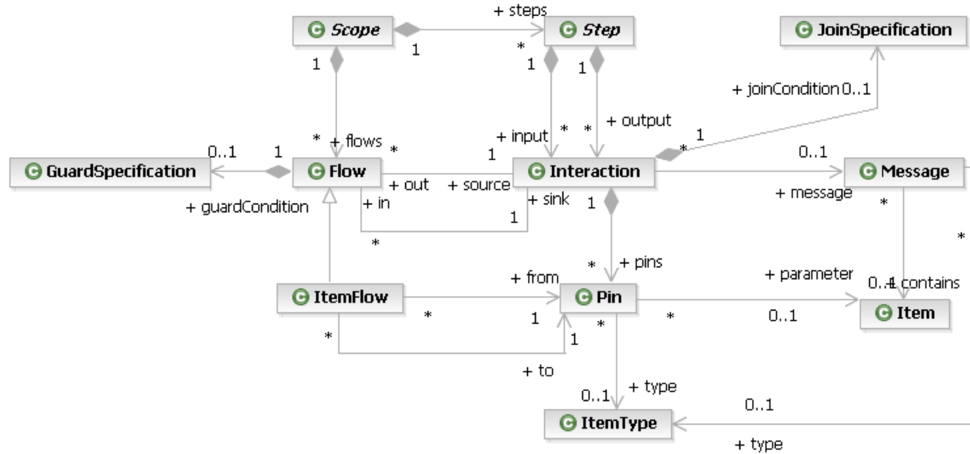


Figure 8.3: The process metamodel of the PIM4SOA.

- Type: defines the type of the items related with the message
- Mode: differentiates messages between regular (normal) or fault (exceptions)

8.1.2 Process Metamodel

The process elements of the PIM4SOA metamodel are presented in Figures 8.3 and 8.4. The process aspect is closely linked to the service aspect, the primary link being the abstract class Scope, which can be instantiated as a Process belonging to a ServiceProvider from that aspect.

The process contains a set of Steps (generally Tasks), representing actions carried out by the Process. A Process consists of StructuredTasks (sub-processes), Steps (atomic tasks and actions, at the PIM level), and Interactions/Flows linking the tasks together. These essentially fall into two categories, interactions with other service providers, or specialised actions requiring implementation beyond the scope of this model. For example, manual tasks to be processed by humans, or extensive computation requiring platform specific code.

The Process also contains a set of Flows between these actions, which may be specialised (ItemFlow) to indicate the transfer of specific data. This allows flexibility in that a business modeler may choose to start by showing

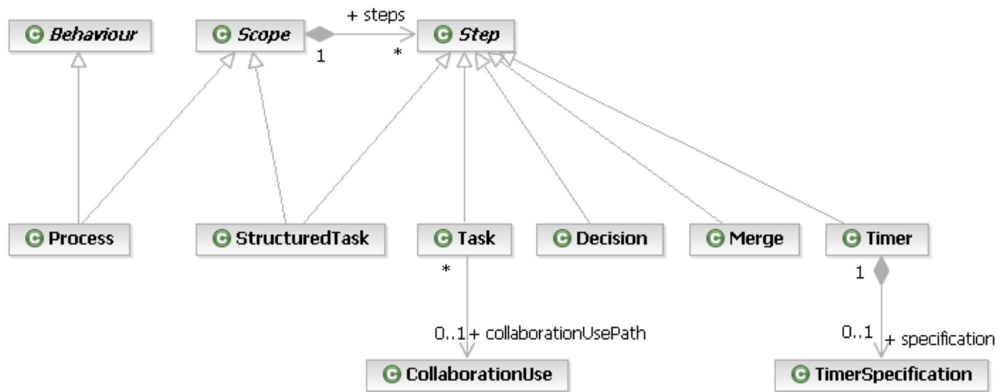


Figure 8.4: Behaviour, Scope and the Steps inheritance hierarchy.

only control flow, and later refine the model to include information. This links in to the Item/ItemType parts of the information aspect. Flows may diverge or reconverge using Guard and Join specifications.

Scope is an abstract container for individual behavioural steps. This is subclassed only by Process and StructuredTask (Process is the top level behavioural object, StructuredTask may be used to group related Steps in a subroutine like manner.)

Step is a single node in a process, such as making a decision or calling an external service. The specialization of Step is Task.

Process implements a behaviour for a service provider, as a set of tasks and decisions (Steps) linked by control flows (Flows), optionally including detail on the exchanged messages / items.

Task represents the low level building blocks of a process—these might be for example calls to another service (which can be transformed largely automatically to an implementation platform, with reference to the relevant collaborations) or might require manual intervention—either in the form of hand coded functions, or human interaction with the process.

- Collaboration Use Path: This is a path through the tree of CollaborationUses associated with this ServiceProvider. It must start

with a CollaborationUse, in which this ServiceProvider participates, and walk through any sub collaborations to the collaboration being implemented by this Task.

Interaction defines an interface for input or output flows on a Step. Can be viewed as a set of Pins, though it is not compulsory to refine the model to this level (depending on aims of the model). If the step is viewed as a service, this is similar to the declaration of a method/function in the interface (specifying a set of parameters or a return value).

8.1.3 Information Metamodel

This section describes the concepts needed to model information at a platform-independent level.

Item defines the set of elements that a role manages.

ItemType represents simple types: string, integer and boolean.

Association represents the association between two entities. It is used to describe complex types. Container, contained and cardinality are the attributes necessary to related elements.

Document represents an object with a specific structure and composed by entities. Document is a stereotyped package containing the structure of the document.

TypeLibrary defines a packaging structure containing some types of the application TypeLibrary is a stereotyped package containing data types.

Entity represents a structure element of information. Entity is a stereotyped class.

8.2 Model to Model Transformations

In this section, we bring together the concepts in one metamodel and relate them to another one in a mapping. Although the metamodels are on different abstraction levels , we show that a mapping is feasible as the platform-specific

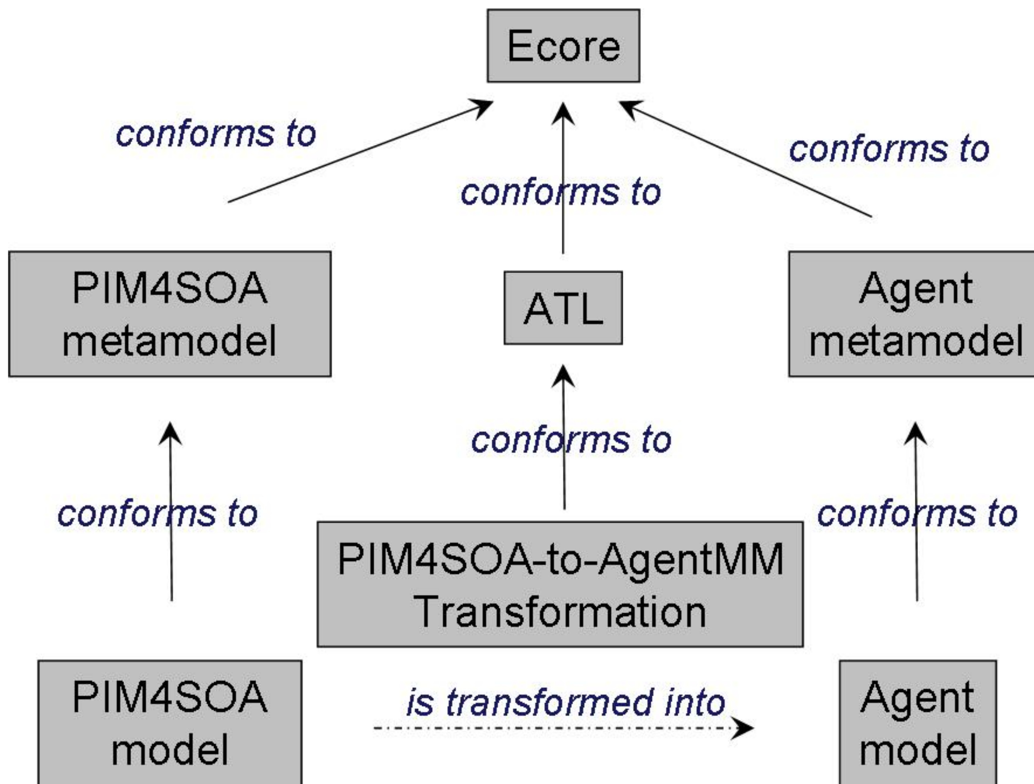


Figure 8.5: The overall picture: From service-oriented architectures to agent systems using MDA standards.

metamodel is more expressive than the PIM4SOA. Therefore the transformation shows that JADE can be used in a MDD scenario to deploy service models.

The implementation of model to model transformations is done using the Atlas Transformation Language (ATL) ([ATL06, JK05]). Figure 8.5 presents how they all come together as a framework.

The definition of a model transformation requires deep knowledge of the corresponding source and target metamodels with respect to their syntax, which is clearly defined by the metamodels themselves, and semantics, which are often not explicitly reflected in the metamodels. Our model to model transformations—discussed in the two following sections—are mainly based on the following observations. The interaction in the PIM4SOA is always done by provider and requester roles only. This allows to use the PIM4SOA

model in a manner where the information on Collaborations and CollaborationUses is mapped to an interaction that is performed by a service requester and a set of roles this specific service requester makes use of. However the concept of a Collaboration is not directly mapped. Due to the fact that non-composed collaborations in the PIM4SOA are binary, it is always clear who requests and who provides the corresponding service. Interactions other than pure service requests-provisions do not exist.

8.3 From PIM4SOA to JadeMM

This section presents how the PIM4SOA metamodel and JadeMM metamodel relate to each other and an overview of how the transformation rules look like.

Transformation 1:

Head: $PIM4SOA.Service : ServiceProvider \rightarrow JadeMM : -Organization$

Body: For each PIM4SOA.Service:ServiceProvider, a corresponding JadeMM:Organization is created. For each service provider's process we generate an organisational Behaviour. The details of the transformation are summarized by Table 8.1.

Transformation 2:

Head: $PIM4SOA.Service : Message \rightarrow JadeMM : ACLMessage$

Body: Each PIM4SOA.Service:Message is transformed into a JadeMM:ACLMessage in a straightforward fashion. The transformation details are presented in Table 8.2.

PIM4SOA.Service : ServiceProvider → JadeMM : Organization		
Target	*Source*	*TR*
Organization.requires	determined from PIM4SOA.-Service.ServiceProvider.-participates	4
Organization.behaviours	behaviour(s) obtained from PIM4SOA.Service.-ServiceProvider.behaviour	3
Organization.membership	determined from PIM4SOA.-Service.ServiceProvider.-participates.collaboration.roles.	4

Table 8.1: Transformation 1 in detail.

Transformation 3:

Head: *PIM4SOA.Processes : Process→JadeMM : −SequentialBehaviour*

Body: The Process from the ServiceProvider is mapped to a sequential behavior in the Organization. The order of the sub-behaviours is determined by following the PIM4SOA.Processes.-Flows and PIM4SOA.Processes.ItemFlows that link the Tasks. The details are presented in Table 8.3.

Transformation 4:

Head: *PIM4SOA.Service : Collaboration, PIM4SOA.Service : Role → JadeMM : Role, JadeMM : Agent*

Body: All PIM4SOA.Service:Roles in the collaboration are mapped to JadeMM:Roles, those that are not bound to the ServiceProvider are also mapped to a corresponding implementing JadeMM:Agent. The subcollaborations of the top level collaboration are navigated to determine what messages from the lower

$PIM4SOA.Service : Message \rightarrow JadeMM : ACLMessage$		
Target	*Source*	*TR*
ACLMessage.content	filled at runtime, but its type is determined by PIM4SOA-.Service.Message.Type	5
ACLMessage.sender	filled at runtime, but its type is obtained from the containing Role, if PIM4SOA.Service.Role.-RoleType has the value of Requester	—
ACLMessage.receiver	filled at runtime, but its type is obtained from the containing Role, if PIM4SOA.Service.-Role.RoleType has the value of Provider.	—

Table 8.2: Transformation 2 in detail.

level roles correspond to each top level role. The implementing Agents for each Role that provides requested information to the ServiceProvider is enriched with a behaviour for invoking that party's web service. Please note that the Collaboration itself is not mapped to any concept in JadeMM. The transformation details are presented in Table 8.4.

Transformation 5:

Head: $PIM4SOA.Information : Entity \rightarrow JadeMM : ConceptSchema$

Body: The information that is sent in Messages is described by so-called Entities in the information metamodel. We map these Entities to ConceptSchema of the Ontology that the JADE Agent has available. The details are presented in Table 8.5.

$PIM4SOA.Processes : Process \rightarrow JadeMM : SequentialBehaviour$		
Target	*Source*	*TR*
SequentialBehaviour.subbehaviours	behaviours obtained from transforming PIM4SOA.Processes.-Process.steps	7

Table 8.3: Transformation 3 in detail.

$PIM4SOA.Service : Collaboration, PIM4SOA.Service : Role \rightarrow JadeMM : Role, JadeMM : Agent$		
Target	*Source*	*TR*
JadeMM:Role.sends	obtained from PIM4SOA.-Service:Role.message if PIM4SOA.Service.Role.-RoleType is Provider	2
JadeMM:Role.receives	obtained from PIM4SOA.-Service:Role.message if PIM4SOA.Service.Role.-RoleType is Requester	2
JadeMM:Agent.behaviours	a generic web service invocation behavior is added to the behaviour set.	—

Table 8.4: Transformation 4 in detail.

Transformation 6:

Head: $PIM4SOA.Information : Attribute \rightarrow JadeMM : Slot$

Body: The attributes are mapped to the concept's attribute in a straight forward manner. The details are presented in Table 8.6.

$PIM4SOA.Information : Entity \rightarrow JadeMM : ConceptSchema$		
Target	*Source*	*TR*
ConceptSchema.slots	obtained from PIM4SOA.-Information.Entity.attribute	6

Table 8.5: Transformation 5 in detail.

$PIM4SOA.Information : Attribute \rightarrow JadeMM : Slot$		
Target	*Source*	*TR*
Slot.type	generated by applying Transformation 5 on PIM4SOA.-Information.Attribute.type if it is a complex type, or a primitive type if the attribute is of primitive type.	5

Table 8.6: Transformation 6 in detail.

Transformation 7:

Head: $PIM4SOA.Processes : Task \rightarrow JadeMM : MessageReceiverBehaviour, JadeMM : MessageSenderBehaviour$

Body: When the PIM4SOA.Processes:Task's input interaction contains a message it is mapped to a JadeMM:MessageReceiverBehaviour. If the PIM4SOA.Processes:Task's output interaction contains a message it is mapped to a JadeMM:MessageSenderBehaviour. It is never the case that both interactions contain a message, since the interaction is unidirectional. The details are presented in Table 8.7.

PIM4SOA.Processes : *Task* → *JadeMM* : *MessageReceiverBehaviour*, *JadeMM* : *MessageSenderBehaviour*		
Target	*Source*	*TR*
JadeMM:-MessageReceiverBehaviour.-receives	obtained from PIM4SOA.-Processes.Task.input.message	2
JadeMM:-MessageSenderBehaviour.sends	obtained from PIM4SOA.-Processes.Task.output.message	2

Table 8.7: Transformation 7 in detail.

8.4 Use Case Scenario

The following section exemplarily explains our approach by discussing a use case scenario that (i) is modeled in accordance to the PIM4SOA and (ii) is transformed to the agent the JADE metamodel. The scenario could be summarized as follows.

A service integrator software provides the service of listing the options offered by different car manufacturers with respect to a dealer's desires and requirements (i.e. price, equipment, etc.). The service integrator software evaluates the dealer's request and selects those car manufacturers that offer products that fulfill the dealer's requirements, followed by sending an initialize product request to the responsible services on the car manufacturers' side. The car manufacturer's internal legacy system evaluates the service request and replies by sending a list of options in a initialize product response. After receiving the manufacturers' list of options, the dealer software collects and evaluates the responses and illustrates the set of options on its web site. The consumer then may evaluate the options and may proceed by for instance further restricting its requirements or selecting particular models to get further information on these.

8.4.1 SOA Model in accordance to the PIM4SOA

Based on the use case description above, we model a SOA that consists of three actors—the dealer software, the service integrator and the manufacturer. To simplify the given use case, we assume that only one car manufacturer is involved. The only reason for this assumption is to keep the

SOA model small so that is fits on one page. We start by modeling the ServiceProvider which is represented in the description above by the service integrator software. The ServiceProvider participates in the CollaborationsUses *Dealer* and *Manufacturer* (see Figure 8.6). The ServiceProvider's role *ServiceIntegrator* is bound to roles defined by the Collaborations *DealerCollaboration* and *ManufacturerCollaboration* via the so-called RoleBindings. For instance the RoleBinding *SI_to_SI3* binds the ServiceProvider's role *ServiveIntegrator* to *ServiceIntegrator3* defined in the Collaboration *DealerCollaboration*. A Collaboration refers to a set of subcollaborations which are again CollaborationsUses, i.e. the *DealerCollaboration* refers to a CollaborationUse *IntegratorInitRequest*. This CollaborationUse refers again to the Collaboration *IntegratorInitRequestCollaboration*; the roles of *IntegratorInitRequestCollaboration* are bound to the *DealerCollaboration*'s roles via the RoleBindings contained in the *IntegratorInitRequest* CollaborationUse. The Collaboration *IntegratorInitRequestCollaboration* refers again to two Roles *Dealer1* and *Integrator1*, which receives a Message *integratorInitRequest*. The Collaboration that defines the interaction between Manufacturer and ServiceIntegratorProvider (*ManufacturerCollaboration*) is structured in a similar fashion as described above for the interaction between Dealer and ServiceIntegratorProvider.

In Figure 8.7, we find the process model that corresponds to the scenario previously described. The *ServiceIntegratorProvider* has a behavior that consists of three tasks: *ReceiveInit*, *SendInitM1*, and *ReceiveInitM1*. Each of these tasks is linked to a CollaborationUse presented in the service model. *ReceiveInit* possesses an input interaction, the reception of the triggering message *integratorInitRequest* from the Dealer, and an output interaction that links the control flow and data flow to the next task through *Flow1* and *Flow3* correspondingly. The attribute *data* from *integratorInitRequest* is taken from *Pin1* to *Pin2* by ItemFlow *Flow3*. The second task, *SendInitM1*, consists in forwarding the request from the Dealer to the Manufacturer. The input interaction, *Interaction3*, receives the control and data flows from the previous task, and *Interaction4* links the control flow and data flow to the next task. *ReceiveInitM1* consists in the reception of the reply from the manufacturer, therefore it only has an input interaction. It is apparent that further tasks and interactions are missing, such as informing the dealer of the acceptance or rejection of the original request, or status of the order, but these are left out to keep the model compact, since they would

not contribute to further understanding of the model itself.

8.4.2 Applying the transformation from PIM4SOA to JadeMM

After the transformation is applied to the example, we obtain the model presented in Figure 8.8. We can see that after applying *Transformation 1*, the ServiceIntegratorProvider is now an Organization with two agent members, two required roles, one implemented role, a behaviour and an ontology. The ServiceIntegratorProviderProcess is a SequentialBehaviour derived from *Transformation 3*, the order of the subbehaviours was determined following the Flows and ItemFlows from the PIM4SOA model. The Tasks from the PIM4SOA model were mapped to MessageReceiverBehaviours and MessageSenderBehaviours according to the Tasks' Interactions following *Transformation 7*. The application of *Transformation 4* does away with the collaboration/subcollaboration tree from the PIM4SOA model, and creates the three roles and two member agents of the Organisation that implement the roles not assigned to the service provider. When the lower level roles that refer to messages are reached in *Transformation 4*, *Transformation 2* is fired and the messages are created. The type of the content of the messages is represented by an Entity in the PIM4SOA model, so *Transformation 5* is fired to create the contents of the Ontology that the Organisation and the Agents will use.

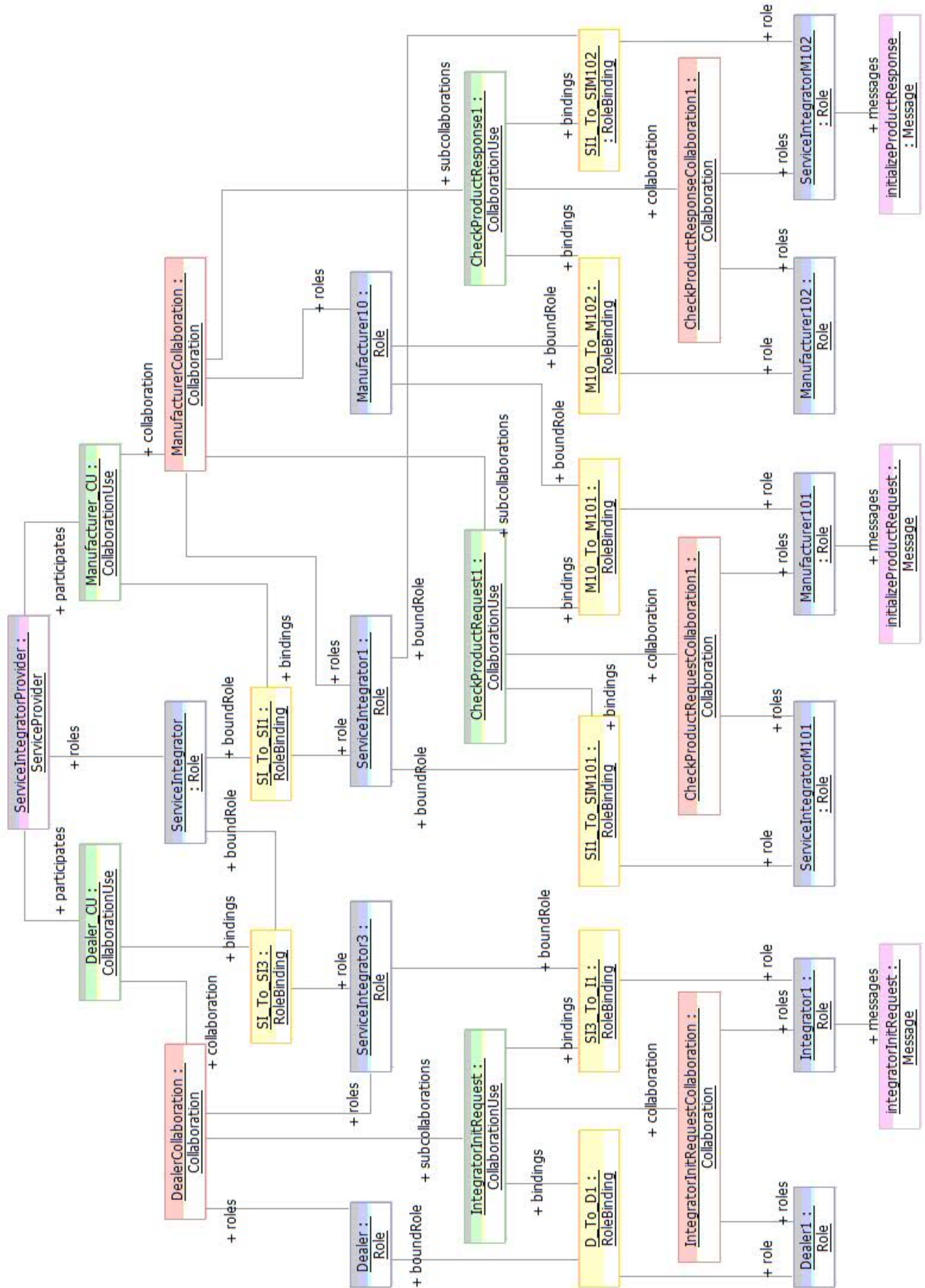


Figure 8.6: The example illustrates the service model.

Figure 8.7: The example illustrates the process model.

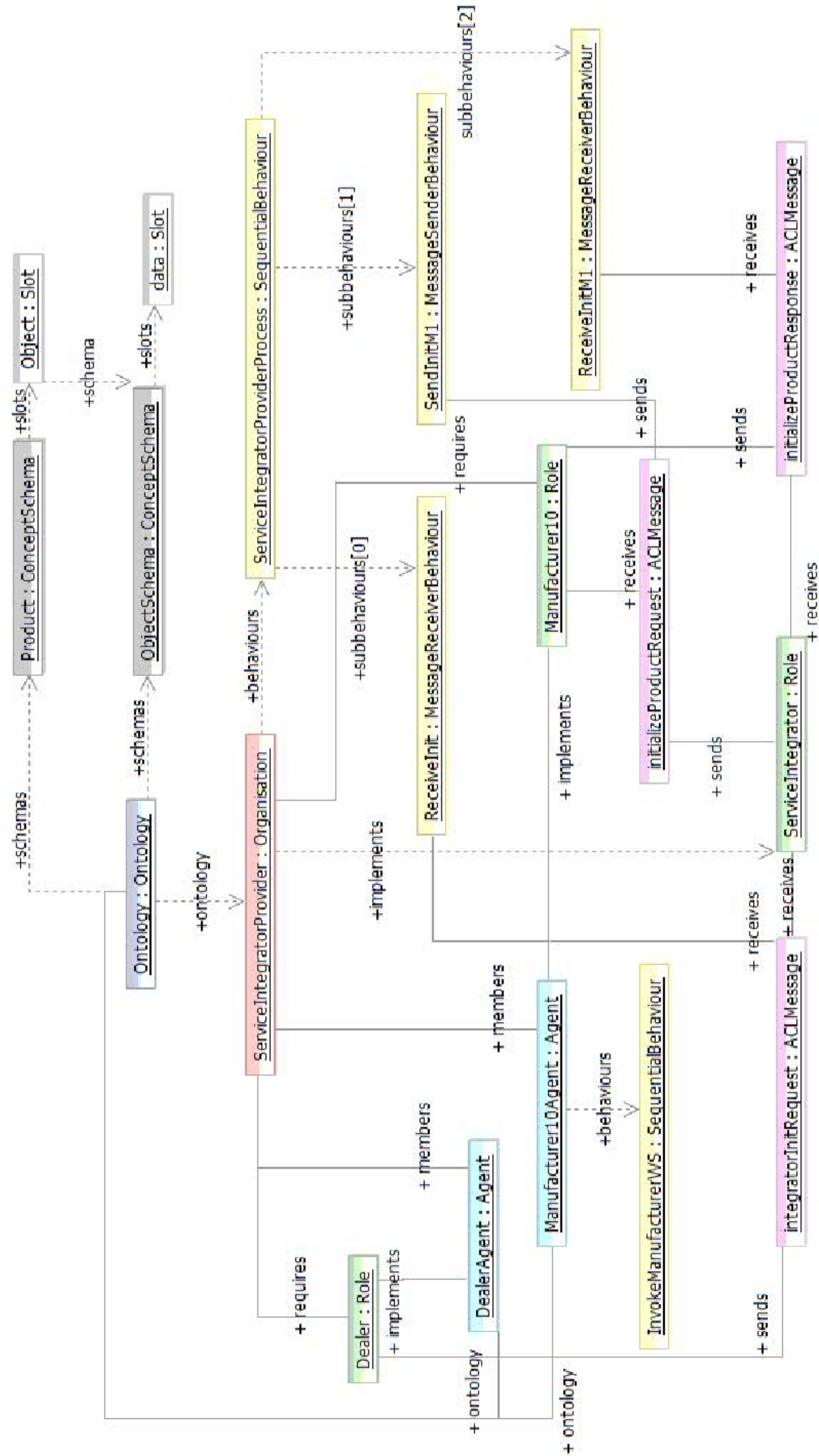


Figure 8.8: The example after the transformation to JADE was applied.

8.5 Summary

This chapter presented a proof-of-concept scenario for the AIF in the context of e-Procurement. In this scenario, an agent-oriented application was specified in a model-driven manner in order to build interoperable agent systems at the PSM level. By the application of MDD, we showed how service models could be deployed in the agent platform JADE. Therefore, we defined model transformations that transfer concepts of a metamodel for SOAs (PIM4SOA) to the metamodel for JADE (JadeMM). Consequently, service models that conform to the PIM4SOA can be executed as JADE MAS in a generic manner.

The development of JadeMM and its relation to the SOA modeling languages led to the later development of the PIM4Agents (see Chapter 5) and JadeOrgs. In addition, PIM4SOA had a strong influence on the development of the OMG-supported SoaML language described in Chapter 9.

Chapter 9

Case Study: Applying Multiagent Systems to a Steel Production Process

Competitiveness is a key factor for companies in today's globalized economy. Potential customers expect goods of highest quality at competitive prices, while companies, especially those located in high wage countries, need to find ways to be very efficient and flexible in the production of these goods. Therefore, the production processes need to be correspondingly flexible and cost effective, while still being able to meet production deadlines.

At the same time, there are two major trends developing in IT infrastructures today [BBC+10]:

First, SOA has emerged as a direct result of business and technology developments of the last decade. The outsourcing of non-core business operations, the rise of process reengineering and the need of system integration have led to business processes based on the integration of services provided by different parties.

Second, modeling has gained a prominent position among software engineering approaches. The use of various abstraction levels in order to model business processes with various levels of detail, along with the application of model transformations in Model Driven Engineering (MDE) approaches have increased its popularity both in academic and industrial IT circles.

The previously described business conditions demand that the applications, which support these production processes, are also flexible and adaptable to changes with the same speed that the market demands. The convergence of SOA and modeling provide the basis to build the integrated development environment that such applications require.

The SHAPE (Semantically-enabled Heterogeneous Service Architecture and Platforms Engineering) European Research Project [LTB+08, LTB+09, CEG+09, EHJL10] was conceived to fulfill this demand. The project provides an integrated development environment that brings together MDE and SOA paradigms. This environment is complemented with innovative service techniques, such as support for flexible business modeling, customization and personalization of services, integration of agent technology and adaptive systems, and support for the use of semantic technologies. The MDE techniques revolve around SoaML, a metamodel for the description of service-oriented landscapes and an OMG standard. It is complemented with other metamodels that describe particular technology platforms in which the systems can be deployed. The project produced the necessary infrastructure for the application of these technologies in real-world applications. Its integrated tool suite delivers the modeling tools, transformations and a methodology framework

which explains how to apply these technologies to real-world scenarios.

The SHAPE approach was tested, demonstrated and evaluated by two industrial use case partners: Saarstahl AG and StatoilHydro. In Saarstahl AG's case, SHAPE technologies were used to integrate legacy systems and the agent-based production line planning system. At StatoilHydro, SHAPE technologies also helped in the integration of legacy systems and are expected to lead to greater efficiency in system development and maintenance in the long run.

In this chapter, we will cover the Saarstahl AG scenario. This industrial use case was modeled as a service-oriented architecture implemented by agents. In this scenario, the SoaML model is transformed into a PIM4Agents model, then into a corresponding JadeOrgs model and finally into Java code.

9.1 Scenario Description

Saarstahl AG[1] is a worldwide known manufacturer of steel products based in the state of Saarland, Germany. It is located in the cities of Völklingen, Burbach and Neunkirchen and specializes in the production of wire rod, steel bars, semi-finished products of various grades, as well as open die forgings. These are important preliminary products for various industries such as automotive, construction, aerospace and general mechanical engineering.

The production chain at Saarstahl is constituted of a series of metallurgical manufacturing processes that depend on one another. Arguably, the critical link in the production chain is the steelwork in Völklingen. This is where the steel is produced following the customer's specifications and requirements. This steel is produced in units called *heats*, which are then grouped into *sequences*. The sequences are cast into billets at the steelwork's continuous casting plants. The heats on each sequence are related because of their similarity with respect to their quality grade and format. Once the billets have been cast, they are forwarded to the rolling mills. At the mills, the billets are warmed up again in order to manipulate the steel and produce bars and wire rods of different sizes and formats. The formats are produced in accordance to fixed, cyclic rolling campaigns. The cycles vary between one and four weeks and depend on the capacity at the rolling mills, the billet supply from the steelwork and the customer orders.

[1] http://www.saarstahl.com/

Internal and external processes in this supply chain are modeled as services to provide Saarstahl with a better information exchange and increased transparency while keeping the processes loosely coupled. Originally there were four use cases defined for this scenario [LTB+08], but in order to explore a higher level of detail and complexity while still meeting the project's time constraints, the focus was shifted to one use case. Therefore, this scenario revolves around the relation between the steelwork and the rolling mills, and their processes within the supply chain.

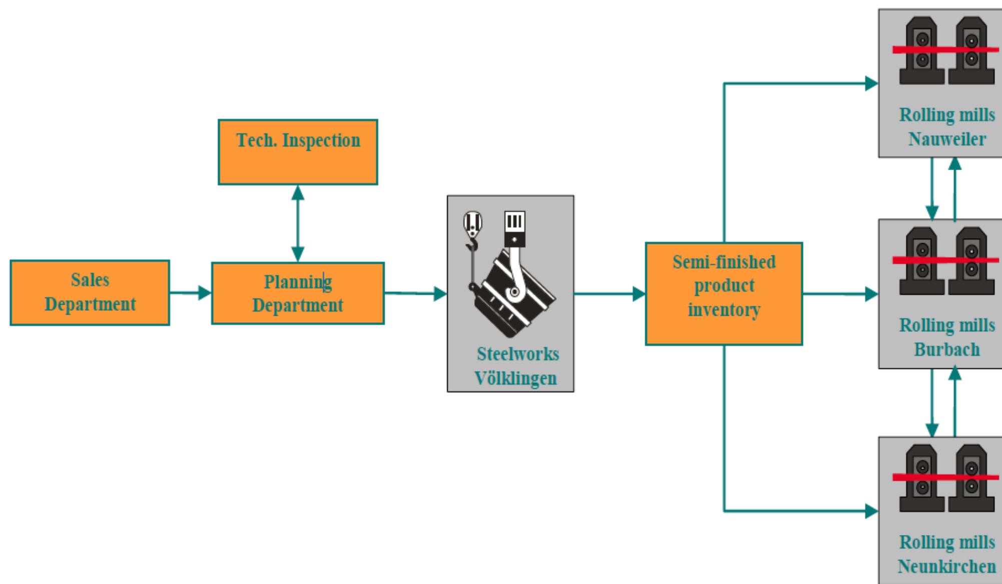


Figure 9.1: Saarstahl departments involved in the Steelwork-Rolling Mills Scenario [CEG+09]

The overview of the interaction between the parties in this scenario is presented in Figure 9.1[2]. In [CEG+09] this use case for the scenario "Correlation steelworks and rolling mills" is defined in detail as follows:

1. The customer uses a service "ordering" to communicate relevant data such as demanded quality, shipping date, quantity, etc. These existing functionalities supported by legacy systems are wrapped behind this new ordering service. The set of data communicated is called "order_specification".

[2] The cited figures in this chapter are reproduced with authorization from the authors

2. Order_specification is passed to sales department. This department creates an order instance *o* and allocates a rolling capacity in a specified slot for *o*.

3. From now on the actions are triggered by the order agent *o*. The main idea concerning planning and scheduling along the supply chain of Saarstahl is to model each single order position as a software agent. Every single order agent calculates and observes its own schedule from order entry until invoicing. Instead of handling a vast number of restrictions subject to the manufacturing step in general, only a few related to a single order position are handled by the entity they are related to—the order position. A decentralised management of manufacturing control is received instead of a centralised, data driven approach.
   First, a service "productionPossible" is used to determine whether Saarstahl is in general in the position to meet the specific requirements. Normally, this is straightforward and answered automatically in seconds, but in special cases concerning new requirements a feasibility study by the research and quality department is necessary. The answer of this service is passed to the sales department. A service "informCustomer" is used for any customer relevant information exchange during production, so the order confirmation.

4. After order confirmation a service called "rollingDispatched" is used by *o* to check status of the allocated rolling. The status gives information on the semi finished product demand of *o*.

5. Rolling campaigns are recalled about five weeks in front of rolling. The following workflow is initiated by each *o* in the campaign:

   (a) *o* registers to "VMM" (a legacy system managing semi finished products) by use of a service called "requestManagement".

   (b) *o* demands semi finished material by use of service "retrieveSF-Products"

   (c) *o* calculates a proposal of a production plan including melting or allocation of material available

   (d) *o* submits proposal to planning department by use of service "requestCommit" and waits for this commitment, which is done manually.

(e) the planning department submits final plan by use of service "inform"

6. *o* realizes production plan:

   (a) Case "melting required" (In this case no suitable material was found inside the semi finished product inventory and hence material has to be melted)

      i. *o* demands, by use of "LMSTCalculation", a LMST (LMST is latest possible melting date in German abbreviation)
      ii. *o* registers to SPL by use of "registerMelting" (SPL is a database containing all order positions which still have to be melted)
      iii. steelwork offers a service "inform" which can be used by *o* to get status information
      iv. after melting *o* continues with 5b.

   (b) Case "material available" (In this case no melting is necessary since enough is available inside the inventories)

      i. *o* allocates corresponding material by use of "bindMaterial" and sets itself disposed for scheduling by use of "scheduleRolling".

Each of the steps presented in this use case description are described in additional detail in [CEG+09] using ARIS Event-driven Process Chain (EPC) notation from the DFKI-IWI[3] group.

9.2 Methodology

Because of the unpredictability of future orders as well as factory capacities, it is critical for Saartahl AG to improve overall efficiency and maximize flexibility. Therefore, Saarstahl's main goal in the context of SHAPE was to explore the potential of service-oriented architectures to achieve this efficiency and flexibility.

In correspondence to the changes in the business processes, the methodology to produce such a system should also provide flexibility to the introduction of changes, such the addition of factory aggregates. The model-driven

[3] http://iwi.dfki.de/

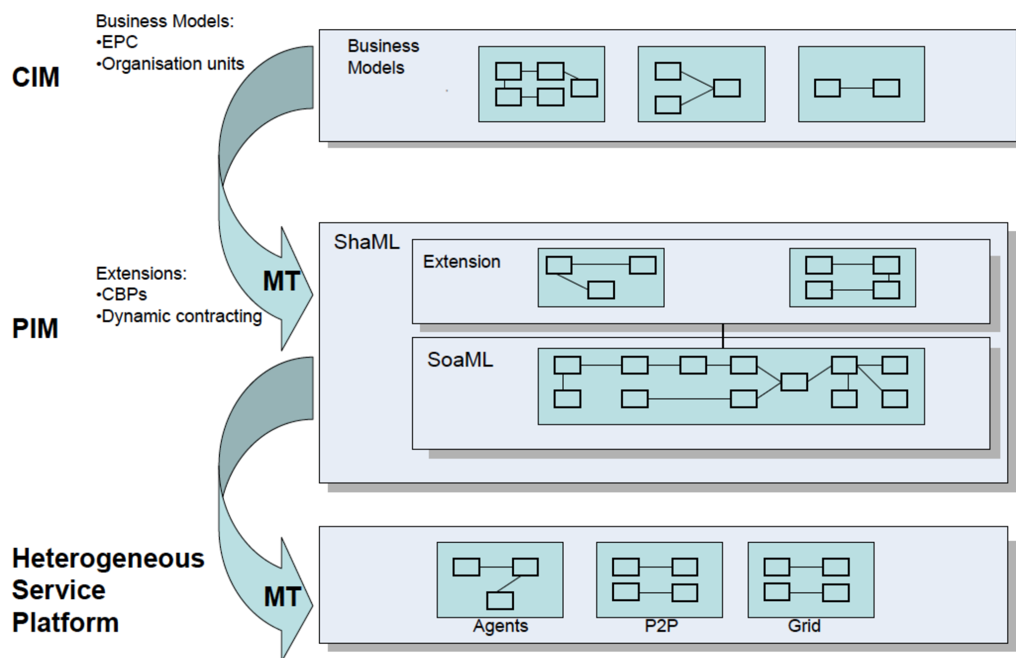


Figure 9.2: Model transformations from business models to heterogeneous executable platforms [CEG+09]

methodology defined in the context of SHAPE provides such flexibility by modeling production process as business models in the abstract CIM layer which changes seldomly. This layer is modeled using a combination or ARIS EPC notation and Business Process Model and Notation[4] (BPMN).

The PIM modeling for the scenario uses and extended version of SoaML, a service-oriented architecture standardization effort by OMG. The extended SoaML, ShaML, is the transformed to a variety of heterogeneous executable platforms. One of this platforms is a MAS implemented in JADE after being transformed into the corresponding PIM4Agents and JadeOrgs models. Figure 9.2 presents how these model transformations come together. In the remainder of this chapter, we concentrate on the PIM layer of the scenario and its transformation to PIM4Agents. Additional information about the CIM layer modeling can be found in [CEG+09].

9.3 Modeling the PIM Layer with SoaML

As in most large companies, the internal organization structure of Saarstahl AG is complex, but, for our presentation, we will consider a simpler structure concentrating only on its organizational units relevant to the scenario[5] The organizational units in Saarstahl are grouped into functional departments and factories. The functional departments include the sales, quality control, planning (PPL) and shipping departments, while the named factories include the Steelwork in Völklingen and the Rolling Mills in Burbach (BU), Nauweiler (NW) and Neunkirchen (NK).

At the SoaML PIM level, the architecture of Saarstahl for the scenario was modeled as depicted in Figure 9.3. In SoaML, every entity that provides or consumes a service is known as a *Participant*. Therefore all the departments, factories and the orders—agents—are modeled as Participants. Services are modeled through interfaces typed as *Service Capability* and service instances through classes typed as *Service Interfaces*. The complete architecture is also a *Participant Architecture*, since it exposes an external *Service Point* for customers to the *purchase* service from the *Manufacturer Services* interface. The *SaarstahlArchitecture* is instantiated by the *SaarstahlImpl* and every architecture component is also instantiated respectively.

[4]http://www.bpmn.org/

[5]The whole organizational structure of the company was presented to us in internal documents, but the complete detailed information is confidential.

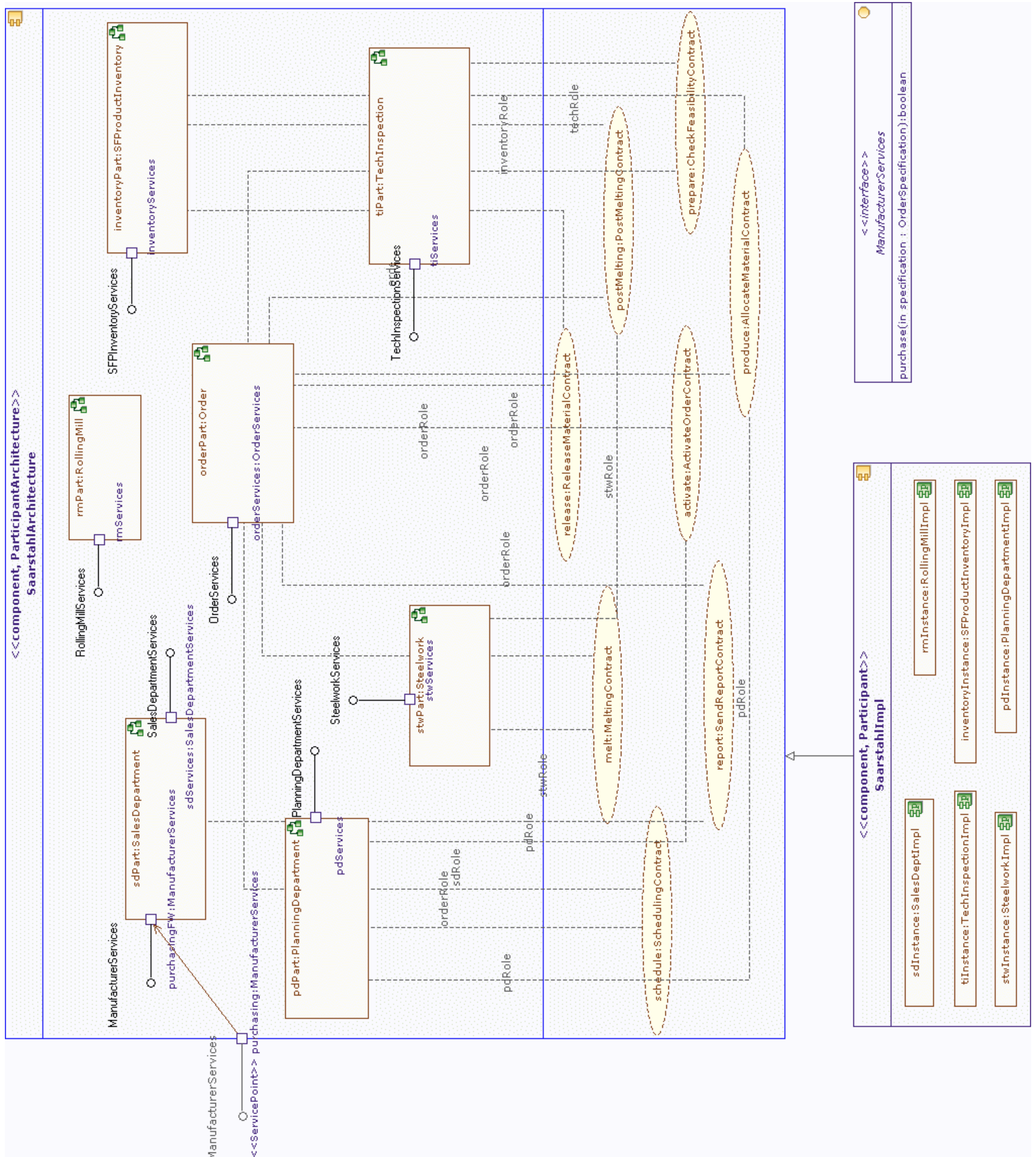


Figure 9.3: Saarstahl Architecture in SoaML [CEG+09]

The architecture encapsulates the organizational structure. The components of the architecture include:

orderPart the agent that tracks the status of each Order,

stwPart the steel melting factory,

pdPart the planning department,

sdPart the sales department,

rmPart the rolling mill(s),

inventoryPart the product inventory, and

tiPart the technical inspection (quality control).

Each of the architecture parts are also modeled in detail with their own service interfaces. For illustration purposes, let us examine in detail the *Order agent*. It manages the working plan for the order of which it is in charge. As illustrated in Figure 9.4, it maintains the order information and it handles the production events through *inform* service.

In Figure 9.3, we can also see the instantiation of the architecture in *SaarstahlImpl*. For this scenario, each part of the architecture is instantiated once.

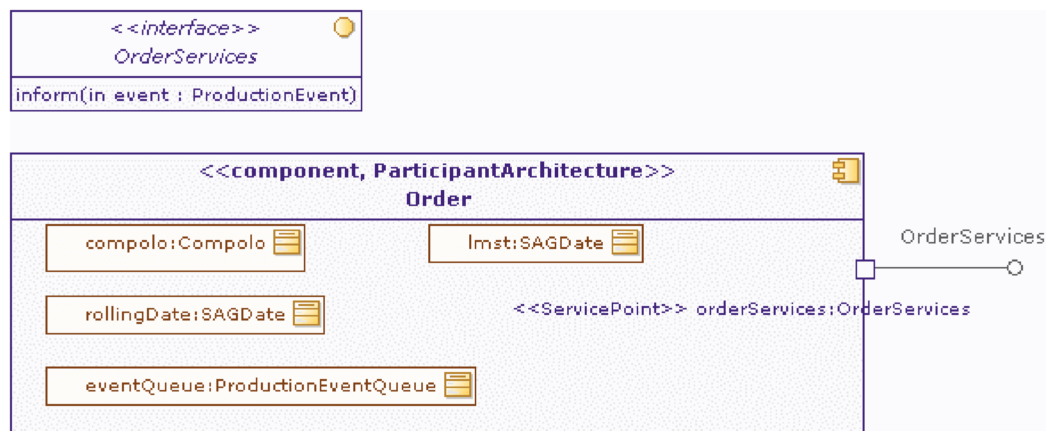


Figure 9.4: Order agent specification [CEG+09]

The *Order agent* also has a group of behaviors that describe, for instance, how the agent handles the incoming events or how it requests a melting position from the steel work. The event handling behavior is presented in detail in Figure 9.5. It consists of a loop that processes the events as they enter the handling queue, the events are produced by the various factories and departments as the order parts of the order change states in production. In order to reduce the stock of semi finished product and avoid producing unnecessary product, the stock is checked for availability of the qualities that the order requests. When no stock of a required quality is readily available, the *Order agent* requests melting the desired quality to the Steelwork as described in Figure 9.6.

The behaviors in Figures 9.5 and 9.6 model how a *Participant*, the *Order agent*, performs its tasks internally. In order to model the interaction among different *Actors* (*Participants*, System Users, etc.), *Service Contracts* are used. These contracts are specified via activity diagrams and help guarantee a correct process flow.

Let us take the interaction between a *customer Actor* and a *manufacturer Actor* in the context of a purchase in order to illustrate the *Service Contracts*. Figure 9.7 depicts the corresponding activity diagram. Each "swimlane" in the diagram contains the activities that each actor is committed to execute. For each activity, the actor should provide the required interfaces and services involved.

It is important to note that just as an agent organization can interact with other agents as a single agent, the *Saarstahl Architecture* is also a *Participant* and offers services to other *Actors*. Under the *Purchasing Process Service Contract*, the *Saarstahl Architecture* plays the role of the *manufacturer Actor*.

9.4 Corresponding PIM4Agents Models

As mentioned in Section 9.2, the presented SoaML model is then transformed into a PIM4Agents model. The technical details of this transformation can be found in [Rab09]. This section presents a sample of diagrams from the model that the transformation produces, concentrating on the organizational structures derived from the SoaML *Architectures* and *Participants*.

The agent diagram produced is depicted in Figure 9.8 and presents the basic agent types in the system. This corresponds to the general customer-manufacturer scenario only (see Figure 9.7). There are two agent types de-

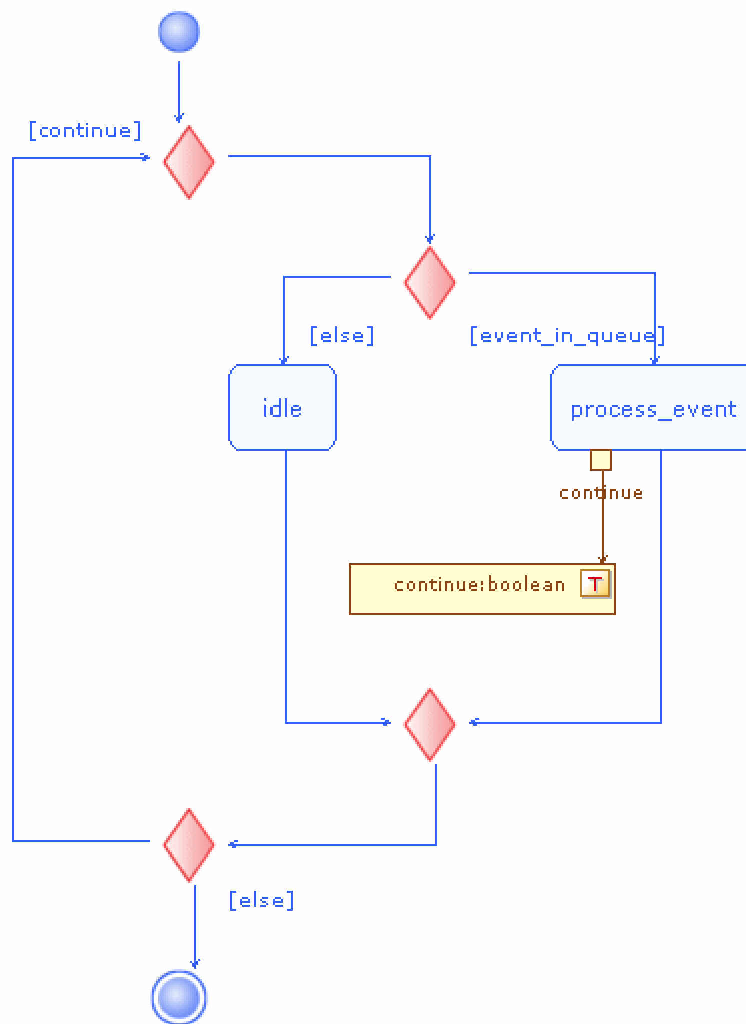


Figure 9.5: Event handling performed by the Order agent [CEG+09]

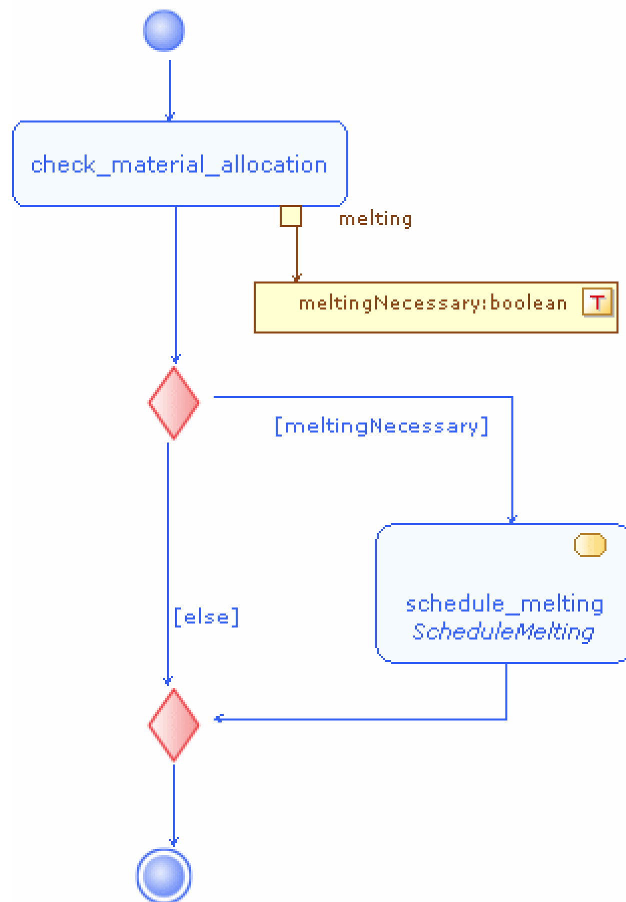


Figure 9.6: Melting request performed by the Order agent [CEG+09]

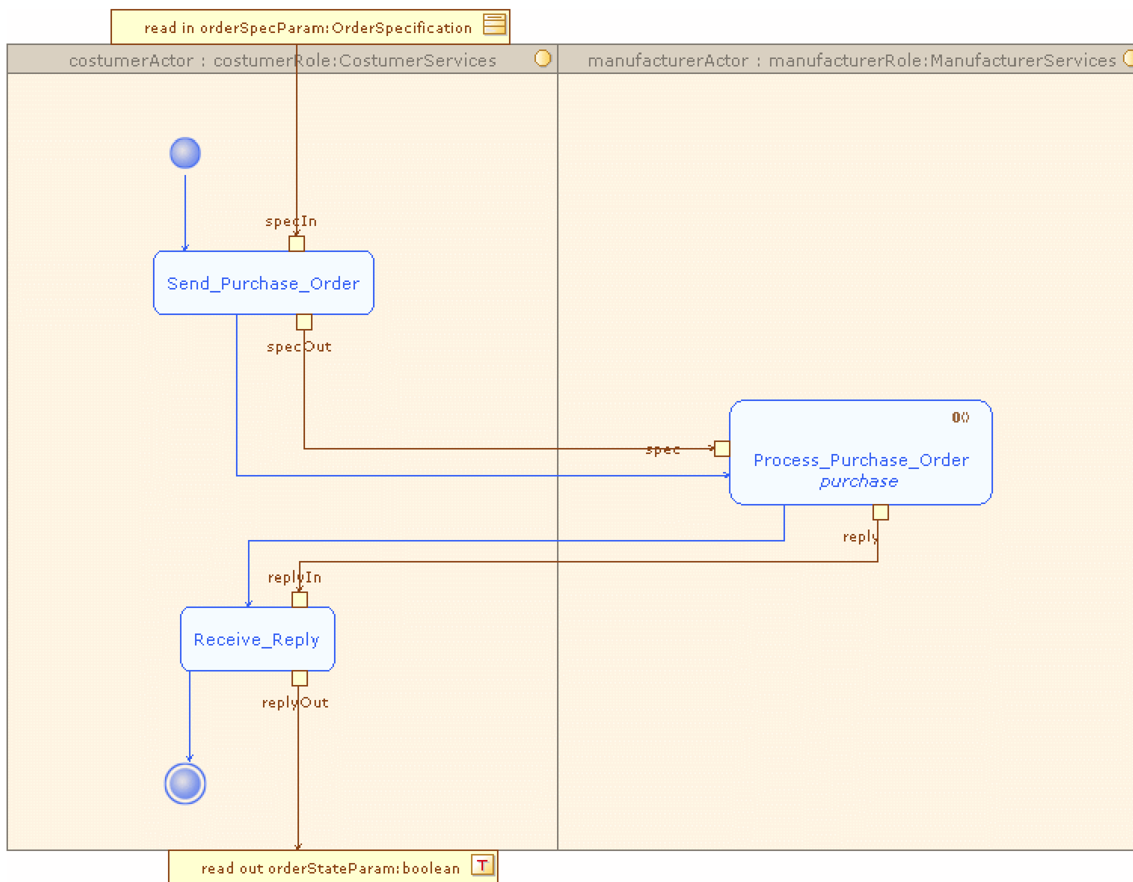


Figure 9.7: Activity diagram for “Purchasing Process” [CEG+09]

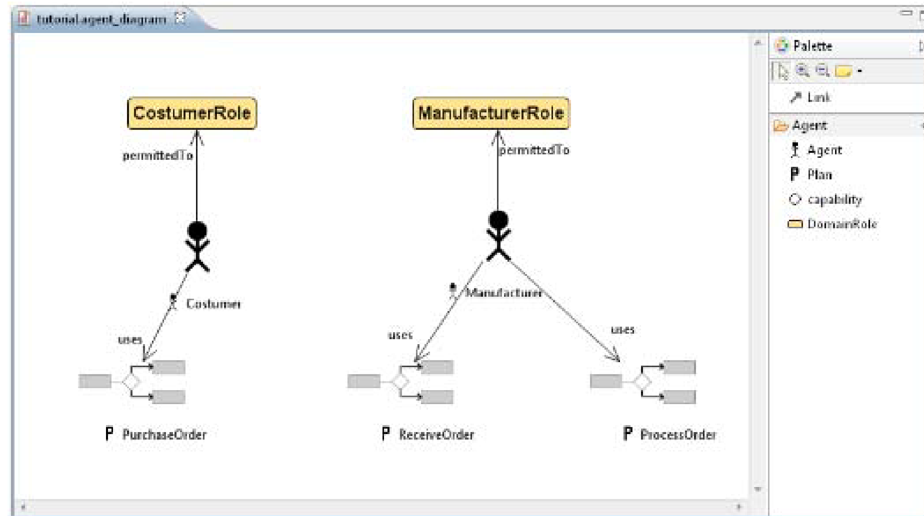


Figure 9.8: Agent diagram [CEG+09]

fined: *Customer* and *Manufacturer*. The *Customer* want to purchase some goods that the *Manufacturer* produces. They play the *CustomerRole* and *ManufacturerRole* respectively. The diagram also presents the plans that the agents have available in order to perform their tasks. For instance, the *PurchaseOrder* enables the *Customer* to purchase a product from a *Manufacturer*.

The *CustomerRole* and *ManufacturerRole* constitute the *Customer Manufacturer Network*. The organization diagram for the network is presented in Figure 9.9. The organization uses the *Ordering Contract* protocol and each of its *Domain Roles* are bound to *Actors* in the protocol. The mapping of the *Domain Roles* to the protocol *Actors* is depicted in Figure 9.10. The *ManufacturerServices_Role* and *CustomerServices_Role* from the Saarstahl scenario are bound to the generic *Manufacturer* and *Costumer Actors* in the *OrderingContract* protocol.

From the this view of the *Customer Manufacturer Network*, we move on to the general view of the complete *Saarstahl Architecture*. This overview is presented in the organization diagram in Figure 9.11. While not all *Participants* shown in Figure 9.3 are present, the correspondence between the elements generated by the model transformation is clear. The transformed *Participants* are bound to their responsibilities through their *Domain Roles*. As previously mentioned, the *Plans* linked to the (sub-)organizations representing the *Participants* implement how these responsibilities are to be

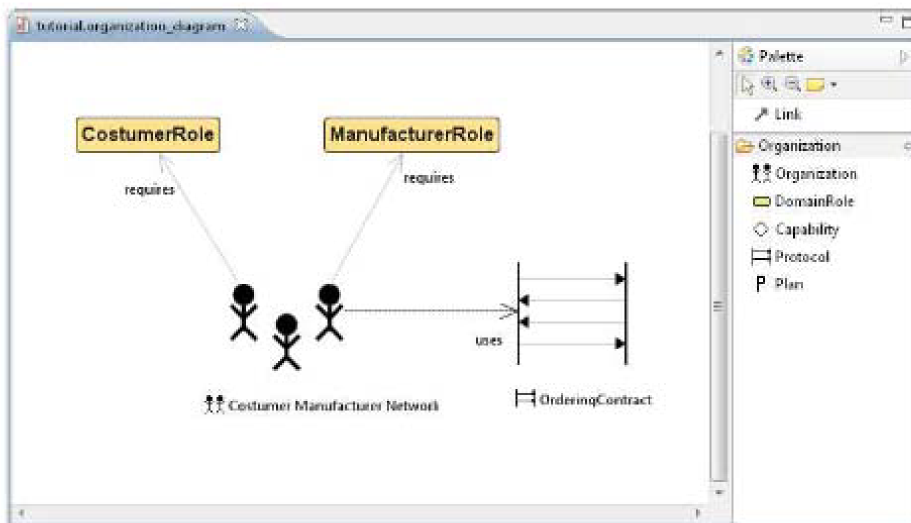


Figure 9.9: Organization diagram for the Costumer-Manufacturer Network [CEG+09]

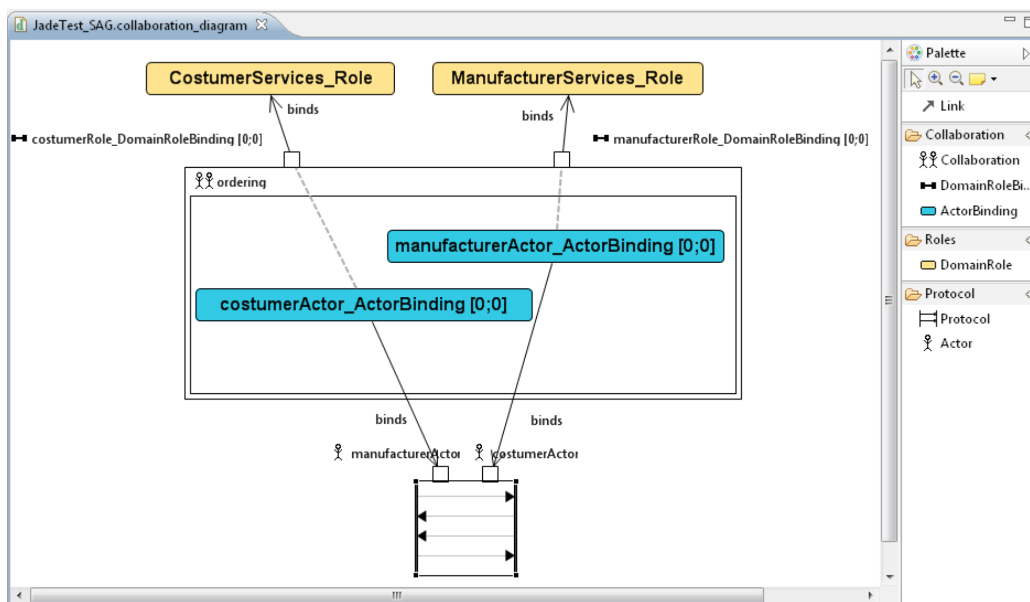


Figure 9.10: Ordering collaboration diagram [CEG+09]

fulfilled.

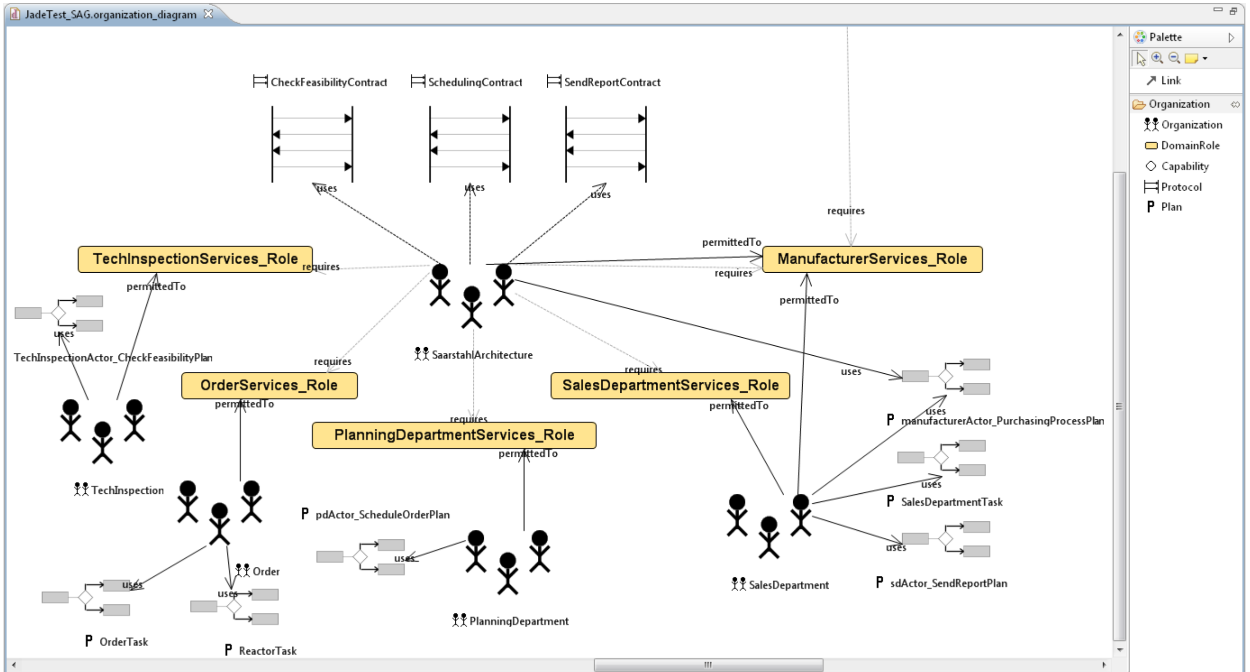


Figure 9.11: Organization diagram for the Saarstahl Architecture [CEG+09]

9.5 Scenario Evaluation

The evaluation of the scenario involves considering how well the technology developed meets the requirements at each of the abstraction level of the system. As mentioned in [EHJL10], performing a complete evaluation of every aspect could be a separate project its own. Nevertheless, this section presents a general evaluation as defined by the SHAPE Consortium.

The evaluation is performed in conjunction with the system stakeholders from the industrial partner Saarstahl and it is therefore performed in two stages (see Figure 9.12):

1. the system architect evaluates if the solution works and if it helps the current IT infrastructure,
2. if the previous stage has a positive result, the administration can evaluate the investment in terms of return on investment (ROI).

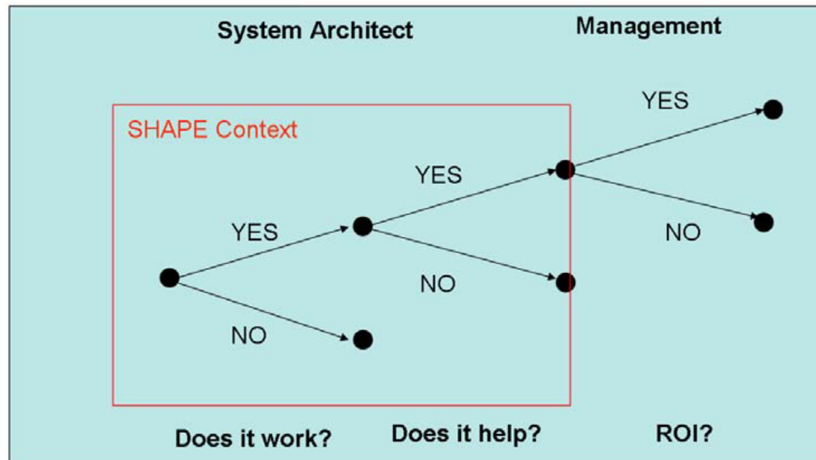


Figure 9.12: Evaluation framework [EHJL10]

With respect to stage 1, we can analyze each level of the levels presented in Figure 9.2 from the point of view of a user:

CIM level CIMFlex allows users without an IT/technical background to model the business processes in an abstract fashion

CIM to PIM This transformation creates a SoaML skeleton model that requires refinement.

PIM level
1. SoaML provides an abstract view of the system as a service oriented architecture.
2. The SoaML to PIM4Agents transformation generates a full PIM4Agents model in which the SoaML participants are mapped into PIM4Agents Organizations and Agents.
3. PIM4Agents allows further refinement of the system model using multiagent system concepts.

PIM to PSM The model transformation generates a JadeOrgs model that can be executed in JADE. Small refinements/editions are needed to get a fully executable model.

The SHAPE technologies enabled Saarstahl to model their processes and generate code that reflects these modeled processes. In particular, they were

able to model the process from order entry to order acceptance after performing an order feasibility check. This shows that the SHAPE technologies work in this scenario.

Saarstahl also successfully managed to integrate an existing legacy system into the Saarstahl architecture in SoaML. This was achieved by wrapping the legacy system behind a Web Service and obtaining a successful communication with the software components generated with the SHAPE toolkit. This demonstrates that the SHAPE technologies helps the integration of current IT infrastructure.

The analysis of stage 2 should consider the impact of the use of the SHAPE technologies in the costs of the software development process by addressing improvement in areas such as:

1. overall project duration, effort (man-hours)
2. number of coordination/analysis meetings per feature/issue
3. system performance (SOA vs. existing environment)

The analysis of this stage is mainly to be performed by Saarstahl and may take a considerable amount of time and effort which we did not have available within the resources of the SHAPE project. Therefore this stage was not performed within the context of the SHAPE project [EHJL10].

9.5.1 SWOT Analysis

In order to summarize the results of the implementation of the SHAPE technologies in the context of the Saarstahl scenario a Strengths, Weaknesses, Opportunities and Threats (SWOT) analysis was performed. This analysis is summarized in Figure 9.13.

Strengths

Among the strengths, we find an enhanced interoperability of existing IT solutions, an improvement of the complete software development process and the wrapping of legacy systems behind SOA participants. Through the solution in this scenario, systems that were isolated from each other, such as the steelwork and rolling mill planning systems and various inventory systems are now able to communicate and propagate their changes so that following the production status of an order is more transparent.

Strengths	Weaknesses
• Improvement of complete software development process • Enhanced interoperability of existing IT solutions • Wrapping of legacy systems behind participants of Saarstahl SOA	• Missing bottom-up approach
Opportunities	**Threats**
• Improved communication between actors/roles in IT projects	• Integration of older techniques

Figure 9.13: SWOT analysis of SHAPE technologies in the Saarstahl scenario [EHJL10]

In order to achieve this interoperability, various legacy systems were wrapped under a Web service interface. Thus enabling a faster solution deployment that reimplementing those systems, but at the same time ensuring than when the time comes to replace any of these legacy systems the transition will be easier since the web service interfaces to be fulfilled by the new system are already defined.

As a general strength, the overall software process has been improved. The use of models not only support the generation of the system's code, but has allowed the different parties involved in the technical specification of the system to have a better, shared understanding of what the system does.

Weaknesses

In spite of the strengths, one weakness has been identified: the lack of a bottom-up approach. The approach developed in SHAPE starts at the top level with CIM modeling and through transformations this information is transferred and refined at the PIM, PSM and code levels.Nevertheless, the integration of already existing participants, like legacy systems, has to be

performed by modeling them at the CIM level. Thus requiring to 'manually' preserve the consistency between the existing systems and their model counterparts. Perhaps a reverse engineering approach could assist in generating the model artifacts at the various abstraction levels.

Opportunities

The application of our approach has presented a big opportunity to improve the communication between the different stakeholders. This opportunity is very closely related to one of the strengths but it is not focused on the technical issues. The improvement in communication is particularly visible in the interactions with non-technical stakeholders. They are now able to provide richer feedback from the business perspective based on the increased understanding they have achieved.

Threats

The main threat we have identified relates to the reliance on a few modern techniques and languages. Even though a model driven approach, like ours, can always be extended to additional languages and platforms, extending such a system implies an additional effort and risk. These should be considered when applying such an approach, since it is not uncommon to find a great variety in the technological landscape of today's enterprise.

The SWOT analysis has presented us with a overview of the performance of the approach in this scenario and it has helped identify the areas where it can be improved.

9.6 Summary

In this chapter, we have presented the application of a model driven approach of multiagent system technologies to the industrial production of steel. The scenario presented examples on how different production systems can be interconnected through a SOA, and how a multiagent system can play a coordination role in the tracking of orders and the production stages related to each order.

The scenario was modeled at the CIM level (ARIS EPC and BPMN) and then transformed and refined to the PIM level (PIM4Agents) and PSM level (JadeOrgs). The generated system was then evaluated in how far the

solutions works and how much it helps the current IT infrastructure. In addition, a Strengths, Weaknesses, Opportunities and Threats analysis was performed in order to identify the areas where the current situation of the SHAPE approach and how it could be improved in the future.

Chapter 10

Conclusions and Future Work

10.1 Summary

In this thesis, we have introduced a new model-driven approach to agent-oriented software engineering in which organizations not only play a crucial role, but are also represented in every abstraction level, even runtime. In this approach, MAS are modeled at the PIM level and transformed into PSM level preserving the organizational structures.

This approach evolved from the perceived need of a common abstract language to model MAS. In project ATHENA, we modeled SOAs at the PIM level and transformed directly into PSMs related to different agent platforms. As these PSMs had lots of commonalities, we extracted the common features into an agent PIM, namely the PIM4Agents. Therefore, the change of domain from the application domain to the MAS domain is performed via a transformation an equivalent abstraction level (PIM-to-PIM transformation) and then the MAS PIM is mapped to the desired PSM(s).

In order to assist the system designer to model effectively, we described a methodology that guides the creation of the various model views in a fashion consequent with the dependencies between these views. Depending on the system requirements or designer preference, the system designer has the option to model the system in a goal-driven or a behavior-driven fashion. In the goal-driven fashion, the responsibilities for each role are modeled as goals. These goals include system overall goals, organization goals and the goals for each agent type. In the behavior-driven way, the achievement of goals is implicit in the successful completion of the behaviors and the roles in the system depend on their required behaviors. Aside for the goal definition stages, the methodology is the same for both variations by defining: the information model, the roles with respect to the responsibilities (goals or behaviors), the organizational structures and their relationships through roles, the communication protocols, the detailed process entailed by each behavior, and the deployment configuration of the MAS.

The PIM4Agents modeling tools support each of these methodology stages and provide a graphical model for each of the views. In addition to the graphical modeling support, model validation is performed on the saved models. The validation avoids the introduction of inconsistencies in the model and guarantees that the following transformations will work successfully.

Once a target agent platform is chosen, the models are transformed into a

PSM for the given platform. In this thesis, we introduced a PSM for the Jade Agent Platform, JadeOrgs. JadeOrgs provides the modeling constructs available in the agent platform and extends this set with the constructs necessary to represent the organizational structures, their roles and their responsibilities. In addition, a definition for these structures was formalized using the Object-Z specification language. As these constructs are not implemented in the Jade Agent platform, JadeOrgs also includes a programming API and a runtime component so that these structures are also available during the execution of the modeled MAS. In order to connect the different abstraction levels, a series of transformations were defined. They consisted in a series of maps of concepts from one abstraction level to the next and in a set of code templates in the serialization stage of the PSM to Java code.

In order to prove the viability of such models and transformations, first an early version of JadeOrgs was applied to a proof of concept in the context of service oriented architectures. From this experience, it was determined that a agent-oriented PIM metamodel would be beneficial and eventually led to the definition of the PIM4Agents. Once the PIM level metamodel was further developed, along with the transformations and graphical modeling tools, the complete approach was applied to a scenario in steel production. The implementation of the scenario proved successful and generated valuable feedback that has been integrated into following version of the tools, metamodel definitions and transformations.

10.2 Future Work

In spite of the success had in the scenarios presented in this thesis, the approach and tools need further validation and can still be extended to address a variety of issues and topics that were deemed beyond the scope of this work. In the following list, we suggest some of these topics that could prove interesting as further research. This list is not comprehensive, but represents a sample of the possible directions in which this work could be improved or extended.

10.2.1 Role Deployment Dynamics

The runtime components in JadeOrgs enable the construction and modification of organizational structures programatically. The current modeling

approach takes a somewhat static view on the definition of roles and organizations. It would be interesting to model and perhaps regulate what kind of dynamics can be incorporated:

1. Can roles change in time as long as the parties involved agree upon it?
2. Can role responsibilities be modified in a running system? Under what conditions?
3. How would a role negotiation take place?
4. Does the organization "own" the roles and therefore the members either comply to the new roles or leave the organization?

Also under the role dynamics, issues related to service level agreements could be addressed. As the terms of such agreements change in time, how could these changes be propagated in a running system while guaranteeing that processes that were already in progress comply to the terms that were valid when the process was started.

10.2.2 Norms and Electronic Institutions

The role that norms play in organizational structures in MAS is an ongoing topic of research following seminal works such as Electronic Institutions [Est02, RA03, Bog07]. We have already sketched how a such norms could be included in our metamodels. Figure 10.1 presents how such norms could be modeled as part of JadeOrgs.

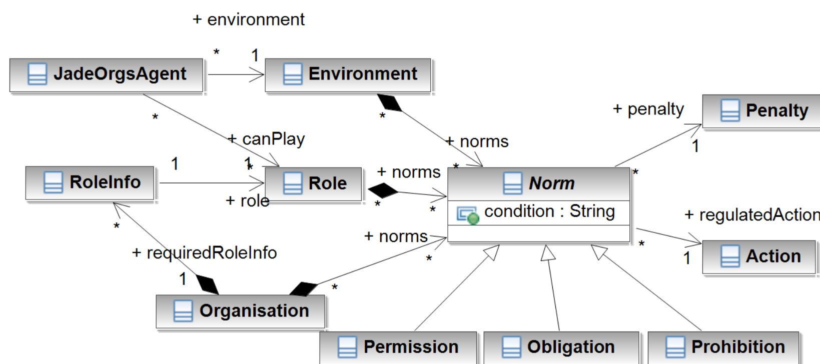


Figure 10.1: Sketch of normative view for JadeOrgs

The interesting issue would not be to just model the norms at PIM and PSM level, but determine a way to enforce them in the targeted agent platforms while still preserving agent and organization autonomy and, hopefully, without requiring a complete redesign of the agent platform itself.

10.2.3 Bottom-up Approach

During the development of the Saarstahl scenario, we noted that the consistency between existing systems or modules, and what is modeled at the CIM or PIM levels is only guaranteed through careful modeling. While care and attention to detail will always be critical to modeling and other development stages, it would be easier if we could reverse engineer current existing systems to guarantee that the modeled interfaces to these systems comply to the actual interfaces that these systems offer. These is also relevant when encapsulating legacy systems behind web services.

The issue here is to automate the selection of relevant features with respect to the level of abstraction desired. Every time the level of abstraction is raised, there is some information loss given that the more abstract concepts are more general than the more specific ones. Perhaps there is no optimal solution for this problem in every scenario, but perhaps general guidelines could be found and a customizable framework be implemented so that this feature extraction can be customized to the application domain in question.

10.2.4 Other Application Domains

In this thesis, we only applied the proposed approach to proof-of-concept scenarios and one industrial scenario in the context of steel production. The combination of MAS and SOAs is potentially a viable solution to scenarios where there are multiple parties involved and coordination is critical, or situations where resource availability or cost is dynamic. Therefore, it would be interesting to further validate our tools in areas in which we have some experience, but because of time constraints did not carry it out in full:

Travel reservation The booking of lodging and travel arrangements always include a series of variables and preferences that need to be satisfied to fulfill the user's requirements. These may include cost, duration, time of day, location of lodging with respect to desired activities, room type preference, airplane sitting preference, preferred loyalty programs,

etc. The amount of variables and available offers make the automatic composition of a travel solution a non-trivial task with very practical commercial application.

Negotiation and instantiation of service-level agreements Given the desire for an agent organization that implements certain terms of service between the involved parties, how can these service level agreements be modeled and how could changes and violations to these terms be handled.

Navigation with augmented reality on mobile devices If a user wants to find and book given service in his/her vicinity, could we model a MAS that would assist him in guiding this user to the location, fulfilling his time and cost constraints, and making a reservation for him with minimum interaction. This would go further than guiding the user's decision process, like Apple's Siri[1], but would consider in the background issues like: time constraints as defined in the user's agenda/calendar and the services time availability (i.e. a restaurant's opening times), or the user's preferences (i.e. dietary needs in the case of a restaurant search, or movie genre in case of a movie theater search) and perform a prebooking for these services and present the user with a list of viable solutions. Such a result list can be then displayed over the map on the navigation system or as a heads-up-display using the mobile device's integrated camera.

[1] http://www.apple.com/iphone/features/siri.html

List of Figures

2.1 Relation among models, metamodels and platforms (based on [MKUW04]) . . . 23
2.2 Example of a CIM, PIM and PSM . . . 26

4.1 Overview of the model-driven approach with PIM4Agents . . . 38

5.1 The Multiagent System View of the PIM4Agents. . . . 49
5.2 The metamodel reflecting the agent aspect of the PIM4Agents. 49
5.3 The metamodel reflecting the organizational aspect of the PIM4Agents. . . . 51
5.4 The metamodel reflecting the goal aspect of the PIM4Agents. 53
5.5 The metamodel reflecting the role aspect of the PIM4Agents. . 54
5.6 The partial metamodel reflecting the behavior aspect of the PIM4Agents. . . . 56
5.7 The hierarchy of StructuredActivities. . . . 57
5.8 The hierarchy of Tasks. . . . 57
5.9 The partial metamodel reflecting the partial interaction aspect of the PIM4Agents. . . . 60
5.10 The partial metamodel reflecting the information model view of the PIM4Agents. . . . 62
5.11 The Condition hierarchy. . . . 63
5.12 The metamodel for the Deployment View of PIM4Agents . . . 64
5.13 The abstract goal decomposition for ManagePaperSubmission 66
5.14 The concrete goal decomposition for ManagePaperSubmission 67
5.15 The Information Model for the CMS . . . 69
5.16 The Role View for the CMS example . . . 70
5.17 The Organization View for the CMS example . . . 72
5.18 The Agent View for the CMS example . . . 74

5.19 The PaperReviewCollaboration for the PC organization 75
5.20 The RequestReview protocol 76
5.21 The ManagePaperSubmissions plan 79
5.22 The CMS deployment view 80

6.1 View of the Project package of the JadeOrgs metamodel . . . 97
6.2 The Core of the JadeOrgs metamodel 98
6.3 Partial view of the Behavior class hierarchy 100
6.4 Representation of the FSMBehaviour 101
6.5 The JadeOrgs Process package 102
6.6 The JadeOrgs Ontology 103
6.7 The JadeOrgs Goals 105
6.8 The Deployment View in JadeOrgs 106
6.9 Agent Description in JADE 107
6.10 The RoleFillerRequest protocol 108
6.11 The TaskRequest protocol 109
6.12 The Product Sale Protocol 111
6.13 Organizational structures for the Product Sale scenario 112
6.14 Loan Application protocol 113
6.15 Instance distribution of the scenario (initial state) 115
6.16 Example of organization structure instances: Two Organizations bound to N Agents through 4 roles 118

7.1 Partial views of the JadeOrgs CMS model 126
7.2 Extract from the Method2Java text transformation 127

8.1 Overview of the AIF technical framework [BEF+07] 134
8.2 The service metamodel of the PIM4SOA. 136
8.3 The process metamodel of the PIM4SOA. 138
8.4 Behaviour, Scope and the Steps inheritance hierarchy. 139
8.5 The overall picture: From service-oriented architectures to agent systems using MDA standards. 141
8.6 The example illustrates the service model. 150
8.7 The example illustrates the process model. 151
8.8 The example after the transformation to JADE was applied. . 152

9.1 Saarstahl departments involved in the Steelwork-Rolling Mills Scenario [CEG+09] 157

9.2 Model transformations from business models to heterogeneous executable platforms [CEG+09] 160
9.3 Saarstahl Architecture in SoaML [CEG+09] 162
9.4 Order agent specification [CEG+09] 163
9.5 Event handling performed by the Order agent [CEG+09] . . . 165
9.6 Melting request performed by the Order agent [CEG+09] . . . 166
9.7 Activity diagram for "Purchasing Process" [CEG+09] 167
9.8 Agent diagram [CEG+09] 168
9.9 Organization diagram for the Costumer-Manufacturer Network [CEG+09] 169
9.10 Ordering collaboration diagram [CEG+09] 169
9.11 Organization diagram for the Saarstahl Architecture [CEG+09] 170
9.12 Evaluation framework [EHJL10] 171
9.13 SWOT analysis of SHAPE technologies in the Saarstahl scenario [EHJL10] 173

10.1 Sketch of normative view for JadeOrgs 179

Bibliography

[AFV04] Mercedes Amor, Lidia Fuentes, and Antonio Vallecillo. Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA. In *Agent-Oriented Software Engineering (AOSE-2004)*, number 3382 in Lecture Notes in Computer Science, pages 93–108, 2004.

[Age05] Agentlink III AOSE Technical Forum Group. Methodologies evaluation. http://www.pa.icar.cnr.it/cossentino/al3tf3/docs/aose-evaluation.ppt (Accesssed March, 2009), September 2005.

[AOS06] AOS. JACK Intelligent Agents, The Agent Oriented Software Group (AOS), http://www.agent-software.com/shared/home/, 2006.

[ATL06] ATLAS Group, INRIA & LINA, University of Nantes. INRIA, ATL - The Atlas Transformation Language Home Page, http://www.sciences.univ-nantes.fr/lina/atl/, 2006.

[Aus62] John L. Austin. *How to do things with words.* Harvard U.P., Cambridge, Mass., 1962.

[Bau02] Bernhard Bauer. UML Class Diagrams revisited in the context of agent-based systems. In *Agent-Oriented Software Engineering II: Second International Workshop, AOSE 2001*, Lecture Notes in Computer Science 2222, page 101118. Springer, 2002.

[BBC+10] Gorka Benguria, Arne J. Berre, Davide Cerri, Brian Elvesæter, Klaus Fischer, Christian Hahn, Sven Jacobi, Einar Landre, Dima Panfilenko, Andrey Sadovykh, and Michael Stollberg.

SHAPE Whitepaper. http://www.shape-project.eu/wp-content/uploads/2010/05/shape_whitepaper.pdf, January 2010.

[BBG+08] Matteo Baldoni, Guido Boella, Valerio Genovese, Roberto Grenna, and Leendert van der Torre. How to Program Organizations and Roles in the JADE Framework. In *MATES*, pages 25–36, 2008.

[BCG+04] Carole Bernon, Massimo Cossentino, Marie Pierre Gleizes, Paola Turci, and Franco Zambonelli. A study of some multi-agent meta-models. In James Odell, Paolo Giorgini, and Jörg P. Müller, editors, *AOSE*, volume 3382 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2004.

[BEF+07] Arne-Jørgen Berre, Brian Elvesæter, Nicolas Figay, Claudia Guglielmina, S. Johnsen, Dag Karlsen, Thomas Knothe, and S. Lippe. The ATHENA Interoperability Framework. In Ricardo Jardim-Gonçalves, Jörg P. Müller, Kai Mertins, and Martin Zelm, editors, *IESA*, pages 569–580. Springer, 2007.

[BGPLHS05] Ghassan Beydoun, Cesar Gonzalez-Perez, Graham Low, and Brian Henderson-Sellers. Synthesis of a generic mas meta-model. In *SELMAS '05: Proceedings of the fourth international workshop on Software engineering for large-scale multi-agent systems*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[BGPP02] Carole Bernon, Marie Pierre Gleizes, Sylvain Peyruqueou, and Gauthier Picard. Adelfe: A methodology for adaptive multi-agent systems engineering. In Paolo Petta, Robert Tolksdorf, and Franco Zambonelli, editors, *ESAW*, volume 2577 of *Lecture Notes in Computer Science*, pages 156–169. Springer, 2002.

[BMO00] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In Paolo Ciancarini and Michael Wooldridge, editors, *AOSE*, volume 1957 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 2000.

[BMO01] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[Bog07] Anton Bogdanovych. *Virtual Institutions*. PhD thesis, University of Technology, Sydney, Australia, 2007.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. TROPOS: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agents and Multiagent Systems*, 8(3), 2004.

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, 1999.

[Bra90] Michael E. Bratman. What is intention? In Philip R. Cohen, Jerry L. Morgan, and Martha E. Pollack, editors, *Intentions in Communication*, pages 15–32. The MIT Press, Cambridge, MA, June 1990.

[Bro90] Rodney A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1&2):3–15, June 1990.

[Bro91] Rodney A. Brooks. Intelligence without reason. In Ray Myopoulos, John; Reiter, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, Australia, August 1991. Morgan Kaufmann.

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[CEG+09] Davide Cerri, Brian Elvesæter, Birgit Grammel, Christian Hahn, Sven Jacobi, Einar Landre, Marcel Muth, and Dima Panfilenko. Case Study Execution and Validation, Interim Report, Work Package 1. Deliverable D1.3, SHAPE Project, European Commission, 7th Framework Programme, December 2009.

[Cos05a] M. Cossentino. From requirements to code with the PASSI methodology. In B. Henderson-Sellers and P. Giorgini, editors, *Agent-Oriented Methodologies*, Hershey, PA, USA, 2005. Idea Group Inc.

[Cos05b] Massimo Cossentino. Methodology evaluation questionnaire. http://www.pa.icar.cnr.it/cossentino/al3tf3/docs/questionnaire.doc (Accesssed March, 2009), 2005.

[CTCG04] Radovan Cervenka, Ivan Trencanský, Monique Calisti, and Dominic A. P. Greenwood. AML: Agent Modeling Language Toward Industry-Grade Agent-Based Modeling. In *Agent-Oriented Software Engineering (AOSE-2004)*, number 3382 in Lecture Notes in Computer Science 3382, pages 31–46, Berlin et al., 2004. Springer.

[DD01] Virginia Dignum and Frank Dignum. Modelling Agent Societies: Co-Ordination Frameworks and Institutions. In *Progress in Artificial Intelligence, LNAI 2258*, pages 191–204. Springer-Verlag, 2001.

[DeL91] Scott A. DeLoach. Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems. In *Agent-Oriented Information Systems '99 (AOIS'99)*, Seattle WA, May 1991.

[DeL02] Scott A. DeLoach. Modeling organizational rules in the multiagent systems engineering methodology. In Robin Cohen and Bruce Spencer, editors, *Canadian Conference on AI*, volume 2338 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.

[DeL05] Scott A. DeLoach. Engineering organization-based multiagent systems. In Alessandro F. Garcia, Ricardo Choren, Carlos José Pereira de Lucena, Paolo Giorgini, Tom Holvoet, and Alexander B. Romanovsky, editors, *SELMAS*, volume 3914 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2005.

[DeL07] Scott A. DeLoach. Developing a multiagent conference management system using the o-mase process framework. In Luck and Padgham [LP08], pages 168–181.

[DGO10] Scott A. DeLoach and Juan C. García-Ojeda. O-mase: a customisable approach to designing and building complex, adaptive multi-agent systems. *IJAOSE*, 4(3):244–280, 2010.

[DR00] Roger Duke and Gordon Rose. *Formal object-oriented specification using Object-Z*. Macmillan, Basingstoke :, 2000.

[DS83] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[D'S01] D. D'Souza. Model-Driven Architecture and Integration - Opportunities and Challenges, Version 1.1, Kineticum, 2001.

[EdlCS02] Marc Esteva, David de la Cruz, and Carles Sierra. Islander: an electronic institutions editor. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, AAMAS '02, pages 1045–1052, New York, NY, USA, 2002. ACM.

[EHJL10] Brian Elvesæter, Christian Hahn, Sven Jacobi, and Einar Landre. Case Study Execution and Validation, Final Report, Work Package 1. Deliverable D1.4, SHAPE Project, European Commission, 7th Framework Programme, May 2010.

[Est02] Marc Esteva. ISLANDER: an electronic institutions editor. In *In First International Conference on Autonomous Agents and Multiagent systems*, pages 1045–1052. ACM Press, 2002.

[Fer92] Innes A. Ferguson. Touring machines: Autonomous agents with attitudes. *Computer*, 25(5):51–55, 1992.

[FFMM94] Timothy W. Finin, Richard Fritzson, Donald P. McKay, and Robin McEntire. Kqml as an agent communication language. In *CIKM*, pages 456–463. ACM, 1994.

[FG98] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98)*, pages 128–135, 1998.

[FHMM07] Klaus Fischer, Christian Hahn, and Cristián Madrigal-Mora. Agent-oriented software engineering: a model-driven approach. *Int. J. Agent-Oriented Software Engineering*, 1(3/4):334–369, 2007.

[Fou01] Foundation for Intelligent Physical Agents. *FIPA ACL Message Structure Specification (fipa00061).* FIPA, 2001.

[Fou02a] Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification. Document number SC00061G. http://www.fipa.org/specs/fipa00061/SC00061G.html, 2002.

[Fou02b] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. Document number SC00037J. http://www.fipa.org/specs/fipa00037/SC00037J.html, 2002.

[Fou02c] Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification. Document number SC00029H. http://www.fipa.org/specs/fipa00029/SC00029H.html, 2002.

[Fou02d] Foundation for Intelligent Physical Agents. FIPA Request Interaction Protocol Specification. Document number SC00026H. http://www.fipa.org/specs/fipa00026/SC00026H.html, 2002.

[Fou02e] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification. Document number SC00008I. http://www.fipa.org/specs/fipa00008/SC00008I.html, 2002.

[Gen91] Michael R. Genesereth. Knowledge Interchange Format. In *KR*, pages 599–600, 1991.

[GF92] Michael R. Genesereth and Richard E. Fikes. Knowledge Interchange Format Version 3.0 Reference Manual. Technical

Report Logic-92-1, Computer Science Department, Stanford University, 1992.

[GL87] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *AAAI*, pages 677–682, 1987.

[GPP+98] Michael P. Georgeff, Barney Pell, Martha E. Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In Müller et al. [MSR99], pages 1–10.

[Gue05] Zahia Guessoum. MAS Meta-Models and MDA, AgentLink III AOSE TFG2. Online at: http://www.pa.icar.cnr.it/c̃ossentino/al3tf2/docs/zahia_slovenia.pdf, 2005.

[HBSS00] Mahdi Hannoun, Olivier Boissier, Jaime Simão Sichman, and Claudette Sayettat. MOISE: An Organizational Model for Multi-agent Systems. In *IBERAMIA-SBIA*, pages 156–165, 2000.

[Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *J. Symb. Log.*, 16(1):14–21, 1951.

[HSB02] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. MOISE+: towards a structural, functional, and deontic model for MAS organization. In *AAMAS*, pages 501–502. ACM, 2002.

[JK05] Frederic Jouault and Ivan Kurtev. Transforming Models with ATL. In *MoDELS 2005, Montego Bay, Jamaica*, 2005.

[Lin01] Jürgen Lind. *Iterative Software Engineering for Multiagent Systems: The MASSIVE Method*, volume 1994 of *Lecture Notes in Computer Science*. Springer, 2001.

[LP08] Michael Luck and Lin Padgham, editors. *Agent-Oriented Software Engineering VIII, 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, volume 4951 of *Lecture Notes in Computer Science*. Springer, 2008.

[LTB+08] Einar Landre, Knut Tungeland, Arne-Jørgen Berre, Brian Elvesæter, Sven Jacobi, Christian Hahn, Stefan Warwas, and Sebastian Kämper. Case study scenario descriptions and requirements specifications, Work Package 1. Deliverable D1.1, SHAPE Project, European Commission, 7th Framework Programme, December 2008.

[LTB+09] Einar Landre, Knut Tungeland, Arne-Jørgen Berre, Brian Elvesæter, Sven Jacobi, Christian Hahn, Stefan Warwas, and Sebastian Kämper. Case Study Execution and Validation Ð Initial Report, Work Package 1. Deliverable D1.2, SHAPE Project, European Commission, 7th Framework Programme, January 2009.

[MKUW04] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[MLF95] James Mayfield, Yannis Labrou, and Timothy W. Finin. Evaluation of kqml as an agent communication language. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *ATAL*, volume 1037 of *Lecture Notes in Computer Science*, pages 347–360. Springer, 1995.

[MMLSF08] Cristián Madrigal-Mora, Esteban León-Soto, and Klaus Fischer. Implementing Organisations in JADE. In *MATES*, pages 135–146, 2008.

[MNP+07] Mirko Morandini, Duy Cu Nguyen, Anna Perini, Alberto Siena, and Angelo Susi. Tool-supported development with tropos: The conference management system case study. In Luck and Padgham [LP08], pages 182–196.

[MP93] Jörg P. Müller and Markus Pischel. The agent architecture inteRRaP: Concept and application. Technical report, German Research Center for Artificial Intelligence, 1993. RR 93-26.

[MS06] Pavlos Moraitis and Nikolaos I. Spanoudakis. The Gaia2Jade Process for Multi-Agent Systems Development. *Applied Artificial Intelligence*, 20(2-4):251–273, 2006.

[MSR99] Jörg P. Müller, Munindar P. Singh, and Anand S. Rao, editors. *Intelligent Agents V, Agent Theories, Architectures, and Languages, 5th International Workshop, ATAL '98, Paris, France, July 4-7, 1998, Proceedings*, volume 1555 of *Lecture Notes in Computer Science*. Springer, 1999.

[Mül98] Jörg P. Müller. The right agent (architecture) to do the right thing. In Müller et al. [MSR99], pages 211–225.

[Obj03] Object Management Group (OMG). MDA Guide Version 1.0.1, Document omg/03-06-01, June 2003, http://www.omg.org/docs/omg/03-06-01.pdf, June 2003.

[Obj04] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Core Specification, Document ptc/04-10-15, October 2004, http://www.omg.org/docs/ptc/04-10-15.pdf, October 2004.

[ONL04] James Odell, Marian H. Nodine, and Renato Levy. A metamodel for agents, roles, and groups. In *AOSE*, pages 78–92, 2004.

[OPF03] James Odell, H. Van Dyke Parunak, and Mitchell Fleischer. Modeling agent organizations using roles. *Software and System Modeling*, 2(2):76–81, 2003.

[PBL05] A. Pokahr, L. Braubach, and W. Lamersdorf. *Jadex: A BDI Reasoning Engine*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 6, pages 149–174. Springer, Berlin et al., 2005.

[PGS03] Juan Pavón and Jorge J. Gómez-Sanz. Agent oriented software engineering with ingenias. In Vladimír Marík, Jörg P. Müller, and Michal Pechoucek, editors, *CEEMAS*, volume 2691 of *Lecture Notes in Computer Science*, pages 394–403. Springer, 2003.

[PGSF06] Juan Pavón, Jorge J. Gómez-Sanz, and Rubén Fuentes. Model Driven Development of Multi-Agent Systems. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of

Lecture Notes in Computer Science, pages 284–298. Springer, 2006.

[PH01] M. Papasimeon and C. Heinze. Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, 2001.

[PL07] Lin Padgham and Michael Luck. Introduction to aose tools for the conference management system. In Luck and Padgham [LP08], pages 164–167.

[PTW07] Lin Padgham, John Thangarajah, and Michael Winikoff. The prometheus design tool - a conference management system case study. In Luck and Padgham [LP08], pages 197–211.

[RA03] Juan A. Rodríguez-Aguilar. *On the Design and Construction of Agent-Mediated Electronic Institutions*. PhD thesis, IIIA-CSIC, Bellaterra, Spain, 2003.

[Rab09] David Raber. A Model-Driven Approach for the Integration of Multiagent Systems and Service-Oriented Architectures in the Steel Industry. Master's thesis, Saarland University, December 2009.

[RG95] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In Victor R. Lesser and Les Gasser, editors, *ICMAS*, pages 312–319. The MIT Press, 1995.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[Sea69] John R. Searle. *Speech acts : an essay in the philosophy of language / John R. Searle*. Cambridge University Press, London :, 1969.

[SF03] Michael Schillo and Klaus Fischer. Holonic multiagent systems. *KI*, 17(4):54–55, 2003.

[SIN06] SINTEF ICT. MOFScript, http://www.eclipse.org/gmt/mofscript, 2006.

[Smi00] Graeme Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[SO04] J. M. Serrano and S. Ossowski. On the impact of agent communication languages on the implementation of agent systems. In *Proceedings of the Eight International Workshop CIA 2004 on Cooperative Information Agents*, volume 3191 of *Lecture Notes in Computer Science*, pages 92–106, Berlin et al., 2004. Springer.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[TC05] I. Trencansky and R. Cervenka. Agent modeling language (AML): A comprehensive approach to modeling MAS. *Informatica*, 29(4):391–400, 2005.

[VBE+10] Julien Vayssiére, Gorka Benguria, Brian Elvesæter, Klaus Fischer, and Ingo Zinnikus. *Rapid Prototyping for Service-Oriented Architectures*, pages 93–106. ISTE, 2010.

[vRDW08] M. Birna van Riemsdijk, Mehdi Dastani, and Michael Winikoff. Goals in agent systems: a unifying framework. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 713–720, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[Wei00] Gerhard Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 2000.

[WH08] Stefan Warwas and Christian Hahn. The contrete syntax of the platform independent modeling language for multiagent systems. In *Proceedings of the Agent-based Technologies and applications for enterprise interOPerability (ATOP 2008). Workshop to be held at the Seventh International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2008)*, 2008.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.

[WJK00] M. Wooldridge, N.R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[Woo02] Michael Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons, 1st edition, June 2002.

[ZJW01] F. Zambonelli, N.R. Jennings, and M.J. Wooldridge. Organizational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 11:303–328, 2001.

[ZJW03] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: the Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):417–470, 2003.