Dissertation zur Erlangung des Grades des Doktors der Ingenieurwissenschaften der Naturwissenschaftlich-Technischen Fakultät der Universität des Saarlandes

A Model-driven Framework for Engineering Multiagent Systems

vorgelegt von

Stefan Helmut Warwas

Saarbrücken, 11. März 2013

SAARLAND UNIVERSITY
COMPUTER SCIENCE

German Research Center for Artificial Intelligence GmbH

Prüfungsausschuss:

Dekan:	Prof. Dr. Mark Groves
Vorsitzender:	Prof. Dr. Jörg Siekmann
Berichterstatter:	Prof. Dr. Philipp Slusallek
	Prof. Dr. Jörg Müller
Akademischer Beisitzer:	Dr. Klaus Fischer

Tag des Promotionskolloquiums: 11. März 2013

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig 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 11. März 2013

(Stefan Warwas)

Meinen Großeltern gewidmet.

Abstract

This dissertation presents the BOCHICA framework for *Agent-Oriented Software Engineering* (AOSE). The framework's task in the software development process is (i) to capture the design decisions for a system under consideration on a platform-independent level of abstraction and (ii) to project this design to a target platform. BOCHICA goes beyond the state-of-the-art in AOSE as it combines the benefits of a platform-independent approach with the possibility to address concepts of custom application domains and execution environments. Several extension interfaces are specified to enable the customization of the underlying modeling language to the engineer's needs. BOCHICA is accompanied by an iterative adaptation process to gradually incorporate extensions. Conceptual mappings for projecting BOCHICA models to executable code are specified. In order to enable BOCHICA for modeling agents that inhabit semantically-enhanced virtual worlds, an according extension model is proposed. Finally, a model-driven reverse engineering approach for lifting the underlying design of already implemented *Multiagent System* (MAS) to the platform-independent layer is introduced. The framework has been successfully evaluated for designing intelligent agents that operate a virtual production line as well as for extracting the underlying design of an already implemented MAS. The evaluation results show that the BOCHICA approach to AOSE contributes to overcome the gap between design and code.

Kurzzusammenfassung

Diese Arbeit präsentiert das Bochica Rahmenwerk für agentenorientierte Softwareentwicklung. Die Aufgabe des Rahmenwerks ist es, die Designentscheidungen für ein IT-System auf einer plattformunabhängigen Ebene festzuhalten und auf eine Zielplattform abzubilden. Bochica erweitert den Stand der Wissenschaft der agentenorientierten Softwareentwicklung durch die Kombination von plattformunabhängigen und plattformspezifischen Eigenschaften. Zu diesem Zweck werden konzeptionelle Schnittstellen für die Anpassung an benutzerspezifische Anwendungsdomänen und Ausführungsumgbungen spezifiziert. Ein iterativer Adaptionsprozess ermöglicht die schrittweise Integration von neuen Konzepten. Für die Projektion von Bochica-Modellen auf eine Agentenplattform werden entsprechende Abbildungsregeln spezifiziert. Um das Bochica Rahmenwerk für die Modellierung von Agenten in semantisch annotierten virtuellen Welten anzupassen wird eine entsprechendes Erweiterung eingeführt. Abschließend wird ein modellgetriebener Ansatz für die Extraktion des zugrundeliegenden Designs eines bereits implementierten Agentensystems auf die platformunabhängige Ebene vorgestellt. Bochica wurde in zwei Fallstudien für die Modellierung von Agenten in einer virtuelle Fabrikumgebung und die Extraktion des Designs eines bereits implementierten Agentensystems evaluiert. Die Evaluierungsergebnisse zeigen, daß das Rahmenwerk die Lücke zwischen einem plattformunabhängigen agentenorientiertem Design und der Zielplattform effektiv verringert.

Danksagung

An erster Stelle möchte ich Herrn Prof. Dr. Philipp Slusallek für die Möglichkeit in einer sehr erfolgreichen Forschungsgruppe arbeiten zu können danken. Das kreative Umfeld und die Arbeit mit Kollegen aus unterschiedlichen Fachrichtungen hatten großen Einfluss auf meine Arbeit. Aus der Multiagenten-Forschungsgruppe möchte ich zuerst Herrn Dr. Klaus Fischer und PD Dr. Matthias Klusch für die vielen interessante Diskussionen und Anmerkungen danken, welche mich bei meiner wissenschaftlichen Arbeit weiter gebracht haben. Ein ganz besonderer Dank geht auch an meine ehemaligen Zimmergenossen und Kollegen Christian Hahn, Patrick Kapahnke, Esteban León-Soto, Cristián Mardrigal-Mora, Stefan Nesbigall, und Ingo Zinnikus. Ihre Erfahrung und Ratschläge haben mir auf meinem Weg sehr geholfen. Herrn Prof. Dr. Jörg Müller danke für die hilfreichen Kommentare und die Übernahme des Zweitgutachtens.

Ganz herzlich möchte ich mich bei meinen Eltern und meiner Familie bedanken, die mich auf meinem Weg begleitet haben und die immer hinter meinen Entscheidungen gestanden haben. Ihre Unterstützung hat es ermöglicht, dass ich mich dem Studium in dieser Form widmen konnte. Ebenso bedanke ich mich bei meinen Freunden, die für den nötigen Ausgleich gesorgt haben.

Abschließend bedanke ich mich bei Frau Dr. Michelle Carnell und der Graduiertenschule für Informatik der Universität des Saarlandes für das Endstipendium, welches mir die Fertigstellung dieser Arbeit enorm erleichtert hat. Vielen Dank an alle!

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

The research field of *Agent-Oriented Software Engineering* (AOSE) is concerned with investigating how concepts, algorithms, and methods developed in the wide area of *Artificial Intelligence* (AI) can be used for engineering intelligent software agents in a systematic way. AOSE strongly differs from *Object-Oriented Software Engineering* (OOSE) since agents possess an internal architecture which governs the information processing and decision making process. Many different agent architectures have been developed (see Chapter 2). Two prominent examples are the reactive *subsumption* architecture [Brooks, 1986] and the deliberative *Belief, Desire, Intention* (BDI) architecture [Bratman, 1987][Rao and Georgeff, 1991]. The *intelligent agent paradigm* for software engineering [Jennings and Wooldridge, 2000] promises to leverage the development of goal-driven and failure tolerant software as required by today's and future *Information Technology* (IT) systems. AOSE is still a young research area. In 2000, the first annual *Workshop on Agent-oriented Software Engineering* [Ciancarini and Wooldridge, 2001] was held and in 2007 the first issue of the *International Journal of Agent-Oriented Software Engineering* (IJAOSE) [IJA, 2007] was published. Since then, many different agent-oriented methods, methodologies, and modeling languages have been proposed [Weiß and Jakob, 2004][Henderson-Sellers and Giorgini, 2005][Sterling and Taveter, 2009]. Although significant research effort has already been made, AOSE has still not arrived in mainstream software development. The *acceptance problem* of agent technology is being discussed within the agent community [Jennings and Wooldridge, 2000][Belecheanu et al., 2006][McKean et al., 2008]. Four of the identified main problems are (i) misunderstandings or wrong assumptions by non-agent experts, (ii) disappointing experiences of the past, (iii) insufficient agent-oriented standards for industry needs, (iv) lack of powerful methods and tools. Further consolidations accompanied by the development of industry strength methods, development processes, and tools are required to make agent technology accessible for mainstream software development.

AOSE should not be seen in an isolated manner: as it is increasingly applied in main stream software engineering, AOSE is confronted (of course) with typical problems of today's software development such as (i) an increasing number of software frameworks, software languages, and execution platforms, (ii) shorter development cycles, and (iii) heterogeneous and distributed IT environments. In order to leverage AOSE, agent-oriented methods and tools have to cope with those problems. A key to tackling the rapidly growing complexity in software development is abstraction. High-level software languages are required to hide the complexity and focus on the design of IT systems. *Model-Driven Software Development* (MDSD) is driven by industry needs to deal with complex software systems. Several aspects of MDSD have been standardized by the *Object Management Group* (OMG) as *Model-Driven Architecture* (MDA) [OMG, 2003]. The underlying idea of MDA is to model the *System Under Considerations* (SUC) on different levels of abstraction and use model transformations to gradually refine them from *Computational-Independent Models* (CIM) to *Platform-Independent Models* (PIM), and finally to *Platform-Specific Models* (PSM) and executable code. According to Kleppe [2008], the level of abstraction of a software language can be defined as the distance between the computer hardware and the concepts of that language. Since the invention of computer systems, the level of abstraction has steadily been increased from opcodes, assembler languages, procedural languages, up to object-oriented languages. The question that arises is what the next generation of software languages will look like. It has been recognized by Belecheanu et al. [2006], that one benefit of the intelligent agent paradigm is intuitive terminology which comes much closer to how humans perceive a problem domain than the terminology of other paradigms. From this point of view, agent technology is a very promising candidate for next generation software systems.

1.1 Problem Statement and Research Questions

Model-driven development of real world agent-based systems is a complex endeavour that demands expressive and well designed modeling languages. In order to efficiently model a SUC, an ideal agent-oriented modeling language would have to be tailored to a certain application domain (e.g. agent-based simulation or virtual worlds), to the target execution environment (e.g. a proprietary virtual reality or simulation platform), as well as to the development process (e.g. modeling support for AOSE methods such as goal-oriented requirements engineering). At the same time it is desirable that the language abstracts from low-level technical details.

The following statement expresses my language-driven view on AOSE:

> *Intelligent agent technology provides the means to embody high-level concepts like goals, roles, and organizational structures which are required by next generation software languages. The combination of experiences which led to the development of MDA and the conceptual foundation of AOSE has the potential to have a major influence on how IT systems will be designed in future mainstream software engineering.*

In the same line of argumentation, I see agent technology as an enabler to address the requirements expressed by [Kleppe, 2008, p. 8] in the context of traditional MDSD: "*The challenge for language engineers is that the software languages that we need to create must be at a higher level of abstraction, a level we are not yet familiar with. They must be well designed, and software developers must be able to intermix the use of multiple languages. Here too, we are facing a new, unknown, and uncertain task.*" The majority of today's agent-oriented modeling approaches have been created to either support (i) an agent-oriented methodology or (ii) a certain agent execution platform. One problem in model-driven AOSE is that most modeling languages only focus on the core concepts of MAS and often fail to address concepts of custom application domains and execution environments. Thus, certain design decisions cannot be captured. As the design is projected to concrete code, this causes extensive manual refinements to close the gap. The manual refinements potentially cause code and design to diverge over time and thus, undermine a model-driven approach. Agent-oriented modeling languages require more flexibility to address those problems. The transition of agent technology from research to main stream software development will only be successful with industry strength methods and tools. In this regard, the following research questions are addressed by this dissertation:

What are the core concepts of an expressive agent-oriented modeling language?

When today's software developers think about *object-orientation*, there is a common understanding of basic concepts like classes, objects, attributes, or inheritance. The *Unified Modeling Language* (UML) [OMG, 2011d] is a standardized and broadly accepted modeling language that specifies concepts common to the majority of object-oriented systems. This level of maturity has not been reached for the intelligent agent paradigm. Besides the conceptual unification and standardization, the expressiveness of modeling languages is essential for their acceptance.

In order to classify the expressiveness of modeling languages, Warmer and Kleppe [2003] defined six *Modeling Maturity Levels* (MMLs). MML 0 means that there is not even a textual specification of the SUC, and MML 5 stands for *models only*. In the context of MDA, current modeling languages aim at MML 4 (*precise models*). In my opinion, most of today's agent-oriented modeling languages only reach MML 3 (*models and text*). In order to close the gap between design and code, agent model artifacts have to be better interleaved (e.g. behaviors, interaction protocols, and organizational structures). Moreover, the question that arises is how 3rd-party software languages (e.g. reasoning and knowledge representation languages) and often neglected concepts like knowledge bases and data and information models can be integrated.

How can agent-oriented modeling languages better support concrete application domains and execution environments?

Abstract concepts such as *agent*, *behavior*, or *interaction* are common to most agent-oriented systems. Depending on the target platform, the application domain, and the overall approach, those concepts might be realized in different ways in order to address custom features. As a modeling language is applied to different scenarios, the demand for adding more and more highly specialized concepts, only relevant for a small sub-set of agent systems, arises. This bears the danger of making the modeling language unusable over time. Today's agent-oriented modeling languages have a static nature and focus on the core concepts of MAS. Platform-specific languages are created for exactly one execution platform while platform-independent ones lack concepts for addressing features of a legacy target platform or application domain. The question of how the benefits of platform-independent and platform and domain-specific approaches can be joined arises.

How to effectively close the gap between platform-independent agent models and concrete code?

Model transformations are used in MDA to close the gap between high-level design artifacts and concrete code. One common problem in model-driven AOSE is that models created by agent methodologies are often implemented manually (at least partially) and thus, code and design diverge over time. Moreover, in real world applications, agent platforms are usually embedded into larger execution environments with additional IT systems. The execution environment depends on the concrete use case. The conceptual mappings have to be tailored to the execution environment in order to exploit platform features and to minimize the need for manual changes. As the number of execution environments an agent platform can be embedded in is arbitrary, one problem is the high development and maintenance cost for having custom transformations for all the target environments the platform is embedded in. The question that arises is how to make the transformation process more flexible in order to share common conceptual mappings across execution environments.

How to make the underlying design of concrete implemented multiagent systems reusable?

Software engineers usually approach complex problems by separating them into sub-problems. In AOSE, sub-problems are solved by autonomous agents that build organizational structures. Each agent internally further decomposes a problem by goal and plan hierarchies. The underlying design of a system reflects an engineers experience with approaching complex problems. It is desirable to reuse patterns and structures that have proven their practical use and have been validated (e.g. interaction protocols, goal hierarchies, behavior templates, and organizational structures). As of today, only little work has been spent on reverse engineering in the context of agent-based systems. A method for agent-oriented reverse engineering is required (i) to extract and reuse design patterns of already implemented MAS, (ii) as a starting point for refactoring (restructuring) systems (e.g. to migrate to MDSD), and (iii) for analyzing and visualizing existing systems. Thus, reverse engineering is an important method to prepare AOSE for main stream software engineering.

1.2 Contributions

In this dissertation, I will propose a novel model-driven framework for AOSE. The framework, called BOCHICA[1], goes beyond the state-of-the-art AOSE research as it combines the advantages of platform-independent and platform-specific modeling approaches. This has been reached by providing conceptual interfaces for integrating 3rd-party software languages (e.g. knowledge representation or reasoning languages) and concepts into the platform-independent core. Additionally, I propose an iterative adaptation process for incorporating conceptual extensions during a typical software development process. The framework is accompanied by a flexible forward transformation architecture for BDI agents which is tailored

[1]Bochica was a semi-god of the Muisca culture, who brought the people living skills and showed them how to organize their lives. The Muiscas are known for the ceremony of “El Dorado”.

to the needs of Bochica. Reusability of design patterns and model artifacts is enabled by the platform-independent core of Bochica and supported by a novel method for model-driven reverse engineering of BDI agents. Finally, an extension model for agents in semantically-enriched virtual worlds is proposed as an extension of Bochica. The overall framework has been evaluated in a virtual production line and a retail case study. The contributions of this dissertation to the state-of-the-art in AOSE research are summarized as follows:

Model-driven AOSE Framework. The Bochica framework provides the means for capturing platform-independent MAS designs and mapping them to concrete code. The extension interfaces provided by Bochica significantly widen the scope of application domains, execution environments, and development processes in which Bochica can be applied. Instead of defining yet another agent-oriented methodology for Bochica, the framework is aligned to existing agent methodologies and software development processes. Moreover, the research focus lies on increasing the MML of the *Domain-Specific Language* (DSL) underlying the Bochica framework to better bridge the gap between high-level model artifacts and concrete code. The Bochica framework was initially introduced in [Warwas, 2012] and [Warwas et al., 2012a]. The framework paper was nominated for the *Best Student Paper Award* at the *4th International Conference on Agents and Artificial Intelligence* (ICAART'12). The adaptation process was also discussed in [Fischer and Warwas, 2012]. The concrete syntax and static semantics of the core DSL was discussed in [Warwas and Hahn, 2008] and [Warwas and Hahn, 2009b]. The development environment which implements the Bochica framework was presented in [Warwas and Hahn, 2009a] and [Warwas et al., 2012b]. It was awarded with the *Best Academic Software Award* at the *8th International Conference on Autonomous Agents and Multiagent Systems* (AAMAS'09).

Forward Engineering. As Bochica gets extended with 3rd-party concepts and software languages, a flexible model transformation architecture is required to effectively close the gap to concrete code. For this purpose, I define conceptual mappings between Bochica and the BDI agent platform Jadex. Based on the experiences our research group has gained with model transformations to various agent platforms, I propose a modular and extensible transformation architecture that enables the reuse of conceptual mappings from Bochica to multiple execution environments which are based on the same agent platform. The transformation approach was initially presented in [Warwas et al., 2012a] and [Warwas, 2012]. Some of the design patterns were presented in [Warwas et al., 2009].

Reverse Engineering. This dissertation contributes a novel reverse engineering approach to the state-of-the-art in model-driven AOSE. For this purpose, conceptual mappings between the BDI agent platform Jadex and Bochica are specified. After the code has been lifted to the model level, the conceptual mappings are applied to extract the underlying design of a concrete implemented MAS. The extracted artifacts can be re-used as blue print for solving similar problems on similar execution platforms, visualizing the underlying structure, or for migrating an implemented agent system to model-driven AOSE. In this context, I have also developed a platform-specific metamodel for the Jadex platform which enables the reverse engineering approach. The method was initially discussed in [Warwas and Klusch, 2011].

Extension Model for Agents in Semantically-enriched Virtual Worlds. In the research project *Intelligent Simulated Realities*[2] (ISReal), our research group developed a deployment platform for semantically-enriched virtual worlds [Nesbigall et al., 2010]. Semantic Web-based annotations of geometric objects in the 3D scene are used by intelligent agents for reasoning and planning. I was responsible for the agent-related aspects. In order to evaluate Bochica in a complex real world scenario, I developed a platform model for agents in semantically-enriched virtual worlds. The ISReal-specific extension includes (i) platform and domain-specific concepts for intelligent ISReal agents, (ii) the integration of Semantic Web reasoning and knowledge representation languages, and (iii) additional conceptual mappings to the ISReal-enabled Jadex platform. The approach was first published in [Warwas, 2012] and [Warwas et al., 2012b].

SmartFactory and IRL Case Studies. The *SmartFactory*[3] is a *living lab* of the DFKI in Kaiserslautern (Germany) which is concerned with innovative technologies for the factory of the future. One objective of the ISReal project was the design of intelligent agents which are able to operate a virtual counterpart of the real SmartFactory and simulate typical workflows (e.g. for training employees and testing virtual production lines). In this context, the ISReal-enabled Bochica framework was evaluated to design and implement intelligent agents which are able to flexibly interact with their virtual environment. In a collaboration with the *Semantic Product Memory*[4] (SemProM) project and the DFKI *Innovative Re-*

[2]`http://www-ags.dfki.uni-sb.de/~klusch/isreal/index.html`
[3]`http://smartfactory.dfki.uni-kl.de/en/`
[4]`http://www.semprom.org`

tail Laboratory[5] (IRL) living lab, our research group developed an agent-based system for guiding products through a supermarket environment and updating the according product memories (e.g. surveillance of temperature, location, and use-by date) [Kahl et al., 2011]. Initially, the system was directly implemented in Jadex (as Bochica was not fully functional at that time). The objective of the IRL case study was to evaluate the reverse engineering approach by extracting and lifting the underlying design of the already existing agent system to Bochica.

1.3 Structure

This dissertation is structured as follows:

Chapter 1 introduces and motivates this thesis and provides an overview of the addressed research questions and the contributions (this chapter).

Chapter 2 discusses the related work in MDSD and model-driven AOSE. This includes an overview of the state-of-the-art of agent-oriented methodologies, agent modeling languages, and agent-oriented modeling tool support.

Chapter 3 introduces the Bochica framework for model-driven AOSE. This includes an iterative adaptation process, the alignment of Bochica to typical software development processes, the extension interfaces, and the transformation approach. Finally, a brief overview of the technical infrastructure is provided.

Chapter 4 provides an overview of the Bochica metamodel and introduces a platform-specific metamodel for Jadex.

Chapter 5 presents a forward transformation approach for BDI agents. For this purpose, conceptual mappings from Bochica to the Jadex BDI platform are specified. The mappings build the foundation for the forward transformation as required by Bochica.

Chapter 6 presents a novel model-driven reverse engineering approach for AOSE. For this purpose, conceptual mappings between the Jadex metamodel and Bochica are specified. Finally, the conceptual mismatches are discussed.

[5]`http://www.dfki.de/irl/uk/project.htm`

Chapter 7 introduces an extension model for agents in semantically-enriched virtual worlds. After a brief overview of the ISReal platform, the conceptual extensions of Bochica for ISReal and the integration of Semantic Web languages are discussed. Finally, the conceptual mappings to the ISReal platform are specified.

Chapter 8 evaluates the Bochica framework in the SmartFactory and IRL case studies. The results are discussed and set into context to the related work presented in Chapter 2.

Chapter 9 concludes this thesis and provides an outlook on future work.

Chapter 2

Related Work

Most books about agent technology and AOSE start with a philosophical discussion about the concept of *agent* by comparing it to the concept of *object*. Usually, properties like *autonomy*, *pro-activeness*, *social ability*, and *reactivity* are highlighted [Wooldridge and Jennings, 1995]. Although those discussions are important, they do not contribute much to the understanding of what it actually means to design and implement a SUC using the agent paradigm. It is important to underline the importance of the internal architecture of agents, which governs the information processing and decision making process, to AOSE. An agent architecture might make use of mentalistic notion (e.g. BDI) or not (e.g. subsumption architecture). The used agent architecture has to be appropriate to the problem at hand. Software engineers have to exploit the characteristics of the agent architecture in order to gain benefits from it. When non-agent experts think about agent technology, some of the aspects which immediately come into their minds are learning, game theory, or planning. Although those aspects are definitely in the scope of AOSE, the current research focus is on a much more fundamental level and addresses questions like (i) what are the core concepts of MAS, (ii) how to combine the different aspects into a coherent industry-strength framework, (iii) what is an adequate development process for implementing MAS, or (iv) when to apply AOSE or stick to traditional OOSE. Weyns [2010, p. 1] describes AOSE as follows: "*Developing multi-agent systems software is 95% software engineering and 5% multi-agent systems theory.*" Although the ratio might be too high, it emphasizes the need to consider systematics and methods developed by traditional software engineering research. In the following, Section 2.1 presents the state-of-the-art in MDSD, which has been developed by traditional software engineering to separate software design and code. Afterwards, Section 2.2 summarizes the related work in model-driven AOSE. For an introduction and overview of the foundations of agents and multiagent systems we refer to [Weiß, 1999] and [Wooldridge, 2009].

2.1 Model-driven Software Development

The underlying idea of MDSD is to design a SUC from an abstract and technology-neutral viewpoint and use model transformations for refining the design until concrete code is generated. The *Object Management Group*[6] (OMG) standardized several aspects of MDSD as *Model-Driven Architecture*[7] (MDA) [OMG, 2003]. This includes the definition of different viewpoints such as the computational and platform-independent viewpoints as well as the platform-specific viewpoint. *Computational-Independent Models* (CIM) specify a system without computational considerations in mind. *Platform-Independent Models* describe the system with computational considerations but abstract from low-level technical details. Finally, *Platform-Specific Models* (PSM) specify the technical details for a concrete execution platform. In the context of MDA, several standards for metamodeling (see Section 2.1.1) and model transformations (see Section 2.1.2) have been published. Moreover, auxiliary standards like the *Object Constraint Language* (OCL) [OMG, 2012] for querying and constraining metamodels or the *Software and Systems Process Engineering Metamodel* (SPEM) [OMG, 2008b] for defining software development processes are available. The remainder of this section discusses the role of models (see Section 2.1.1) and model transformations (see Section 2.1.2) in MDA. Finally, Section 2.1.3 provides a brief overview of available modeling frameworks.

2.1.1 Models and Metamodels

According to OMG [2003, p. 2-2], a model "*... of a system is a description or specification of that system and its environment for some certain purpose.*" Metamodels are models that define the abstract syntax of a modeling language in terms of objects and their relations (see Figure 2.1). The vocabulary available for defining metamodels is provided by other metamodels - so called meta-metamodels. Meta-metamodels are recursively defined by themselves. Usually, the graphical notation of UML class diagrams is used to visualize metamodels. Two important meta-metamodels are the *Meta Object Facility*[8] (MOF) [OMG, 2011b] and *Ecore* [Steinberg et al., 2008]. MOF was standardized by OMG and builds the foundation of MDA and UML. Ecore can be seen as a sub-set of MOF which was introduced to simplify MOF (see Section 2.1.3). MDA is about software languages and the conceptual mappings from one to another language.

[6]`http://www.omg.org`
[7]`http://www.omg.org/mda/`
[8]`http://www.omg.org/mof/`

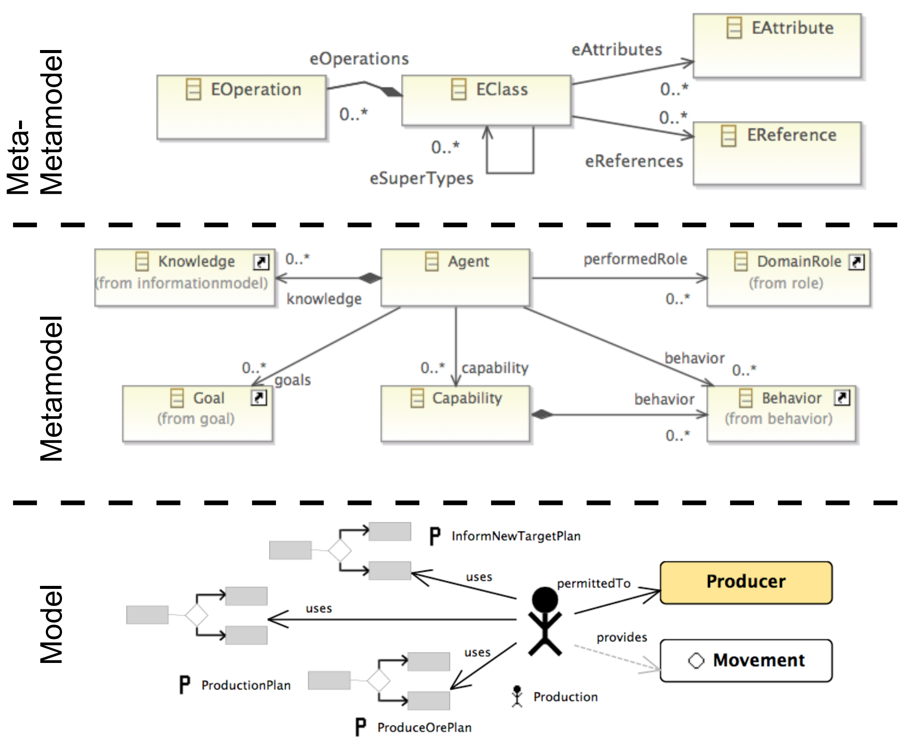


Figure 2.1: The top-most layer depicts the meta-metamodel layer (here Ecore). Metamodels are instances of meta-metamodels and specify the abstract syntax of modeling languages (here an agent-oriented language). Finally, the lowest layer depicts an instance of the metamodel which describes a concrete SUC using the vocabulary defined by the metamodel. The concrete syntax is used to visualize the model elements.

A graphical modeling language consists of:

- *abstract syntax* (grammar): The abstract syntax of a modeling language defines the available concepts and their relations for modeling a SUC. In MDA, this is done by metamodels.
- *concrete syntax* (notation): The notation defines graphical representations for the concepts of the abstract syntax.
- *static semantics*: The static semantics describes constraints that have to be fulfilled by a model in order to be well-formed.
- *dynamic semantics*: The dynamic semantics defines the meaning of well-formed expressions.

According to Kleppe [2008], the semantics of a modeling language can be defined (i) denotational, (ii) by providing a reference implementation, (iii) by mapping the language's concepts to another language with well-defined semantics, or

(iv) by providing the operational semantics (e.g. by a state transition system). For example, UML is a semi-formal graphical modeling language. The abstract syntax of UML is described by a metamodel, called the *UML Superstructure* [OMG, 2011d]. The mapping between the graphical notation of UML and its abstract syntax is expressed using natural language. The semantics of UML is defined by formal OCL constraints and additional natural language remarks.

2.1.2 Model Transformations

Model-to-model transformations are used within MDA for mapping the concepts of one modeling language to the concepts of another one. Model-to-text transformations generate source code for a target platform. Figure 2.2 depicts the model-to-model transformation approach used in MDA. Transformation rules define conceptual mappings from the concepts of the source to the target metamodel. The transformation engine generates the target model by applying the mapping rules to the source model. One can distinguish between *vertical* and *horizontal* transformations. Vertical transformations map a model of one level of abstraction (e.g. PIM) to another level of abstraction (e.g. PSM), whereas horizontal transformations stay on the same level of abstraction. Moreover, it is important to distinguish between following transformation types (taxonomy aligned to [Chikofsky and Cross II, 1990]):

- *Forward Engineering:* transforming the model of a SUC to a lower level of abstraction (e.g. from PIM to PSM)
- *Reverse Engineering:* analyzing a SUC and creating a model on a higher level of abstraction (e.g. PSM to PIM)
- *Reengineering:* implementing a SUC in a different way; e.g. by reverse engineering it and applying a forward transformation
- *Restructuring (refactoring):* improving or changing the structure of a model without modifying the functionality
- *Roundtrip Engineering*: forward and reverse transformations can be done in an arbitrary order (requires an additional synchronization mechanism between the transformations)

In the context of MDA, the *Queries, Views, and Transformations* (QVT) [OMG, 2011c] language has been standardized for model-to-model transformations. Code generation from models has been standardized by *Model to Text* (M2T)

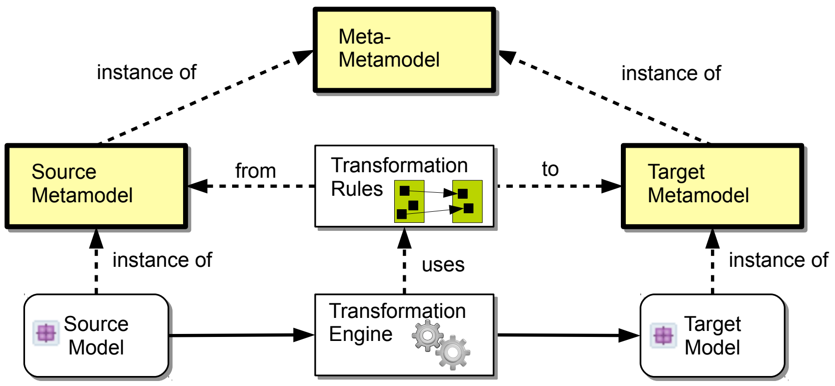


Figure 2.2: In order to transform a source model to a target model, mapping rules between concepts of the source and target metamodels have to be specified. The transformation rules as well as the source model are the input to the transformation engine.

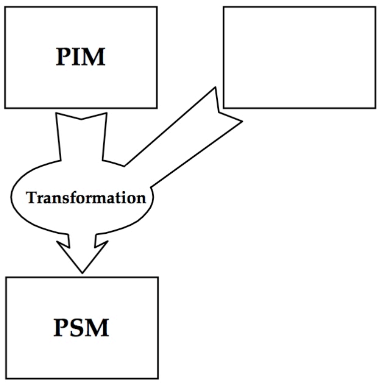


Figure 2.3: Abstract view of PIM to PSM transformations according to [OMG, 2003, p.2-7].

[OMG, 2008a]. Figure 2.3 depicts an abstract and technology-independent view of how models in MDA are mapped from PIM to PSM. The platform-independent model is complemented by a *Platform Description Model* (PDM) which provides additional information about the target platform. Both models are the input to the model transformation which generates the platform-specific model. Usually, there exist multiple ways of how to structure and implement a SUC given a certain target platform. The specification of a model transformation to a certain execution platform implies to anticipate design decisions. In the context of OOSE, object-oriented *design patterns* have been developed [Gamma et al., 1995]. Design patterns provide reusable solutions to well-known and recurrent problems. The application of design patterns during code generation helps to guarantee a certain quality of the generated code. An interesting research question that requires more attention is how design patterns for the needs of AOSE look like.

2.1.3 Modeling Frameworks and Workbenches

Besides MDA, there exist several other DSL frameworks and language workbenches for MDSD. The *Visual Studio Visualization and Modeling SDK*[9] (VSVMSDK) is Microsoft's approach to MDSD. It is available as part of the *Visual Studio*[10] development environment and provides tool support for developing custom DSLs. VSVMSDK does not implement OMG standards. Moreover, it is more pragmatic compared to MDA (e.g. there exists nothing comparable to CIM, PIM, and PSM in MDA). The tool also supports the user in designing the graphical notation. A further modeling framework is based on the *Graph-Object-Property-Role-Relationship* (GOPRR) [Kelly et al., 1996] language. It has its origin in the 90's before MDSD according to MDA was available. The MetaEdit+[11] workbench provides the development environment for specifying custom DSLs. A DSL is defined by concepts, properties, and additional rules. The workbench also provides a design tool for the graphical notation. The *Eclipse Modeling Framework*[12] (EMF) [Steinberg et al., 2008] provides a collection of tools for MDSD based on Eclipse technology. This encompasses implementations of MDA standards such as UML[13], QVT[14], OCL[15], and SPEM[16]. EMF is based on the Ecore metamodel which has the same role for EMF as MOF for MDA. Since Ecore builds the foundation for several aspects of this dissertation, we provide a brief overview of Ecore and EMF.

Eclipse Modeling Framework. Ecore defines object-oriented concepts such as `EClassifier`, `EPackage`, and `EClass` (see Figure 2.4). An `EClass` is an `ENamedElement` and contains `EAttributes`, `EReferences`, as well as `EOperations`. It is important to note that Ecore unifies two different aspects: (i) it is the meta-metamodel for specifying DSLs based on Eclipse technology and (ii) it is a metamodel for Java. EMF provides automatic mappings from Ecore to UML classes, Java, and *XML Schema Definitions* (XSD) [W3C, 2004b] - and vice versa. Graphical DSLs are defined using the infrastructure provided by the *Graphical Modeling Framework*[17] (GMF). Likewise, textual DSLs can be specified using *EMFText*[18] or *XText*[19]. Besides the standardized QVT, there exist other model transformation

[9] `http://archive.msdn.microsoft.com/vsvmsdk/`
[10] `http://www.microsoft.com/visualstudio/`
[11] `http://www.metacase.com/mwb/`
[12] `http://www.eclipse.org/modeling/emf/`
[13] `http://www.eclipse.org/modeling/mdt/?project=uml2`
[14] `http://www.eclipse.org/m2m/`
[15] `http://www.eclipse.org/modeling/mdt/?project=ocl`
[16] `http://www.eclipse.org/epf/`
[17] `http://www.eclipse.org/modeling/gmp/`
[18] `http://www.emftext.org/index.php/EMFText`
[19] `http://www.eclipse.org/Xtext/`

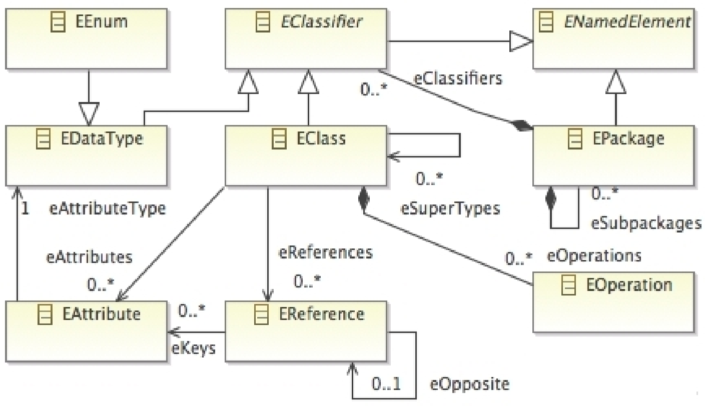


Figure 2.4: This figure depicts the central concepts of the Ecore metamodel.

languages like the declarative *Atlas Transformation Language*[20] (ATL). Model-to-text transformations are supported by *XPand*[21] and the *Java Emitter Templates*[22] (JET). Moreover, metamodel and DSL zoos, which enable the reuse of existing modeling languages, became available during the recent years. Two examples are the *EMFText Concrete Syntax Zoo*[23] and the *Atlantic Metamodel Zoo*[24]. Finally, it is worth mentioning the Teneo[25] and Dawn[26] projects which provide the infrastructure for model repositories and collaborative modeling.

2.2 Model-driven Agent-oriented Software Engineering

MDA is driven by industry needs to separate software design from low-level technical details. Model-driven AOSE transfers experiences and technology from traditional software engineering to the young research field of AOSE. At the same time, AOSE offers a powerful conceptual foundation which goes beyond OOSE. Standardization plays an important role for increasing the acceptance of new technology. The *Foundation for Physical Agents*[27] (FIPA) is a standardization organization for agent technology. Published standards like the *Agent Management Specification* [FIPA, 2004] define the general structure for agent platforms. More-

[20]`http://www.eclipse.org/atl/`
[21]`http://www.eclipse.org/modeling/m2t/?project=xpand`
[22]`http://www.eclipse.org/modeling/m2t/?project=jet`
[23]`http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo`
[24]`http://www.emn.fr/z-info/atlanmod/index.php/Zoos`
[25]`http://wiki.eclipse.org/Teneo`
[26]`http://wiki.eclipse.org/Dawn`
[27]`http://www.fipa.org`

over, several communication-related aspects such as interaction protocols [FIPA, 2003] or an *Agent Communication Language* (ACL) [FIPA, 2002a][FIPA, 2002b] were standardized. The remainder of this section provides an overview of available agent technology (Section 2.2.1), discusses the state-of-the-art of agent-oriented methodologies (Section 2.2.2), modeling languages (Section 2.2.3), and modeling tools (Section 2.2.4).

2.2.1 Agent Architectures and Platforms

Agent platforms provide the environment for implementing and executing MAS. This also encompasses the internal architecture of agents which has to be addressed by the developer. Many different agent architectures have been proposed. One of the most influential ones is the *Procedural Reasoning System* (PRS) [Georgeff and Ingrand, 1989] architecture which belongs to the family of BDI architectures. BDI agents have an explicit representation of beliefs and goals. Goals represent a desired goal state (*what* should be achieved), whereas plans determine *how* an adopted goal is reached. *Deliberation* is the process of deciding which goal to pursue. The process of deciding how to achieve a goal is called *means-end reasoning*. In PRS systems, the means-end reasoning process is realized by a look-up mechanism (also called planning from *second principles*). For this purpose, a PRS agent is equipped with a plan library with predefined behavior templates (means). The templates specify the agent's behavior in certain situations - similar to *methods* in (offline) *Hierarchical Task Network* (HTN) planning [Nau et al., 2004]. Plan templates related to the currently active goals (ends) are being looked-up and instantiated. In PRS, *metalevel reasoning* is the process of deciding which option to choose (if there are multiple ones). Another agent architecture is the reactive subsumption architecture [Brooks, 1986]. In contrast to BDI, it works without an explicit representation of the world state. Instead, subsumption agents use prioritized layers of behaviors which are triggered by the agent's perceptions. BDI agent systems are especially interesting for AOSE since they provide execution semantics for high-level concepts like goals. For the development of agents based on other agent architectures, goals are design artifacts which are only implicitly represented at the code level. Two further agent architectures worth mentioning are the cognitive Soar architecture [Milnes et al., 1992][Laird, 2012] and the hybrid *Integration of Reaktive behaviour and Rational Planning* (InteRRaP) [Müller and Pischel, 1994][Müller, 1997] architecture.

There exist various agent platforms for the different agent architectures. The *Java Agent Development Framework*[28] (Jade) [Bellifemine et al., 1999] is a popular

[28]`http://jade.tilab.com`

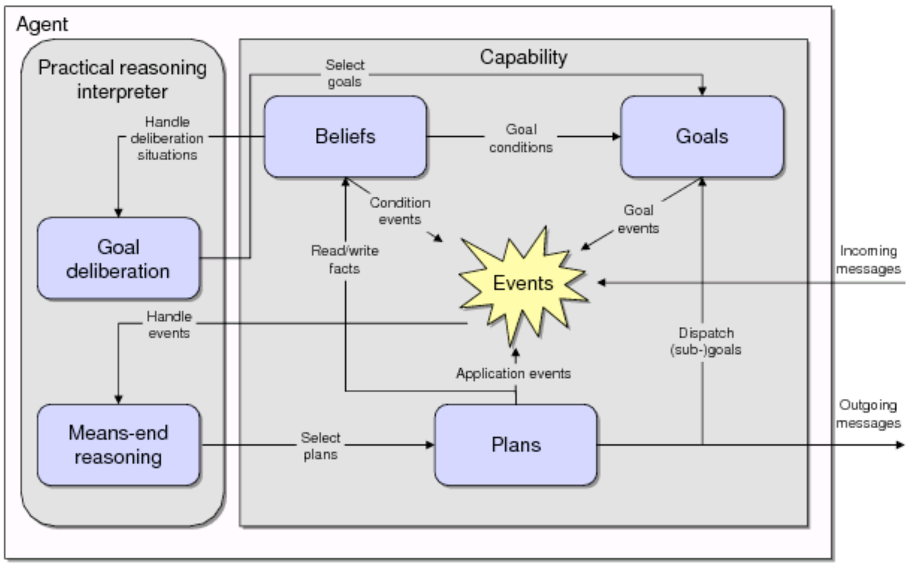


Figure 2.5: This figure depicts the Jadex BDI architecture (taken from[33]). The practical reasoning interpreter implements an agent's goal deliberation and means-end reasoning process.

and mature FIPA-based agent platform. Jade can be seen as a Java-based toolbox which provides the infrastructure for agent-based systems (e.g. a FIPA-based communication service, a distributed execution platform, agent management tools, and support for mobile devices). Jack[29] [Busetta et al., 1999] is a commercial BDI agent platform. The Jack language extends Java with agent-oriented concepts (e.g. for posting goals and declaring agents). Jack also supports organizational structures, called *teams*. Other agent platforms focus on the needs of certain application domains. For example, *XaitControl*[30] is an agent platform for computer games. The behavior of XaitControl agents is defined by finite state machines and the focus is clearly on performance – less on features. According to Dikenelli et al. [2005] and Dikenelli [2008], *SEAGENT*[31] is an agent platform based on Semantic Web technology. Agents are equipped with knowledge bases based on the *Ontology Web Language* (OWL) [W3C, 2009] and support the invocation of semantic service descriptions based on OWL-S[32]. SEAGENT plans are defined in HTN-style.

Several chapters of this dissertation consider the agent platform Jadex[34] [Pokahr et al., 2005][Pokahr and Braubach, 2009] as target platform. Figure 2.5 depicts the Jadex BDI architecture. The Jadex platform provides typical services such

[29]`http://www.agent-software.com/products/jack/index.html`
[30]`http://www.xaitment.com/english/products/xaitcontrol/xaitcontrol.html`
[31]`http://seagent.ege.edu.tr`
[32]`http://www.ai.sri.com/daml/services/owl-s/`
[33]`http://jadex-agents.informatik.uni-hamburg.de`
[34]`http://vsis-www.informatik.uni-hamburg.de/projects/jadex/`

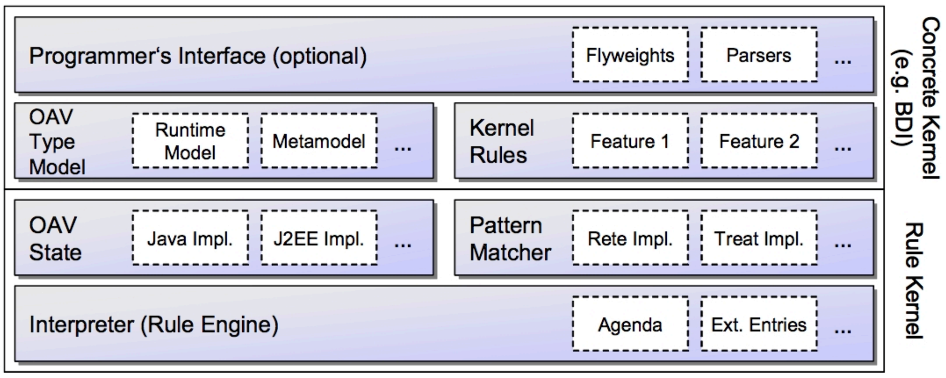


Figure 2.6: Overview of the Jadex kernel infrastructure (taken from [Pokahr and Braubach, 2009, p. 776]).

as a directory service, a FIPA-based messaging service, and tools for managing the platform. Jadex agents are configured by XML files and behaviors are implemented as Java-based plans. The XML files make use of XML Schema definitions which encompass the configuration of applications, agents, capabilities, and the environment. Jadex supports four different goal types such as *perform goal* (perform some action), *achieve goal* (achieve a certain goal state), *maintain goal* (maintain a certain state), and *query goal* (provide an answer). There exist no specialized concepts for organizational structures nor an explicit representation of interaction protocols. Figure 2.6 depicts the technical infrastructure of the Jadex rule kernel (Jadex version 2). The bottom layer depicts the generic rule engine Jadex is built on. The current state of the system is maintained in an *Object Attribute Value* (OAV) representation. The initial state of an agent is initialized by XML files. Jadex can be extended with other agent architectures, too. This is achieved by providing a so called *kernel module* for the considered agent architecture. The Jadex BDI kernel is one example kernel module. The top-most layer of Figure 2.6 depicts the agent engineer's interface to the underlying layers.

2.2.2 Agent-oriented Methodologies

Several software development processes like the classical *waterfall model* [Royce, 1987] and the iterative *spiral model* [Boehm, 1986] originated from traditional software engineering. During the recent years, iterative and agile development processes gained more and more attention by software developers. For example, the *Rational Unified Process* (RUP) [Kruchten, 2003] is a widely accepted iterative development process and provides a customizable framework for configuring the development process (see Figure 2.7). RUP uses UML for capturing the design decisions. It has been widely recognized within the agent community that

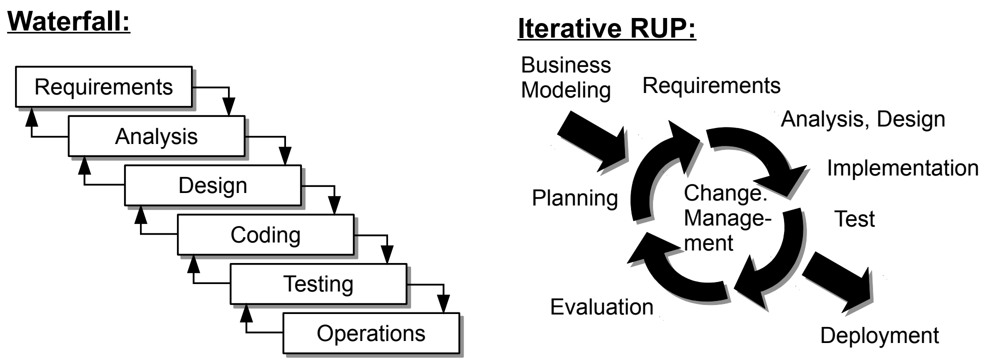


Figure 2.7: This figure depicts the waterfall model according to Royce [1987] and the iterative RUP according to Kroll and Krutchten [2003, p. 7].

the existing software engineering methodologies do not satisfy the needs of AOSE (e.g. [Zambonelli et al., 2003], [Padgham and Winikoff, 2004, p. 22]). During the recent years, various agent-oriented methodologies have been proposed. The *FIPA Methodology Technical Committee* [35] and the *FIPA Working Group: Design Process Documentation and Fragmentation*[36] are two initiatives for the unification and standardization of agent-oriented methodologies. As of today, there exists no standardized agent-oriented approach and the methodologies are still driven by research. The framework presented by this dissertation is related to agent methodologies as it provides the means for capturing design decisions and bridging the gap between high-level designs and executable code. For this purpose, we provide a brief overview of the relevant approaches. The methodologies were selected due to their influence in the community and the relevance to our approach.

Gaia. According to Wooldridge et al. [2000] and Zambonelli et al. [2003], Gaia is an agent-oriented methodology which follows a sequential development process. Gaia covers the agent-oriented analysis and design phases. The design artifacts are kept abstract and leave many aspects open (e.g. concrete interaction protocols or behavior patterns are not defined). Gaia highlights the role of organizational structures and the environment. During the analysis phase, organizational structures, interactions, and an environment model are defined. The architectural and detailed design phases further refine the models by adding agent and service models. Moraïtis et al. [2003] and Moraïtis and Spanoudakis [2004] discussed how to implement a personal (travel) assistant manually using the Gaia methodology and Jade.

INGENIAS. According to Gómez-Sanz [2002], Pavón and Gómez-Sanz [2003], and Pavón et al. [2005], INGENIAS is an agent-oriented methodology which sup-

[35]`http://www.fipa.org/activities/methodology.html`
[36]`http://www.pa.icar.cnr.it/cossentino/fipa-dpdf-wg/`

ports the development of agents with a mental model. INGENIAS originated from the MESSAGE [Garijo et al.] methodology and is aligned to RUP. Much research effort has been spent on detailed design and implementation. In [Gómez-Sanz et al., 2009], testing and debugging of interaction protocols in INGENIAS was discussed. The INGENIAS approach has been evaluated for scenarios like a surveillance system [Gascueña and Fernández-Caballero, 2007] and crisis management [García-Magariño et al., 2009]. In order to unify the benefits of INGENIAS with other approaches, the combination with Tropos [Fuentes-Fernández et al., 2006] and Prometheus [Gascueña and Fernández-Caballero, 2009][Fernández-Caballero and Gascueña, 2010] was discussed.

O-MaSE. According to DeLoach [2006], García-Ojeda et al. [2008] and DeLoach and García-Ojeda [2010], the *Organization-based Multiagent System Engineering* (O-MaSE) methodology originated from the MaSE [DeLoach and Wood, 2001] methodology and has an organizational view on AOSE. For example, it supports *policies* for constraining the behavior of a system. The O-MaSE methodology does not define a fixed development processes. Instead, O-MaSE provides a framework for combining different *method fragments* for the requirements, analysis, and design phases. *Method construction guidelines* support this process. O-MaSE does not prescribe a certain agent architecture but was used to engineer BDI agents. The process framework was initially based on the *Open Process Framework* (OPF) [Firesmith and Henderson-Sellers, 2001] and was migrated to SPEM. According to DeLoach and García-Ojeda [2010], O-MaSE has been evaluated in sequential and iterative development processes.

Prometheus. According to Padgham and Winikoff [2004], Prometheus[37] is a methodology for developing BDI agent systems. It covers the three development phases (i) system specification, (ii) architectural design, and (iii) detailed design (see Figure 2.8). Testing and debugging has been discussed by Padgham et al. [2005b], Zhang et al. [2007], and Zhang et al. [2008]. Although Prometheus is not limited to a certain agent execution platform, the design artifacts of the detailed design phase are inspired by the Jack platform. During the system specification phase, system goals, typical processes of a system (called *scenarios*), and perceptions and actions are collected. Similar goals, perceptions, and actions are grouped to *functionalities*. The architectural and detailed design phases are concerned with identifying agent and capability types by grouping functionalities and specifying interaction protocols using AUML. The integration of AUML for specifying interaction protocols has been discussed by Winikoff [2007] and Padgham et al. [2007]. Prometheus was applied to scenarios such as an online book store [Padgham and

[37]`http://www.cs.rmit.edu.au/agents/SAC2/methodology.html`

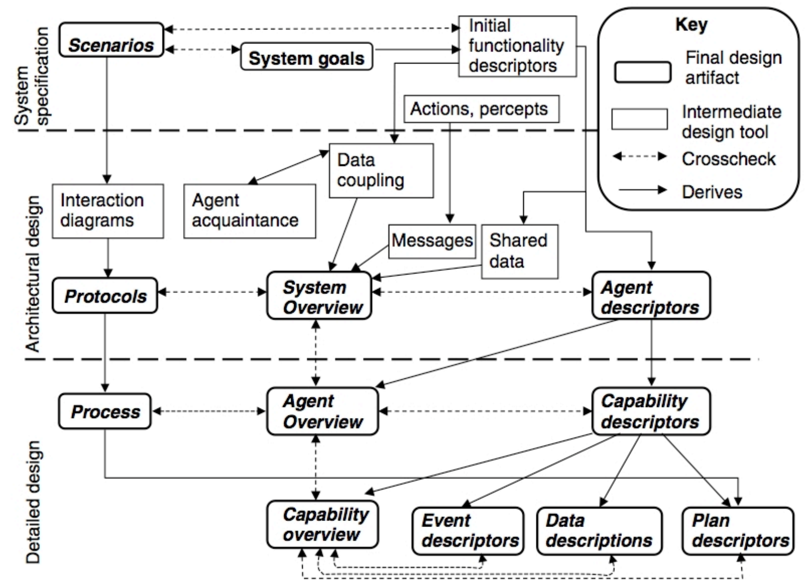


Figure 2.8: This figure depicts an overview of the Prometheus methodology according to [Padgham and Winikoff, 2004, p. 24].

Winikoff, 2004] and a conference management scenario [Padgham et al., 2008].

Tropos. According to Giunchiglia et al. [2002], Bresciani et al. [2004] and Penserini et al. [2006], Tropos[38] is an agent-oriented methodology dedicated to BDI agents. The Tropos methodology covers the development phases starting form early and late requirements until architectural and detailed design. However, the research focus was clearly on the early phases (e.g. [Giorgini et al., 2005]). In Tropos, models undergo an incremental step-wise refinement. The development process starts with identifying actors (stakeholders), the system's goals and their dependencies. Goals are further decomposed into sub-goals and means-end-analysis is used for identifying plans and resources (means) necessary for achieving a goal (end). The architectural design phase is concerned with identifying sub-actors and specifying information and control flows. The detailed design phase uses UML activity diagrams for defining the behavior of agents and AUML sequence diagrams for the interaction between agents. It has been discussed how Tropos can be used for developing Jack [Bresciani et al., 2004], Jade [Penserini et al., 2006], and Jadex [Morandini et al., 2008] agent systems. There exists also an extension of Tropos for adaptive systems [Morandini, 2011].

Table 2.1 summarizes important properties of the selected agent-oriented methodologies. There exists a number of other approaches like PASSI [Cossentino and Potts, 2002], MaSE [DeLoach and Wood, 2001], MASSIVE [Lind, 2001], or MESSAGE [Caire et al.] which are not discussed by this dissertation. Weiß and Jakob

[38]`http://troposproject.org`

	Process	Agent Architecture	Phases
Gaia	sequential	not specific	analysis, architectural and detailed design
INGENIAS	RUP	BDI	analysis, design, implementation
O-MaSE	not predefined	not specific/ BDI	analysis, design, implementation
Prometheus	iterative	BDI	system specification, architectural and detailed design, implementation
Tropos	sequential refinement	BDI	early and late requirements, architectural and detailed design, implementation

Table 2.1: This table summarizes the introduced methodologies.

[2004], Henderson-Sellers and Giorgini [2005], and Sterling and Taveter [2009] provide a good overview of the state-of-the-art. It can be stated that the different approaches have two commonalities: (i) most of them are dedicated to BDI agents and (ii) the majority proposes to follow an iterative development process. The methodologies differ in their foci – e.g. on early are late phases. Currently, the agent-oriented methodologies are undergoing a consolidation phase. There are several initiatives for unifying the different approaches using process languages like SPEM. We expect that this phase will continue for some more time. The next section provides an overview of agent-oriented modeling languages.

2.2.3 Agent-oriented Modeling Languages

Expressive agent-oriented modeling languages are required to capture the design decisions in formal models which can be used for MDSD. In order to classify the expressiveness of modeling languages, Warmer and Kleppe [2003] defined six *Modeling Maturity Levels* (MMLs): (0) *no specification*, (1) *textual specification*, (2) *text and models*, (3) *models with text*, (4) *precise models*, and (5) *models only.* As already mentioned in the introduction, current agent-oriented modeling languages are mostly on MML three (e.g. Prometheus behavior models are specified with text). Only certain aspects reach MML four. There exist several approaches to unify the diverse field of agent-oriented modeling languages. The *FIPA Modeling Technical Committee*[39] is an initiative for developing an agent-oriented metamodel similar to UML. In this context, the *Agent UML*[40] (AUML) approach was founded.

[39]`http://www.fipa.org/activities/modeling.html`
[40]`http://www.auml.org`

As of today, both initiatives had only limited success. The most important outcome was an agent-specific variant of UML sequence diagrams for specifying agent interactions. In the context of OMG, the *Agent Metamodel and Profile*[41] (AMP) approach aimed on extending UML with agent concepts. Finally, approaches like the *FAME Agent-oriented Modeling Language* (FAML) [Beydoun et al., 2009] try to unify existing agent metamodels. The same has been done for graphical notation (e.g. [Padgham et al., 2009]). However, the success of those unification approaches is limited. Before we provide an overview of agent-oriented modeling languages, we introduce following four categories:

- **Platform-independent vs. Platform-specific:** This category directly corresponds to the definition of the platform-independent and platform-specific layers of MDA. Platform-independent languages abstract from concrete technologies or implementation details. This enables the engineer to focus on the overall design of the SUC without technical details. Platform-specific languages are able to exploit the low-level features of the target platform.

- **Agent Architecture-independent vs. Agent Architecture-specific:** The agent architecture specifies the internal information processing and decision making process of an agent. Every agent architecture needs different model artifacts. For example, the PRS architecture requires concepts for supporting *metalevel reasoning.* Of course, the possibility to address agent architecture-specific properties is essential to exploit agent technology.

- **Application Domain-independent vs. Application Domain-specific:** All agent-oriented modeling languages are domain-specific in the sense that they address the domain of MAS. However, we additionally distinguish whether an agent-oriented DSL is specific for a certain application domain such as *Agent-Based Simulation* (ABS), virtual worlds, or eBusiness. Application domain-specific languages allow the engineer to better address concepts of the considered application domain.

- **Methodology-independent vs. Methodology-specific:** Finally, we distinguish whether an agent-oriented modeling language offers concepts for supporting a certain methodology. For example, a method for goal-oriented requirements engineering (as the one of Tropos) requires specialized concepts.

[41] `http://www.omgwiki.org/AMP-team/doku.php`

Platform-independent Agent Modeling Languages

In the following, we separate the modeling approaches into platform-independent (this section) and platform-specific ones (succeeding section). Most agent-oriented platform-independent modeling languages have been created in order to support a certain agent methodology. As most methodology languages do not have an own name, we will refer to them with *<methodology-name>-ML*.

DSML4MAS. Version 1.0 of the *Domain-Specific Modeling Language for MultiAgent Systems* (Dsml4Mas) has been presented by Hahn et al. [2009a]. The abstract syntax of Dsml4Mas is based on the *Platform-Independent Metamodel for Agents* (Pim4Agents). Dsml4Mas was designed to capture the core concepts of MAS on a platform, methodology, and application domain-independent level of abstraction. The strength of the language is the tight integration of agents, organizational structures, interaction protocols, and behaviors. Dsml4Mas 1.0 does not cover goals, knowledge bases, and environment interactions. Dsml4Mas has been further developed as part of this dissertation. Version 2.0 provides interfaces for specializing the language for application domains, execution environments, and methodologies (see Chapter 3). Moreover, goals were introduced and the overall expressiveness was increased. Dsml4Mas has been realized with Ecore and GMF.

INGENIAS-ML. Compared to other methodology-oriented modeling languages, INGENIAS-ML offers many specialized and fine-grained concepts. The original INGENIAS modeling language is based on GOPRR [Kelly et al., 1996] but there also exists a mapping to Ecore [García-Magariño et al., 2007]. INGENIAS-ML provides concepts for modeling agents, goals, interactions, organizational structures, environments, and simple behaviors. Use cases can be modeled to analyze a SUC as part of the INGENIAS methodology. Moreover, it is possible to specify deployment configuration.

O-MaSE-ML. The modeling language supporting the O-MaSE methodology covers concepts like agents, goals (including detailed dependencies), organizational structures, UML-style interaction protocols, capabilities, services, behaviors (including behavior bodies), and domain objects. One speciality is the possibility to specify policies to constrain the system's behavior. O-MaSE-ML is well structured but is not based on Ecore. Instead, it uses an XML-based approach.

Prometheus-ML. Prometheus-ML is a well structured methodology language. It is not based on Ecore and uses an XML-based approach. Prometheus-ML covers concepts like goals, interaction protocols, and actions. Only the header information of behaviors can be modeled (e.g. triggering events). The concepts of the detailed design phase are aligned to Jack. The language supports high-level model artifacts like scenarios and functionalities which are required by the Prometheus methodology. Prometheus-ML integrates the textual AUML approach presented by [Winikoff, 2007] and [Padgham et al., 2007] for specifying agent interactions.

Tropos-ML. Tropos-ML supports the Tropos methodology and is based on Ecore [Susi et al., 2005]. It covers the concepts of the requirements and architectural design phases. This includes concepts like actor, hard and soft goals, plans (no plan bodies), and resources. The language supports means-end analysis and the specification of dependencies and contributions between goals and plans. One speciality of the Tropos language is the separation into a formal and an informal part. According to Fuxman et al. [2001], the *Formal Tropos* language is used for the verification of requirements. An extension for adaptive systems (TAOM4AS) was introduced in [Morandini et al., 2009] and [Morandini, 2011].

Platform-specific Agent Modeling Languages

Platform-specific modeling approaches are usually driven by concrete needs of developers (e.g. to get a graphical overview of interdependencies between platform artifacts). Moreover, they are usually more pragmatic and are able to address low-level features of an execution platform. Several agent-oriented approaches follow a mixed approach: some aspects are modelled graphically, whereas others are still coded. In the following, we focus on those aspects which abstract from source code.

Jack. The *Jack Development Environment*[42] (JDE) provides a proprietary integrated graphical *design tool* [AOS, 2011a] and *plan editor* [AOS, 2011b]. The design tool provides so called *design views* which abstract from the underlying code. The modeling support encompasses agents, events, goals, single messages (no complete interaction protocols), and teams. The behavior of agents can be visualized similar to today's workflow languages and are annotated with Java code. The view elements are directly linked to artifacts of the source code. JDE allows graphical tracing of executing plans. It does not support high-level design artifacts like those of Prometheus.

Jadex BPMN. According to Braubach et al. [2010], Pokahr et al. [2010], and Jander and Lamersdorf [2011], the Jadex platform has been extended with a kernel module for the execution of workflows based on the *Business Process Modeling Notation* (BPMN) [OMG, 2011a]. BPMN models are created with the modeling tool of the Eclipse *SOA Tools Platform*[43] (STP) project. The business processes are annotated with Java expressions – e.g. to express conditions. Moreover, the

[42]`http://aosgrp.com/products/jack/index.html`
[43]`http://www.eclipse.org/stp/`

Goal Process Modeling Notation (GPMN) was developed to support goal-oriented workflows. GPMN supports goal-oriented decomposition of the system. Plans to achieve the goals can be modeled in BPMN. GPMN models are executed by Jadex BDI agents.

Jadex DE. Kardas et al. [2009] presented a platform-specific modeling approach for Jadex. The language covers the concepts of the Jadex BDI XSD files which are used within the Jadex platform to configure BDI agents. The contribution consists of graphical modeling support of Jadex BDI XML files. However, the approach does not allow the graphical specification of plan bodies (only references to Java class files). Moreover, expressions (e.g. preconditions) are plain strings. The model can be used to generate the according Jadex files.

SEAGENT. According to Dikenelli [2008], the *SEAGENT Development Environment* (SDE) provides graphical modeling support for the SEAGENT platform. This includes HTN-based behaviors, interaction protocols, roles, and goals. Since SEAGENT is dedicated to agents based on Semantic Web technology, the modeling language offers specialized concepts for OWL-based knowledge bases and support for the invocation of OWL-S-based service descriptions.

WOLF. According to Caire et al. [2008], the *Workflow and Agent Development Environment*[44] (WADE) is an extension of the agent platform Jade for executing workflows. The concepts of the workflow language are based on the *XML Process Definition Language* (XPDL) [WfMC, 2008] which is a standard of the *Workflow Management Coalition*[45] (WfMC). It encompasses typical workflow concepts like process, activity, and transition. There exists a direct mapping of the workflow elements to Java-based Jade behaviors that implement the workflow. Code activities contain Java code to be executed. Modeling support for WADE is provided by the *WOrkfLow LiFe cycle management environment* (WOLF) plug-in for Eclipse.

Table 2.2 provides an overview of the selected approaches. The depicted results were gained by evaluating the available publications, manuals, and the available software. We created the summary to the best of our knowledge. Regarding the platform-specific approaches we only considered those parts which abstract from source code (e.g. only behaviors in WADE). It is important to note that the number of circles and bullets does not express that one approach is superior to another one. The compared languages are created for different purposes and thus, have different foci. Moreover, the bullets do not express how well the different aspects are interleaved. For example, a language might support interaction protocols, behaviors, and organizational structures but it does not necessarily imply that the different aspects are tightly integrated in order to generate fully executable code.

[44]`http://jade.tilab.com/wade/`

[45]`http://www.wfmc.org`

	Specific for				Concepts						
	M	P	A	D	Goal	Beh.	Prot.	Org.	Env.	KB	Depl.
Dsml4Mas 1.0						●	●	●	○		●
Dsml4Mas 2.0	○	○	○	○	●	●	●	●	○	○	●
INGENIAS-ML	●		●		●	○	●	●	○	○	●
O-MaSE-ML	●		○		●	●	●	●	●		
Prometheus-ML	●	○	●		●	○	●	○	●	○	
Tropos-ML	●		●		●	○			○		
Jack		●	●		●	●	○	●	●	●	
Jadex BPMN		●	●		○	●					
Jadex DE		●	●		●	○	○				○
SEAGENT		●	●	●	●	●	●	○	○	●	
WOLF		●	●			●					

Table 2.2: A bullet means that the aspect is covered by an approach, whereas a circle stands for "to some degree". Following abbreviations are used: (M) methodology-specific, (P) platform-specific, (A) agent architecture-specific, and (D) application domain-specific. Moreover, following concepts were evaluated: goal, behavior (Beh.), protocols (Prot.), organizations (Org.), environment (Env.), knowledge base (KB), and deployment (Depl.).

Those aspects are hard to measure. However, the application to software projects will turn out over time which ones are mature enough for real world projects.

2.2.4 Agent-oriented Modeling Tools

Besides agent-oriented methodologies and expressive modeling languages, model-driven AOSE requires powerful tool support. This includes modeling support during design-time, model transformations for generating executable code, and support for software engineering methodologies. This section provides an overview of modeling tools belonging to the languages and methodologies introduced in the previous sections.

agentTool III. AgentTool III[46] [DeLoach and García-Ojeda, 2010] provides tool support for the O-MaSE methodology based on Eclipse technology. It provides diagrams for modeling agents, capabilities, the domain, goal decompositions, behaviors, etc. Models are based on XML. Code generation is provided for Jade. Additionally, agentTool III supports the computation of different metrics. The *AgentTool Process Editor* (APE) [García-Ojeda et al., 2009] is based on the *Eclipse*

[46]`http://agenttool.cis.ksu.edu`

Process Framework (EPF)[47] and enables engineers to create custom development processes based on O-MaSE method fragments.

DDE. The DSML4MAS development environment has been developed as part of this dissertation. It is based on Eclipse technology (see Section 3.5) and MDA standards. Forward transformations to Jack and Jade for DSML4MAS 1.0 were presented by Hahn et al. [2009a]. An early prototype of a model transformation from DSML4MAS to Jadex was presented in the master thesis of Alexander Trenz [Trenz, 2011]. A model transformation to the Malaca platform has been presented by Ayala et al. [2010]. The development environment offers several diagrams for modeling organizational structures, interactions, deployment configurations, behaviors, etc. Moreover, model validation based on OCL is provided.

IDK. The *INGENIAS Development Kit*[48] (IDK) [Gómez-Sanz et al., 2008] is a stand-alone Java application (version 2.8) which supports the INGENIAS methodology. It covers several diagrams for capturing the design decisions of the INGENIAS methodology. The integrated *Intelligent Agent Framework* (IAF) generates code for Jade. IAF uses a proprietary code generator which uses annotated code templates. Additionally, IDK is able to generate reports and documentation (e.g. HTML).

Jack DE. As already discussed, JDE is a commercial stand-alone Java development environment for the Jack platform. JDE was a very early agent-oriented tool with graphical modeling support. Unfortunately, it makes extensive use of proprietary technology which hampers interoperability with other tools. However, JDE provides interesting features like graphical tracing of executing plans.

Jadex DE. The platform-specific modeling environment for Jadex was presented in [Kardas et al., 2009]. The development environment is based on Eclipse technology and provides graphical modeling support for Jadex. The model transformation to Jadex XML files has been implemented in MOFSCRIPT.

PDT. According to Padgham et al. [2005a] and Sun et al. [2010], the *Prometheus Design Tool*[49] (PDT) is an Eclipse-based modeling tool for the Prometheus methodology. PDT offers various diagrams for the analysis and design phases. The

[47] `http://www.eclipse.org/epf/`
[48] `http://sourceforge.net/projects/ingenias/`
[49] `http://www.cs.rmit.edu.au/agents/pdt/`

Prometheus language is not based on Ecore and uses an XML-based approach. It offers code generation for the Jack platform.

SDE. According to Dikenelli [2008], the *SEAGENT Development Environment* (SDE) is an Eclipse-based tool for modeling SEAGENT agents. The modeling language is based on Ecore and provides modeling support for HTN-based behaviors. The diagrams are well structured. SDE also integrates the SEAGENT platform.

TAOM4E. According to Morandini et al. [2011], the *Tool for Agent-Oriented visual Modeling for the Eclipse platform*[50][51] (TAOM4E) provides modeling support for the Tropos methodology. TAOM4E offers code generation support for Jadex through the *t2x* plug-in. Several auxiliary tools like *eCat* [Morandini et al., 2008] for goal-oriented testing and the T-Tool[52] for checking a specification, based on *Formal Tropos*, for consistency have been developed.

WOLF. WOLF [Caire et al., 2008] is an Eclipse plug-in which provides behavior modeling support for WADE (Jade). The modeling support is restricted to behaviors. It is possible to trace workflows at runtime. Although the idea behind WADE and WOLF is simple, the provided abstraction is useful in real world applications.

Table 2.3 summarizes the introduced agent-oriented modeling tools. The most famous target platforms are Jade and Jadex. Jack is supported by commercial tools developed by *Agent-Oriented Software*[53] (AOS). Most modeling tools are based on Eclipse technology. Unfortunately, agent-oriented approaches make only little use of MDA and EMF. This hampers interoperability between the different approaches.

Summary

This section presented an overview of the state-of-the-art in MDSD and model-driven AOSE. Section 2.1 discussed the role of models, metamodels, and model transformations in MDA. Afterwards, Section 2.2 provided an overview of agent architectures and platforms, agent-oriented methodologies, modeling languages, and modeling tools. As can be seen by the discussions in this section, the research

[50]`http://code.google.com/p/taom4e/`
[51]`http://selab.fbk.eu/taom/`
[52]`http://disi.unitn.it/~ft/ft_tool.html`
[53]`http://aosgrp.com/index.html`

	Methodology	Target Platforms	Technology
agentTool III	O-MaSE	Jade	Eclipse, GEF, XML
DDE	—	Jack, Jade, Jadex, ISReal (Jadex)	Eclipse, EMF, GMF, OCL, QVT, ATL, XPand
IDK	INGENIAS	Jade	Java, GOPRR/EMF
Jack DE	—	Jack	Java
Jadex DE	—	Jadex	Eclipse, GMF, MOFSCRIPT
PDT	Prometheus	Jack	Eclipse, GEF, XML
SDE	—	SEAGENT	Eclipse, GEF
TAOM4E	Tropos	Jadex	Eclipse, GEF, EMF
WOLF	—	Jade	Eclipse, GEF, Java

Table 2.3: This table summarizes the presented modeling tools.

field of model-driven AOSE is quiet diverse. A modeling approach has to consider the agent architecture, features of the execution platform, different software languages, and methodologies. Only few approaches make use of standardized technology developed by traditional software engineering. Chapter 3 introduces a model-driven framework for AOSE which unifies the benefits of platform-specific and platform-independent agent modeling approaches based on MDA principles.

Chapter 3

The Bochica Framework

The Bochica framework for model-driven AOSE follows a language-driven approach to the development of agent-based systems. In our point of view, an ideal agent-oriented modeling language would have to be tailored to a certain application domain, agent architecture, execution environment, and methodology (see Figure 3.1). At the same time, the modeling language should abstract from technical details in order to separate system design from implementation. The better a modeling language covers the concepts of a SUC, the less manual refinements are required at code level. The Bochica framework approaches this problem by an expressive platform-independent agent-oriented core DSL and several extension interfaces which are used to tailor the core DSL to the engineer's needs. The remainder of this chapter provides an overview of the core DSL (Section 3.1) and discusses the integration of Bochica into typical software development processes (Section 3.2). Moreover, the conceptual interfaces for extending Bochica are specified (Section 3.3) and an extensible transformation architecture for bridging the gap to concrete code is proposed (Section 3.4). Finally, a brief overview of the used technology stack is provided (Section 3.5).

3.1 Core Modeling Language

The Bochica core DSL is based on a platform and application domain-independent agent-oriented modeling language, called Dsml4Mas. Dsml4Mas was initially introduced by Hahn et al. [2009a]. As part of this dissertation, it has been further developed regarding the covered concepts, expressiveness, and extensibility. Bochica structures the language concepts into three different degrees of abstraction (see Figure 3.2). The *macroscopic* layer covers concepts for defining the organizational structures of a SUC. The internal structure and behavior of agents is defined by concepts of the *microscopic* layer. Finally, the *deployment* layer encom-

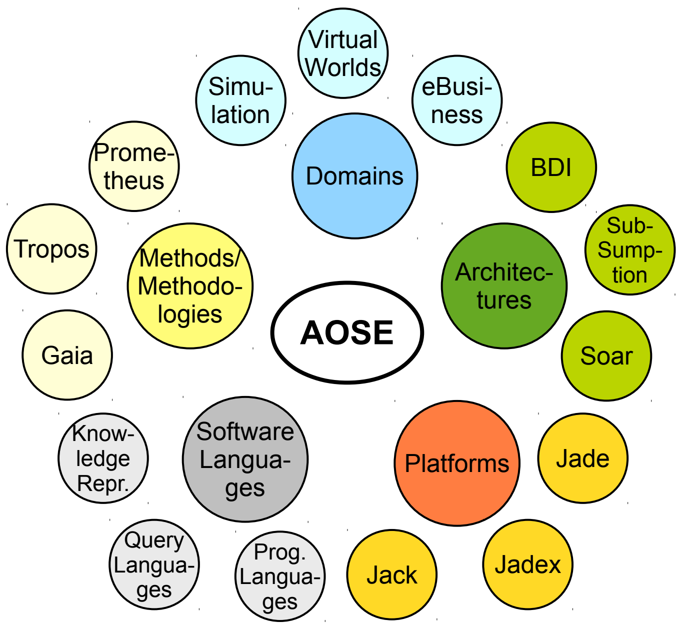


Figure 3.1: This figure depicts an overview of the diverse field of AOSE. The perfect modeling language for AOSE would have to be tailored to a custom configuration of the different aspects.

passes concepts for specifying deployment configurations. Agent platforms usually do not exist in isolation. In real world applications, the target agent platform is embedded into a larger execution environment with additional IT components. The setting depends on the application domain and the concrete use case. Since the core DSL focuses on the core concepts of MAS, it fails to address concepts specific to certain domains. The aim of the extension mechanism is to prevent the core DSL from getting cluttered by concepts that are only relevant for a small subset of applications. Moreover, by extending Bochica large parts of the existing infrastructure can be reused. The common conceptual foundation also eases the reuse of model artifacts. In the following, we provide a brief overview of the three layers. The abstract syntax of Dsml4Mas is presented in Chapter 4.

Macroscopic. The overall structure of a MAS is specified by Dsml4Mas in terms of organizational structures. The responsibilities inside an `Organization` are represented by `DomainRoles` and `AbstractGoals`. The communication between the involved parties is defined by `Interactions`. An `Interaction` defines the valid message sequences between interaction roles (called `Actors`). The design artifacts of the macroscopic layer serve as a contract between the agents and are used for deriving the basic structure of artifacts of the microscopic layer (e.g. behaviors).

Microscopic. The microscopic layer of Bochica defines concepts for modeling the internals of agents. This encompasses concepts like `Behavior`, `Event`,

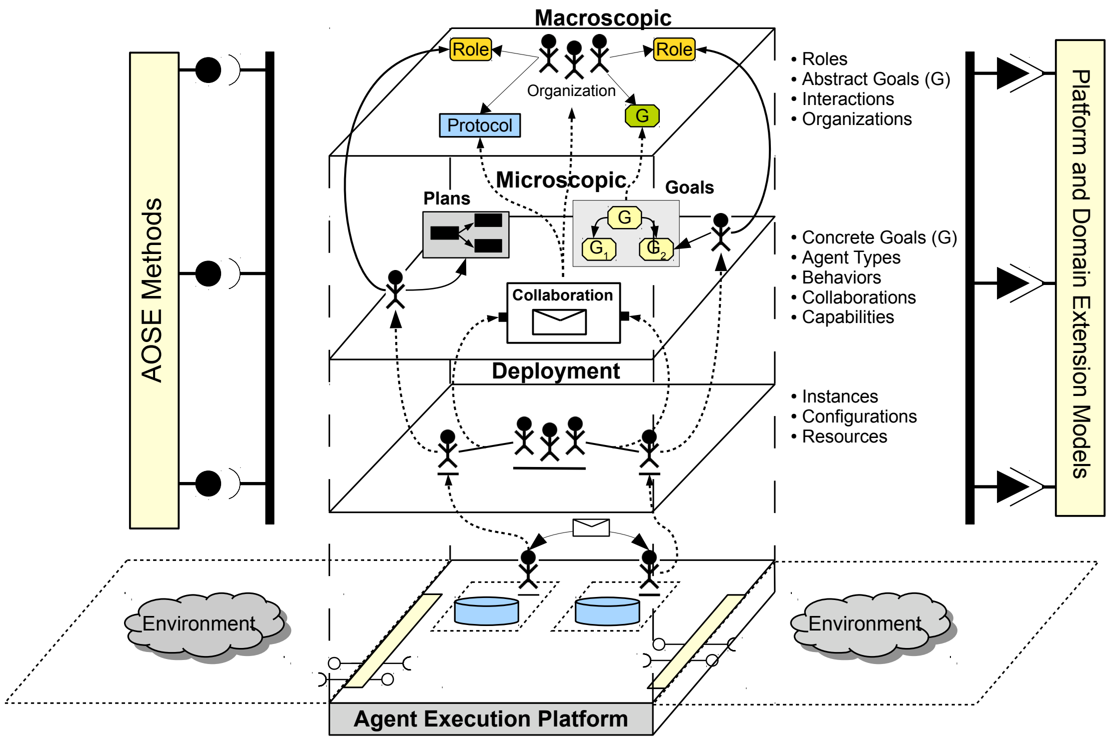


Figure 3.2: This figure depicts the big picture of the Bochica framework. The bottom layer shows a target agent platform embedded into a larger system environment. The modeling language underlying the Bochica framework is structured into a macroscopic, microscopic, and deployment layer. The Bochica framework provides interfaces for customizing the core DSL to custom application domains, execution environments, and software development processes.

`Resource`, `KnowledgeBase`, and `Collaboration`. A `Behavior` specifies the behavior of agents in terms of `Activities` which are connected by `Flows`. `ConcreteGoals` are used to refine the `AbstractGoals` from the macroscopic layer. The internals of an `Organization` are defined by `Collaborations`. A `Collaboration` specifies the bindings between roles of an `Organization` and `Actors` of an `Interaction`.

Deployment. The deployment layer specifies concrete instances of `Agents` and `Organizations` defined by the microscopic and macroscopic layers. Moreover, an `AgentInstance` contains `Initializers` for specifying the initial state of an agent.

3.2 Development Process

The role of Bochica in the overall software development process is to provide the means for capturing design decisions and bridging the gap between design and code (see Figure 3.3 a). Instead of proposing yet another agent-oriented methodology, we align the Bochica framework to already existing ones. Thus, Bochica is

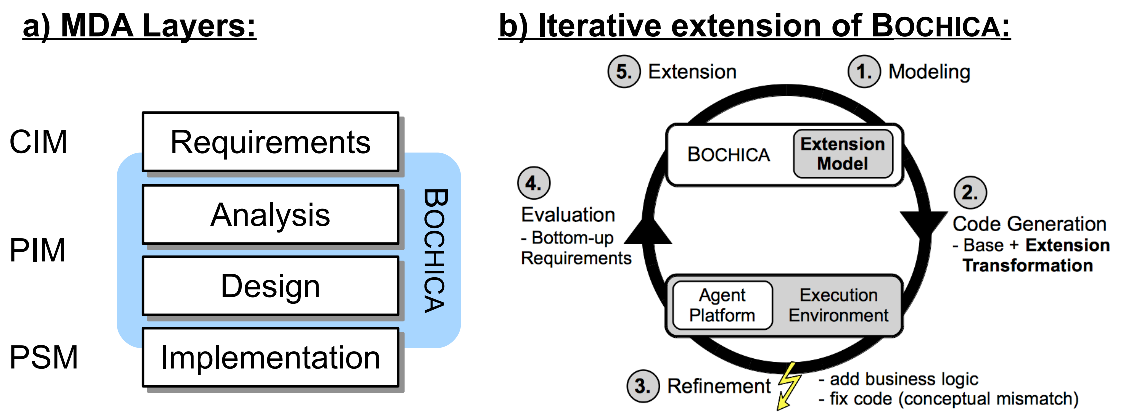


Figure 3.3: (a) depicts the different abstraction layers as defined by MDA. The blue box indicates that the conceptual framework underlying Bochica is dedicated to the analysis and design layers. The ability to extend Bochica with additional method, application domain, and platform-specific concepts simplifies the transition from CIM to PIM and from PIM to PSM. (b) shows an iterative adaptation process for customizing the Bochica framework.

orthogonal to the existing methodology-driven approaches introduced in Section 2.2.2. As of today, iterative and agile software development processes turned out to better meet the demands of software projects than strictly sequential ones. This has also been recognized by the majority of agent-oriented methodologies (see Section 2.2.2). In the following, we take the widely applied RUP as basis for the discussion of how to integrate Bochica into a typical iterative software development process.

According to Kruchten [2003], RUP distinguishes between the four phases *inception*, *elaboration*, *construction*, and *transition*. Each phase undergoes at least once the whole cycle from requirements to code and produces a deployable artifact (see Figure 2.7). Each phase in RUP is dedicated to answers certain questions. For example, the inception phase focuses on determining the feasibility of the overall project, while later iterations phases narrow down the concrete software architecture. Thus, the possibility to produce early prototypes which can be refined in later iterations is important to RUP. The gained experiences of each iteration are used to derive requirements for the next iteration. Of course, the prototype of the first iteration is simple and gets refined during the succeeding iterations. A further output of the inception phase is the definition of the concrete development process (e.g. the utilized methods) and the used tools.

In the context of AOSE, the O-MaSE approach provides a customizable process framework which composes an individual development process by method fragments (similar to RUP). In order to adapt Bochica to a development process, the framework provides interfaces for integrating agent-oriented methods (see Section 3.3.4). The interface enables 3rd-party providers to define own model artifacts which extend and complement the Bochica core DSL. For example, as part of the Prometheus methodology so called *functionalities* of the system are identified in the system specification phase. In the design phase, related functionalities are grouped together to define agent types. In order to support the Prometheus method for the system specification phase, additional model artifacts for capturing the design decisions with Bochica are required. This is enabled by the method interface presented in Section 3.3.4.

3.2.1 Iterative Adaptation

One of the main reasons in model-driven AOSE which causes design and code to diverge over time is that the modeling language is not expressive enough for capturing all important design decisions. This makes extensive manual code refinements necessary. We propose an iterative adaptation process to gradually tailor Bochica to the needs of user-specific application domains and execution environments. The custom concepts build a so called *extension model*. The activities of the adaptation process depicted in Figure 3.3 b) are characterized as follows:

- **Modeling.** The Bochica core DSL is used to capture the design decisions of the SUC. As the core DSL gets extended by an extension model in later iterations, the framework minimizes the need for customizations at code level.

- **Code Generation.** A forward transformation is used to automatically generate code for a target environment. We distinguish between a *base transformation* which maps the concepts of the core DSL to code and an *extension transformation* which complements a base transformation with mapping rules for the custom concepts of the extension model.

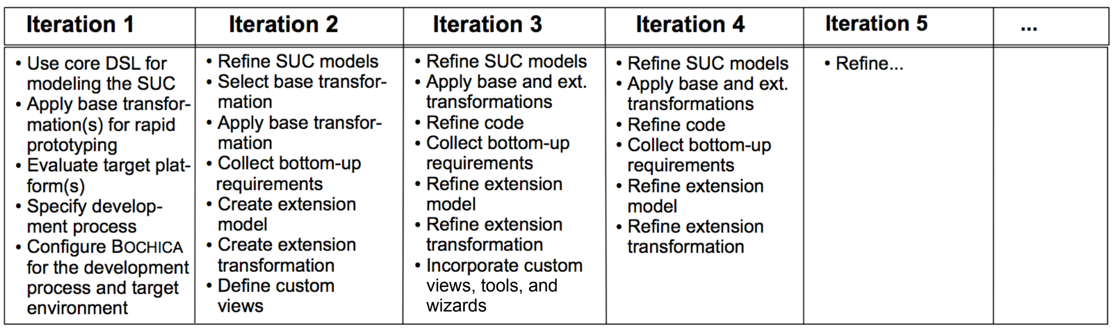
- **Refinement.** The generated code is refined by adding business logic where necessary. Concepts for capturing the design decisions for the SUC which are not covered by the modeling language and/or the model transformation are manually refined at the code level.

- **Evaluation.** Aspects of the SUC which required manual refinements of the generated code are candidates for further extensions of Bochica. This is especially the case when there are no adequate concepts for expressing important design decisions. We call those requirements *bottom-up requirements*.

- **Extension.** The collected requirements are used to create an extension model that extends the Bochica core DSL with missing concepts. Moreover, required views and tools are integrated. Finally, an extension transformations is created with additional mapping rules for the custom concepts.

Figure 3.4 depicts an example development process which exemplifies the adaptation of Bochica to a target environment over several iterations. It is assumed that there already exists an ecosystem consisting of base transformations from Bochica to various target platforms (e.g. Jack and Jadex). The initial iteration is dedicated to determine the target platform, the definition of the development process, and the configuration of the tool stack. For this purpose, a simple prototype of the SUC is designed using the Bochica core DSL. Existing base transformations are used for mapping the model to different target platforms. If no transformation is available for the considered target platform in the initial iteration, the design has to be mapped manually. The generated code can be used to evaluate the target platforms and estimate the adaptation effort. Finally, the individual development process is defined (similar to RUP and O-MaSE) and the target platform and additional tools are selected (e.g. Bochica plug-ins). At the beginning of the second iteration, Bochica is configured with the available plugins. The models of the SUC are refined and the base transformation to the selected target platform is applied. Based on the gained experiences with the code generation and the manual refinement, bottom-up requirements are collected (e.g. using UML class diagrams). An extension model is created which extends the core DSL with the missing concepts. Moreover, an extension transformation complements the base transformation with additional conceptual mappings. The third iteration further refines the models of the SUC using the additional concepts and uses the extension transformation for generating executable code. The conceptual gap is gradually closed with each iteration. This process continues until the language suffices the engineer's needs.

It is important to note that the goal of the customization process is not to clutter Bochica with arbitrary low-level concepts. A balance between abstraction and custom concepts has to be found. The goal is to complement the core DSL for missing concepts which are required to capture the important design decisions for the SUC. However, the balance between abstraction and platform-specific details also depends on the demands of the engineers. The conceptual extension of the core DSL requires an experienced language engineer who finds the right level of abstraction. The discussion in this section mainly focused on the alignment of Bochica to the iterative RUP. However, we see no restrictions for transferring the approach to other iterative and non-iterative development processes. For example,



Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	...
• Use core DSL for modeling the SUC • Apply base transformation(s) for rapid prototyping • Evaluate target platform(s) • Specify development process • Configure BOCHICA for the development process and target environment	• Refine SUC models • Select base transformation • Apply base transformation • Collect bottom-up requirements • Create extension model • Create extension transformation • Define custom views	• Refine SUC models • Apply base and ext. transformations • Refine code • Collect bottom-up requirements • Refine extension model • Refine extension transformation • Incorporate custom views, tools, and wizards	• Refine SUC models • Apply base and ext. transformations • Refine code • Collect bottom-up requirements • Refine extension model • Refine extension transformation	• Refine...	

Figure 3.4: This figure depicts an example development process for adapting Bochica to a target environment.

sequential processes (similar to the waterfall model presented in Section 2.2) also benefit from specialized concepts for an application domain or execution environment. In the following, Section 3.2.2 defines the stakeholders and their tasks for the application of Bochica and Section 3.2.3 discusses the Bochica ecosystem.

3.2.2 Framework Stakeholders

The application of the Bochica framework requires the interplay of different stakeholders. In the following, we characterize the involved parties and define their tasks. It is important to note that methodologies and software development processes define own stakeholders which are not specific to Bochica.

Language Engineer. Since Bochica is based on a DSL, the language engineer is responsible for extending it with new concepts. Detailed knowledge of the core DSL and the underlying metamodel is required to align new concepts to existing ones. We distinguish between (a) language engineers who further develop the core DSL and (b) those who create 3$^{\text{rd}}$-party extensions. The first one is not involved in the development process. The latter one is involved in the evaluation and extension tasks. The language engineer has to choose the right level of abstraction for the conceptual extensions.

Tool Developer. The tool developer is responsible for building the development environment based on the core DSL. This includes (i) writing model transformations (ii) creating new or extended diagrams, and (iii) providing further usability extensions such as wizards and additional tools. He has to make sure that the tools cover the (required) functionality of the target platform. The tool developer is involved in the evaluation and extension tasks.

Agent Engineer. The agent engineer is the end user of the development environment. According to his needs, he installs the required plug-ins and uses an agent methodology to design the MAS. Model repositories are used to cooperate with colleagues and reuse existing model artifacts. The agent engineer is also responsible for refining the generated code where necessary. He is involved in the modeling, code generation, and evaluation tasks.

3.2.3 Ecosystem

The development of real world agent-oriented software systems requires the collaboration of different parties. We expect that an ecosystem of providers which offer agent-oriented solutions (based on Bochica) for certain application domains and user groups will establish (see Figure 3.5). One group of providers focuses on the creation of extension bundles which address the demands of certain application domains. Other providers offer mature and tested model artifacts which can be incorporated into software projects. Furthermore, the providers offer consulting and training for model-driven AOSE and the modeling solutions.

The actual end-users utilize the infrastructure and functionality provided by the Bochica framework to configure the development environment for a SUC. The common conceptional core provided by the Bochica framework enables the exchange and reuse of model artifacts. For example, BDI experts use organizational structures and means-end decompositions for decomposing a problem into smaller sub-problems. Protocol experts are responsible for specifying communication and negotiation protocols. Research results (e.g. about the properties of a specific auction protocol) can be directly linked to model artifacts in the model repository. This information helps engineers in constructing MAS. Specialized views enable the experts to abstract form other details. Validated and tested model artifacts are shared through model repositories.

3.3 Extension Interfaces

This section specifies the extension interfaces of the Bochica framework. The benefit of extending Bochica in opposite to creating a custom solution is that large parts, which are common to most MAS, can be reused. In principle, each concept of the underlying core DSL can be specialized. However, the remainder of this section provides an overview of central concepts which we consider most interesting for 3rd-party extensions. The interfaces are defined using the underlying Bochica metamodel and additional OCL constraints. An overview of the complete metamodel is provided in Chapter 4. In the following, we distinguish between conceptual extensions for application domains and execution environments (Section 3.3.1), the integration of external software languages (Section 3.3.2), the data model (Section 3.3.3), and the method interface (Section 3.3.4).

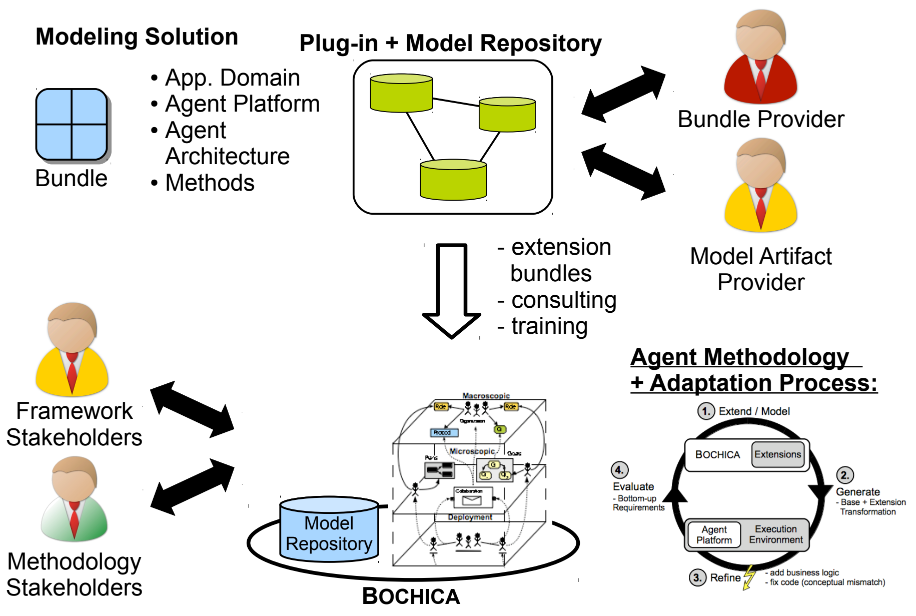


Figure 3.5: The Bochica ecosystem.

3.3.1 Conceptual Interface

Conceptual extensions of Bochica are used (i) to introduce alternative ways of modeling existing aspects (e.g. behaviors or interaction protocols), (ii) to introduce new aspects which are not covered by Bochica (e.g. commitments), and (iii) to define customizations for a certain application domain or execution environment. In the following, an overview of the agent, behavior, and interaction aspects of Bochica is provided. We consider those aspects most relevant for extensions.

Agent Interface

Figure 3.6 depicts the metamodel which defines the concept of `Agent` in Bochica. An `Agent` is an autonomous entity which performs `DomainRoles` of organizational structures, has access to `Resources`, and uses `Sensors` to perceive its environment. An agent's beliefs about the environment are stored in a `KnowledgeBase` for reasoning and planning. `Goals` represent desired goal states of an `Agent` and `Behaviors` define plan templates which guide an agent's actions in order to achieve a `Goal`. A `Capability` groups closely related `Behaviors` and `Knowledges` (variable slots) to a functional unit. Concepts like `Sensor`, `Resource`, and `KnowledgeBase` highly depend on the application domain and execution environment. Thus, they are primary candidates for 3rd-party extensions. For example, a `KnowledgeBase` requires a knowledge representation language for encoding the beliefs. Which language to use depends on the requirements of the SUC.Concepts like `DomainRole`

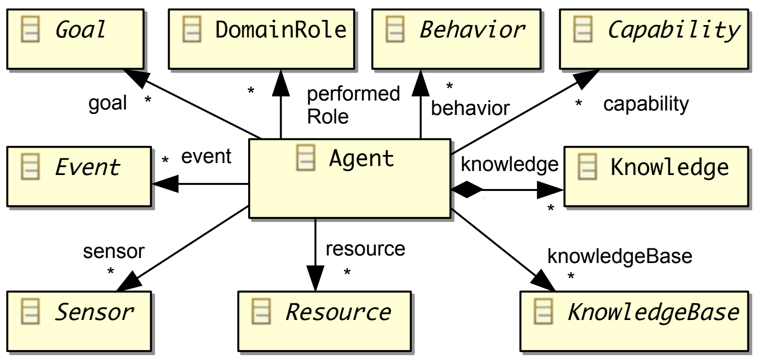


Figure 3.6: This figure provides a high-level overview of the definition of the concept `Agent` in Bochica.

and `Knowledge` interleave the concept of `Agent` with other aspects of Bochica (e.g. organizational structures and the data model).

Behavior Interface

The concept of `Behavior` is an abstract placeholder for any kind of behavior specification in Bochica. The Bochica framework provides one concrete type of behavior, called `Plan` (see Figure 3.7). A `Plan` consists of `Activities` which are connected by `Flows`. Moreover, a `Plan` contains `Knowledges` for storing local information. A `Task` is an atomic `Activity` which is not further decomposed. The `Knowledges` contained by a `Task` are the task's input and output parameters. The enumeration `ParameterDirection` defines the access rights to `Knowledges`. `StructuredActivities` are `Activities` that contain nested `Activities`. Bochica provides a default set of `Tasks` and `StructuredActivities` which are introduced in Chapter 4.

One often neglected aspect in agent-oriented modeling languages is a clean scoping mechanism of variables between agents and behaviors. Figure 3.8 depicts how an `Agent's Knowledges` are passed to `Plans`. The `PlanUse` concept contains a set of `KnowledgeBindings` which define a mapping between `Knowledges` of an `Agent` to `Knowledges` of a `Plan`. The scoping mechanism also enables an `Agent` to use the same `Plan` in different configurations. Listing 3.1 depicts additional OCL invariants which constrain the binding. Furthermore, a `Plan` consists of nested `StructuredActivities` that span a sub-scope with own `Knowledge` declarations. Inside a `StructuredActivity`, the `Knowledges'` names have to be unique (see OCL invariants in Listing 3.1). The `getVariable()` operation of `Activity` is responsible for resolving a variable symbol in the surrounding scope if it cannot be resolved in the local one. The described mechanism significantly increases the expressiveness of Bochica models and eliminates imprecisions of conceptual

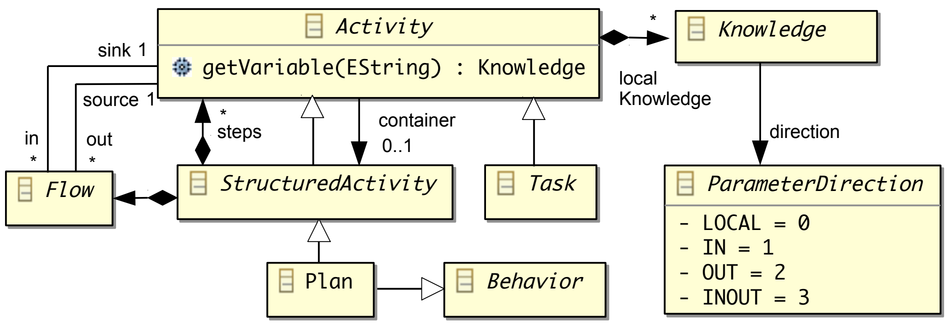


Figure 3.7: `Tasks`, `StructuredActivities`, and `Flows` are core concepts of `Plans`. The `ParameterDirection` defines access restrictions on `Knowledges`.

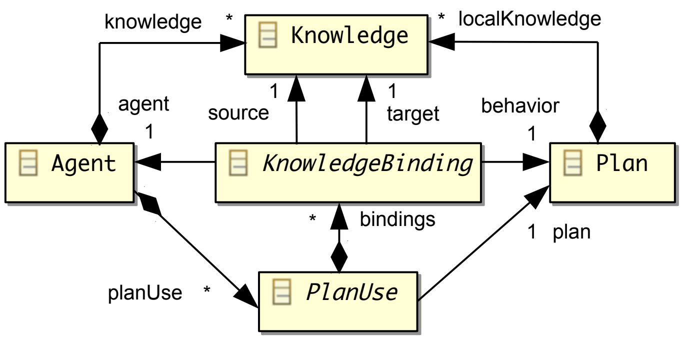


Figure 3.8: `PlanUses` define `KnowledgeBindings` for providing `Plans` access to an `Agent's Knowledges`.

mappings to execution platforms. Moreover, it defines a clean interface for conceptual extensions. Most 3rd-party behavior extensions to Bochica will customize the existing `Plan` definition. Especially `Tasks` and `StructuredActivities` are interesting for specifying custom `Activities` for the interaction with the execution environment. However, some application domains might require a different realization of `Behavior`.

Interaction Interface

Interaction protocols are used within MAS to specify valid message sequences for the communication between agents. The implementation of agent communication protocols by software engineers is one of the most demanding tasks in AOSE since MASs are concurrent systems. Usually, agent execution platforms like Jade, Jack, or Jadex only provide a platform messaging service (e.g. FIPA-based) and a low-level API which enables the engineer for sending and receiving messages on a programmatic level. In most cases, there exists no explicit representation of communication protocols at the platform level. Bochica follows a contract-based approach for the specification of interaction protocols. This means that there exists

```
// source knowledge is part of agent
context PIM4Agents::behavior::KnowledgeBinding inv:
self.agent.localKnowledge -> includes(self.source)

// target knowledge is part of behavior
context PIM4Agents::behavior::KnowledgeBinding inv:
self.behavior.localKnowledge -> includes(self.target)

// source and target knowledges have same type
context PIM4Agents::behavior::KnowledgeBinding inv:
self.source.type = self.target.type

// parameter direction has to be IN, OUT, or INOUT
context PIM4Agents::behavior::KnowledgeBinding inv:
self.target.direction <> PIM4Agents::informationmodel::
    ParameterDirection::Local

// names have to be unique inside a scope
context PIM4Agents::behavior::StructuredActivity inv:
self.localKnowledge -> forAll(d |
  self.localKnowledge.name -> select (e | e.name = d.name)
    -> size() = 1)
```

Listing 3.1: OCL invariants for scoping between agents and behaviors. The context defines the metamodel concept the invariant is defined for and `self` refers to an object the invariant is checked for.

a global specification of valid message sequences, conversation states, timeouts, etc. (similar to UML interaction diagrams) that takes the function of a contract for the involved interaction roles. The protocol specification is used to generated model artifacts (e.g. behavior templates) which can be used as starting point for the behavior specification. In previous work, Hahn et al. [2011] proposed an approach for generating behavior templates from Pim4Agents interaction protocols. Those behavior templates use explicit send/receive tasks and have to be manually refined (e.g. with business logic). However, the approach resulted in complex behavior patterns (like those on the platform level) since the full complexity of sending messages, collecting responses, handling timeouts, etc. was lifted almost one-to-one to the platform-independent layer. The benefit of the approach was a partial generation of the required behavior structures. However, the question arises, how to abstract from fine-grained behaviors with explicit send/receive tasks and shift the focus on a higher level of abstraction and on the business logic.

In the following, we abstract from concrete protocol specification approaches and discuss the interfaces for integrating 3rd-party protocol specification languages

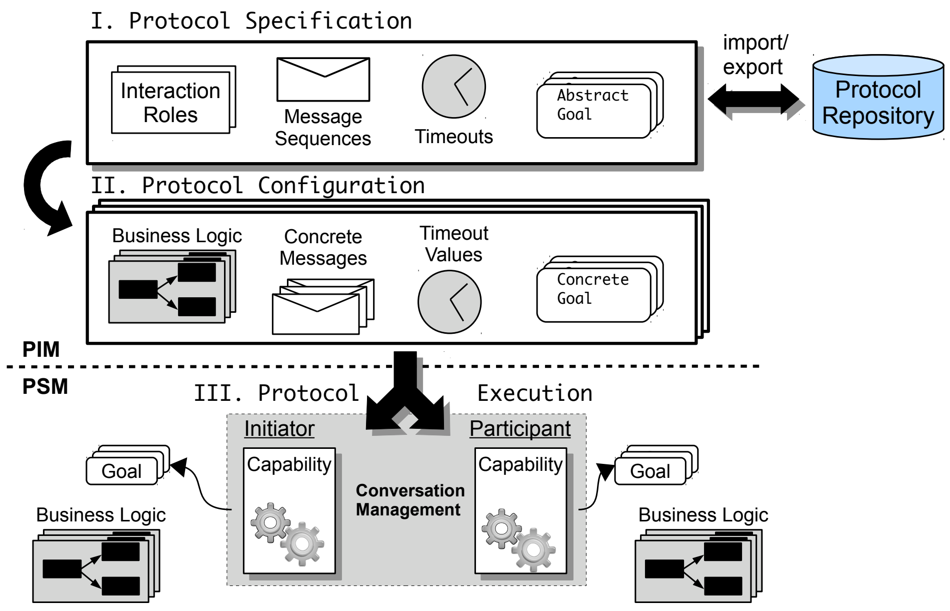


Figure 3.9: This figure depicts an overview of how to incorporate interaction specification approaches into Bochica.

into Bochica (see Figure 3.9). Bochica distinguishes between (i) the actual protocol specification, (ii) the protocol configuration, and (iii) the implementation and execution. A protocol specification defines interaction roles, the message sequences, and timeouts. Moreover, the responsibilities of the involved interaction roles, when a certain state of the interaction is reached, are represented by abstract goals. A protocol configuration defines (i) concrete role bindings, (ii) concrete message types (including content types), concrete timeout values, (iii) concrete goals realizing the abstract goals of the protocol specification, and (iv) business logic for processing the concrete goals. There can be an arbitrary number of protocol configurations for one protocol specification. The underlying assumption is that the protocol specification is expressive enough to generate code which controls the protocol execution (including sending/receiving messages, handling timeouts, etc.). Thus, the engineer's task is reduced to (i) import a required protocol, (ii) define role bindings, concrete messages, and timeout values, and (iii) realize abstract goals and provide the business logic. The execution of the protocol is transparent for the agent modeler as the code for message handling and sending is generated. As the protocol specification is mapped to the execution platform, the platform-independent artifacts are mapped to artifacts which make use of the platform's messaging service and API. This might be done by interpreting the interaction protocols or generating capabilities and code which handle the execution (e.g. conversation management).

The Bochica metamodel defines the abstract concept `Interaction` which is an abstract placeholder for interaction protocol specifications. A `ProtocolConfiguration` defines how an `Interaction` is used in a concrete setting. For this purpose, role bindings between `DomainRoles` and `Actors`, the content types (concept `Type`; see Section 3.3.3) of messages, and concrete timeout values are defined. The interface between a protocol specification and the concrete behavior is defined by `AbstractGoals`. An `AbstractGoal` represents the business logic to be executed when a conversation state is reached. The Bochica framework defines one type of `Interaction`, called `Protocol`. The `Protocol` concept has been intensively discussed by Hahn et al. [2011]. León-Soto [2012] proposed an alternative extension of the `Interaction` concept for defining modular protocols. At the time of the creation of that extension, the Bochica framework was not available so that it was hard-wired into the core DSL. With the Bochica framework it is now possible to integrate such extensions without touching the core DSL. End users can choose which approach to use. The interaction metamodel is discussed in more detail in Section 4.1. A design pattern for mapping `Interactions` and `ProtocolConfigurations` to code is proposed in Chapter 5 as part of the forward transformation. Finally, the case study presented in Section 8.1 provides a concrete example.

3.3.2 Software Language Interface

There exists a large number of software languages that are relevant for developing agent-based systems such as (i) knowledge representation languages, (ii) reasoning languages, (iii) rule languages, (iv) communication languages, and (v) programming languages. A software language is always developed with a certain purpose in mind. Thus, it depends on the concrete use case which one to use. Bochica provides abstract language interface concepts which can be extended by external language plug-ins (see Figure 3.10). The main concept is `Expression`. There exist several specialized expression types such as `BooleanExpression`, `Rule`, and `ContextCondition`. The abstract expression types are used throughout the framework. For example, an `AchieveGoal` has a target and failure condition of type `BooleanExpression` and a `Plan` has a context condition of type `ContextCondition`. `ControlFlows`, which are used to specify the execution order of `Activities`, can specify a precondition of type `BooleanExpression`. External plug-ins can specialize the abstract expression types with concrete languages. We assume that an external language is also based on Ecore. This is not a hard restriction since more and more software languages such as Java and *SPARQL Protocol And RDF Query Language* (SPARQL) [W3C, 2008b] are becoming available in public metamodel

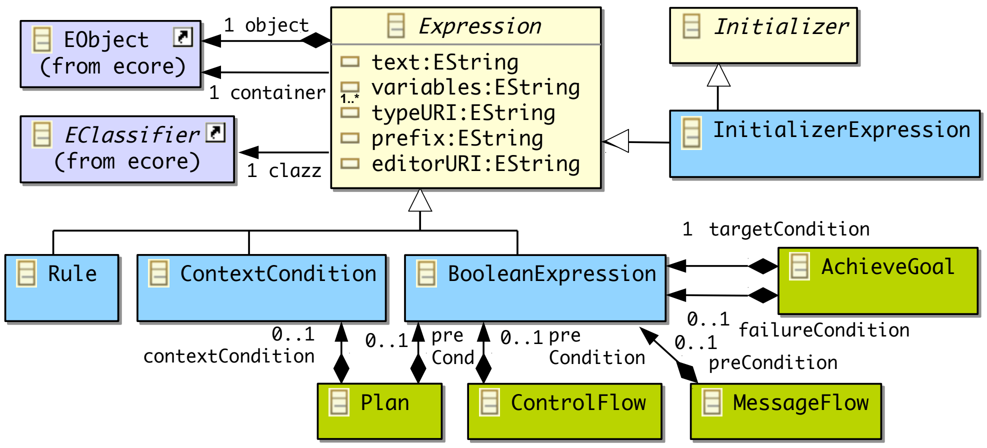
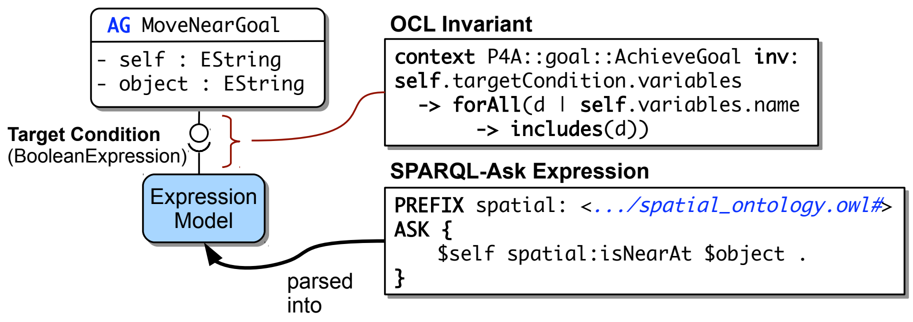


Figure 3.10: The `Expression` concepts abstracts from concrete software languages. The four concepts `Rule`, `ContextCondition`, `BooleanExpression`, and `InitializerExpression` are specializations of `Expression` and are used throughout the framework.

zoos (e.g. EMFText concrete syntax zoo[54], Atlantic metamodel zoo[55]). The `text` attribute of the `Expression` concept holds the plain expression string as defined by the user. We use a reflection-based approach for parsing expression strings into a language-specific expression model (interface concept `EObject`) and assign it to the `Expression` object's `object` attribute (see Figure 3.10). The value of the `typeURI` attribute is set by the external plug-in and is used to resolve the concrete language concept of the external metamodel (attribute `clazz`). The `variables` attribute contains the list of variable symbols of the expression. Since the variable symbols are declared in each software language differently, the computation of the values has to be done by the external plug-in. The `editorURI` can be used to specify an external editor for editing the expression string (e.g. to support syntax highlighting). Finally, the `prefix` attribute is used to display the expression type to the user. Figure 3.11 depicts an example expression in SPARQL. SPARQL is a Semantic Web query language. The shown SPARQL-Ask expression is parsed into a SPARQL model and plugged into the target condition slot of an `AchieveGoal`. The OCL invariant is used to ensure that variable symbols of *any* `BooleanExpression` (here SPARQL-Ask) are bound in the surrounding scope (here `AchieveGoal`). Parsing the expression string into an expression model has following benefits: (i) the syntactical correctness of an expression is ensured, (ii) invariants ensure that variable symbols are bound in the surrounding scope, and (iii) the expression models can be processed by model transformations. Moreover,

[54]`http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo`
[55]`http://www.emn.fr/z-info/atlanmod/index.php/Atlantic`



```
context P4A::goal::AchieveGoal inv:
self.targetCondition.variables
  -> forAll(d | self.variables.name
       -> includes(d))
```

```
PREFIX spatial: <.../spatial_ontology.owl#>
ASK {
    $self spatial:isNearAt $object .
}
```

Figure 3.11: The depicted MoveNearGoal makes an agent (parameter `self`) walk to a target object (parameter `object`). The target condition is defined by a SPARQL-Ask expression which is parsed into a SPARQL model and plugged into the target condition slot of the `AchieveGoal`. The invariant ensures that the variable symbols self and object are bound in the surrounding scope.

the benefit of our approach is that technical details, such as the integration of the knowledge base and reasoning languages into the concrete agent execution platform, are hidden on the modeling level. At the same time, models can be tailored to a certain target environment. Of course, the integration at the platform level has to be done at some point (we discuss it in Section 3.4) but the agent engineer has a consistent view and can concentrate on the design of the overall system.

3.3.3 Data Model Interface

The data model interface of Bochica is based on the abstract concept `Type` (see Figure 3.12). `Knowledges` are used throughout Bochica to store data with a certain `Type`. An `InternalType` encompass Bochica concepts like `Event`, `Goal`, and `Message`. They are required for accessing model artifacts from inside a plan (e.g. the parameters of an `Event`). The `type` attribute of a `Knowledge` defines the type of its value. The `ExternalType` concepts is the interface to external type definitions. The definition of concrete data types has been separated from Bochica and is based on the Ecore metamodel provided by EMF (see Section 2.1.3). Ecore is used to model classes, their operations, attributes, and relations among each other (see Figure 2.4). Reusing Ecore as data model has several advantages: we get (i) graphical modeling support (UML class diagram style) and (ii) import from UML, XML Schema (including XML de-/serialization), and existing Java code for free. `EClassifiers` defined by an Ecore-based data model are made available within Bochica by the concept `EType`. Moreover, Bochica defines basic data structures such as `Sequence`, `Set`, or `HashMap`. The data model of Bochica can be extended by external plug-ins. For example, it would also be possible to base the data type definitions on other metamodels than Ecore.

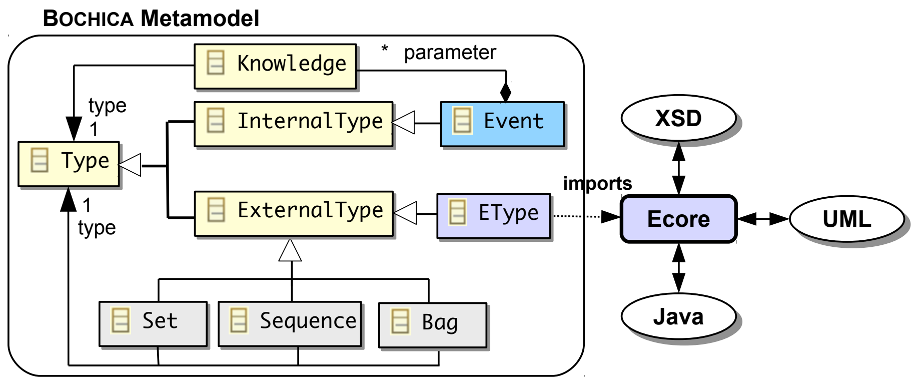


Figure 3.12: The definition of concrete data types has been separated from the core DSL and is based on Ecore. The concept `EType` imports Ecore to Bochica.

3.3.4 Method Interface

During the recent years, many agent-oriented methodologies have been proposed (see Section 2.2.2). In contrast to most other approaches, our research focus was always on the development of an expressive platform-independent agent-oriented modeling language which is compliant to MDA. The role of a modeling language is to provide the means for capturing the design decisions for a SUC. Our research focus was less on the methodology part. Since both aspects are complementary, we use method plug-ins for extending the Bochica framework to the needs of methodologies. In the same way as Bochica can be extended with new agent concepts, methodology providers can contribute plug-ins with new views and method-specific concepts. For example, the Prometheus methodology collects in the system specification phase abstract functionalities and goals of a SUC. In the architectural design phase the functionalities and goals are grouped to agents. A Prometheus extension for Bochica could extend the framework with the missing concepts for collecting abstract functionalities in the system specification phase (since the required concepts are not natively covered by the core DSL). The architectural design phase could be based on existing concepts of the core DSL to benefit from Bochica. The interface concept in Bochica is called `MethodArtifact`. Instead of having a separate modeling language and tool for each methodology, most of the methodologies could be based on the same framework. This joins the efforts of the involved parties and eases maintenance of the tool chain.

3.4 Forward Transformation Architecture

Model transformations in MDA are used to gradually refine a model of a SUC until executable code is generated. The possibility to customize Bochica for certain

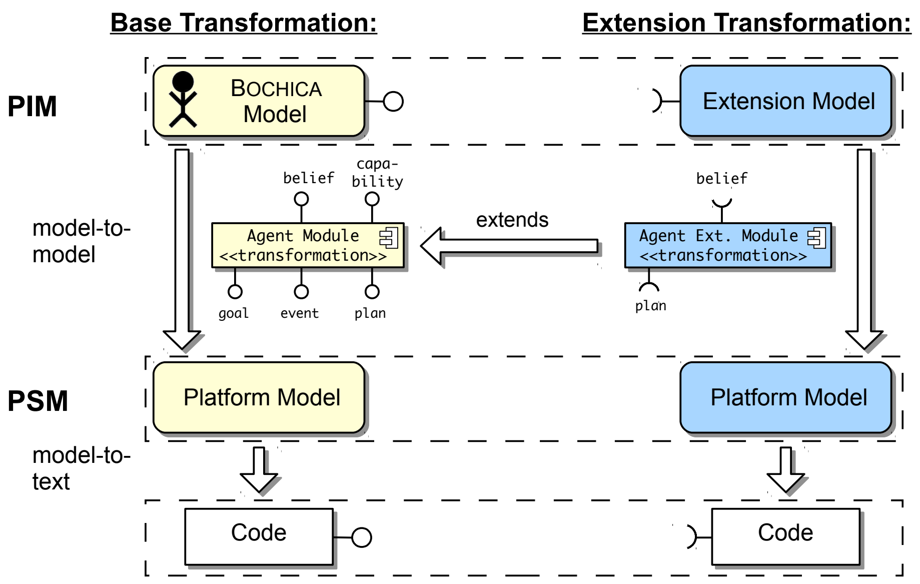


Figure 3.13: A base transformation maps the core concepts of the Bochica framework to an agent platform. An extension transformation complements the base transformation as Bochica gets extended with additional concepts.

application domains and execution environments requires a transformation architecture which is able to handle those extensions. Based on our experiences with model-driven AOSE, we propose a transformation architecture for supporting the Bochica framework (see Figure 3.13). *Base transformations* are responsible for mapping the concepts of the core DSL to executable artifacts of an agent platform. A transformation consists of several *modules* which group closely related conceptual mappings together (e.g. agent mappings or behavior mappings). In Figure 3.13, we visualize transformation modules as UML components with mapping rules as interfaces. Each port can be seen as an extension point that can be modified by external plug-ins. For example, Figure 3.13 depicts an agent module that consists of mapping rules for configuring an agent's beliefs, behaviors, goals, etc. As Bochica gets extended with new concepts, an existing base transformation is no longer complete regarding the covered concepts. Thus, an *extension transformation* is required which extends an existing base transformation for the new concepts (if the target platform shall be enabled for the extension). For example, the extension module in Figure 3.13 might equip every agent with a default plan for the registration at a proprietary execution environment during start-up. We see three possibilities how the extension mechanism of the proposed transformation architecture can be realized. Some model transformation languages (e.g. QVT) allow to write a new transformation which inherits from an existing one. Thus, an existing mapping rule can be overloaded by a new and extended one.

Other transformation languages like XPand use an aspect-oriented approach for hooking into an existing transformation and extending it. A further possibility is to chain transformations $t_0 \circ \ldots \circ t_n$, where t_0 is a base transformation and the succeeding transformation supplements the result of the proceeding ones. The model transformation architecture depicted in Figure 3.13 has been realized with model transformation languages like QVT and XPand (see Chapters 5 and 7). The benefit of our approach is that large parts of the conceptual mappings can be reused as Bochica gets extended. The code generation patterns use by the base transformation will improve over time. Finally, the transformation architecture supports rapid prototyping as described in Section 3.2. During the first iterations, existing base transformations can be used to develop an early prototype. As the core DSL gets customized in later iterations, additional conceptual mappings are added to the extension transformation.

3.5 Technology Stack

This section provides a brief overview of the technology stack used to realize the Bochica framework. The set of components which make up the Bochica implementation is called *DSML4MAS Development Environment* (DDE) 2.0. DDE makes extensive use of MDA standards and Eclipse technology (see Figure 3.14). The abstract syntax of the core DSL has been specified using Ecore and additional OCL constraints. The mapping between the abstract syntax and graphical symbols has been specified using GMF. Bochica models are serialized by EMF into *XML Metadata Interchange* (XMI) [OMG, 2011e] files. Current model-to-model transformations are based on QVT and ATL. Code generation is based on XPand, JET, and MOFSCRIPT[56]. Technically, the Bochica extension mechanism is based on the Eclipse Equinox *Open Services Gateway initiative*[57] (OSGi) framework and EMF. A typical plug-in for Bochica consists of (i) conceptual extensions, (ii) the definition of a custom graphical representation, and (iii) an extension transformation for the required target environment (assuming that the base transformation already exists). Figure 3.15 depicts an overview of DDE. Model repositories store model artifacts such as interaction protocols, behaviors, and organizational structures. The artifacts are reused across projects. DDE provides support for validating model artifacts using OCL constraints. After a SUC has been modeled, forward transformations are used to generate executable code. Likewise, reverse transformations are used to import the underlying design of al-

[56]`http://www.eclipse.org/gmt/mofscript/`
[57]`http://eclipse.org/equinox/`

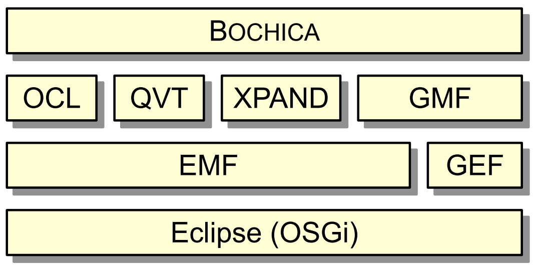


Figure 3.14: This figure depicts the technology stack underlying the BOCHICA framework.

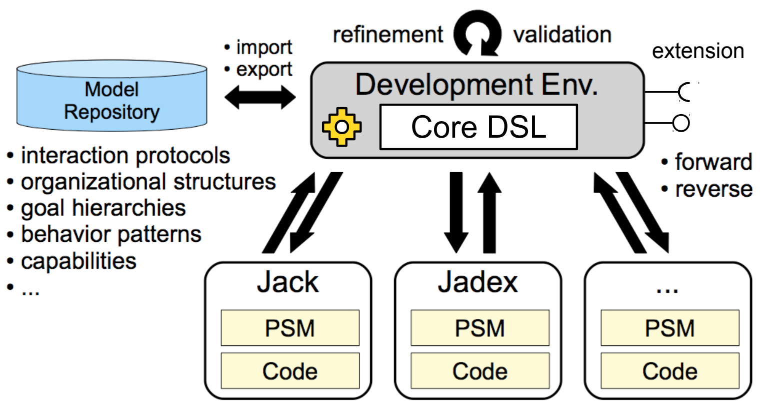


Figure 3.15: Overview of the BOCHICA development environment.

ready implemented MAS. The lifted models are refined, validated, and shared via the model repository (see Chapter 6). The current model repository is file-based but model repositories like Teneo are just becoming available.

3.6 Summary

This chapter introduced the BOCHICA framework for model-driven AOSE. This included an overview of the core DSL, the integration into software development processes, the extension interfaces, a forward transformation architecture, and the technology stack used for implementing the framework. The following chapters introduce a base transformation for the BDI agent platform Jadex (Chapter 5), a reverse transformation for Jadex (Chapter 6), and an extension model and transformation for agents in semantically-enhanced virtual worlds (Chapter 7). Finally, the framework and the extension mechanism are evaluated in Chapter 8 in two case studies.

Chapter 4

Metamodels

Metamodels are used by MDA to specify the abstract syntax (vocabulary) of a modeling language. A modeling language is used for capturing the design decisions of a SUC. The metamodels presented in this chapter provide the conceptual foundation for the mapping rules between Bochica and the Jadex BDI platform defined by Chapters 5 and 6. The Bochica core DSL is based on Dsml4Mas. The abstract syntax of Dsml4Mas is defined by the Pim4Agents metamodel. In order to realize the MDA stack for the Jadex platform, a PSM for Jadex is required. The remainder of this chapter introduces the platform-independent Pim4Agents metamodel (see Section 4.1) and the platform-specific Jadex metamodel (see Section 4.2). Finally, Section 4.3 summarizes this chapter.

4.1 Platform-independent Metamodel for Agents

The Pim4Agents metamodel has been continuously further developed by the DFKI multiagent system research group over the recent years. As part of this dissertation, Pim4Agents has been refined regarding expressiveness and the conceptual extension interfaces. Chapter 3 already introduced selected aspects of Pim4Agents as part of the Bochica extension interface specification. Figure 4.1 depicts the core concepts of Pim4Agents. Section 4.1.1 introduces the concepts of the macroscopic layer, Section 4.1.2 the microscopic layer, and Section 4.1.3 the deployment layer.

4.1.1 Macroscopic Metamodel

The macroscopic layer of Bochica defines the overall structure of a SUC in terms of `Organizations` and `DomainRoles`. Figure 4.2 depicts the according concepts of Pim4Agents. The responsibilities of an `Organization` are represented by

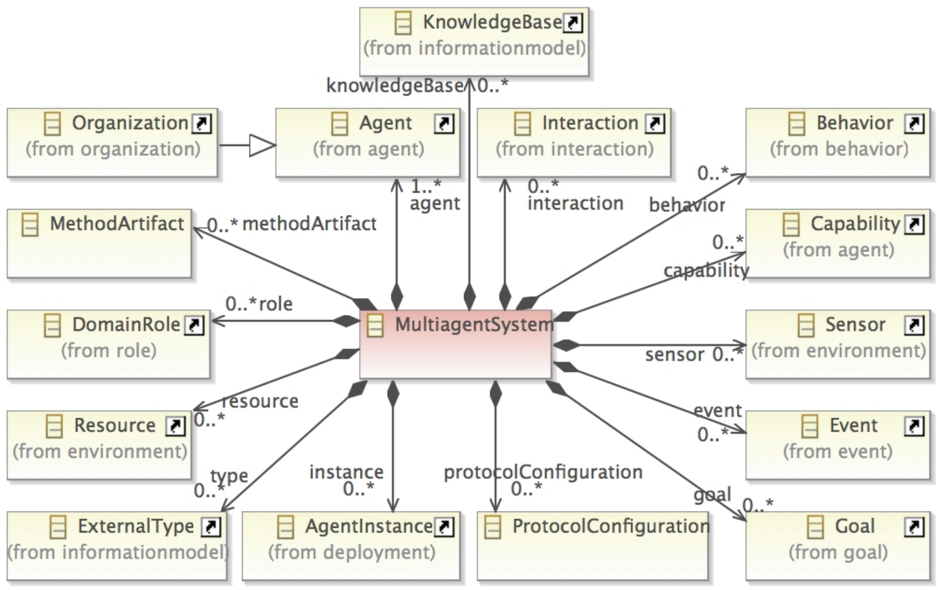


Figure 4.1: This figure depicts the overall structure of a PIM4AGENTS model. The `MultiagentSystem` is the top-most container and aggregates the different model artifacts.

`DomainRoles` and `AbstractGoals`. `DomainRoles` are performed by `Agents` that are modeled at the microscopic layer. `AbstractGoals` can be decomposed by the `subgoal` relationship. The `conflictingGoals` relationship defines conflicts between `AbstractGoals`. The communication inside an `Organization` is specified by `Interactions`. An `Interaction` can be seen as a contract that defines the communication between different `DomainRoles`. `Interactions` play an important part in BOCHICA since they are used to derive `ConcreteGoals` and `Behaviors` of the microscopic layer. Figure 4.3 depicts the PIM4AGENTS interaction metamodel. The `Protocol` concept realizes the abstract `Interaction` concept and defines the valid message sequences between `Actors` (interaction roles). An `Actor` represents a named set of role-fillers at design-time. Moreover, `Actors` can be decomposed into sub-`Actors`. The states of an interaction are represented by `MessageFlows`. A `MessageScope` defines the `ExchangeMode` for sending and receiving `ACLMessages` (e.g. `Sequence` or `Loop`). `MessageScopes` are forked and joined by `MessageFlows` that are contained by `Actors`. Timeouts between sending a message and receiving the response(s) are defined by the `TimeOut` concept. Responsibilities of an `Actor` when a certain state (concept `MessageFlow`) of a conversation is reached (e.g. the evaluation of an incoming call-for-proposal message) are represented by `AbstractGoals`. `AbstractGoals` are realized by `ConcreteGoals` of the microscopic layer. It is important to note that `Interactions` do not define the content types of `ACLMessages`, nor concrete role-fillers and timeout values. For a detailed

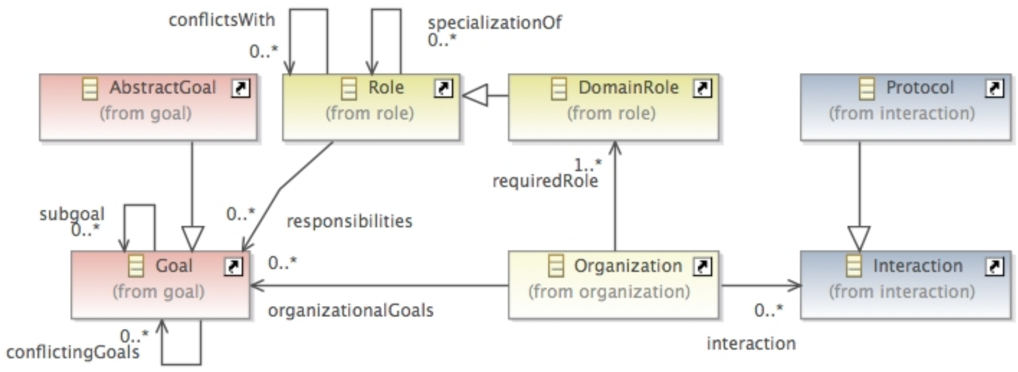


Figure 4.2: This figure provides an overview of the Pim4Agents concepts for the macroscopic layer.

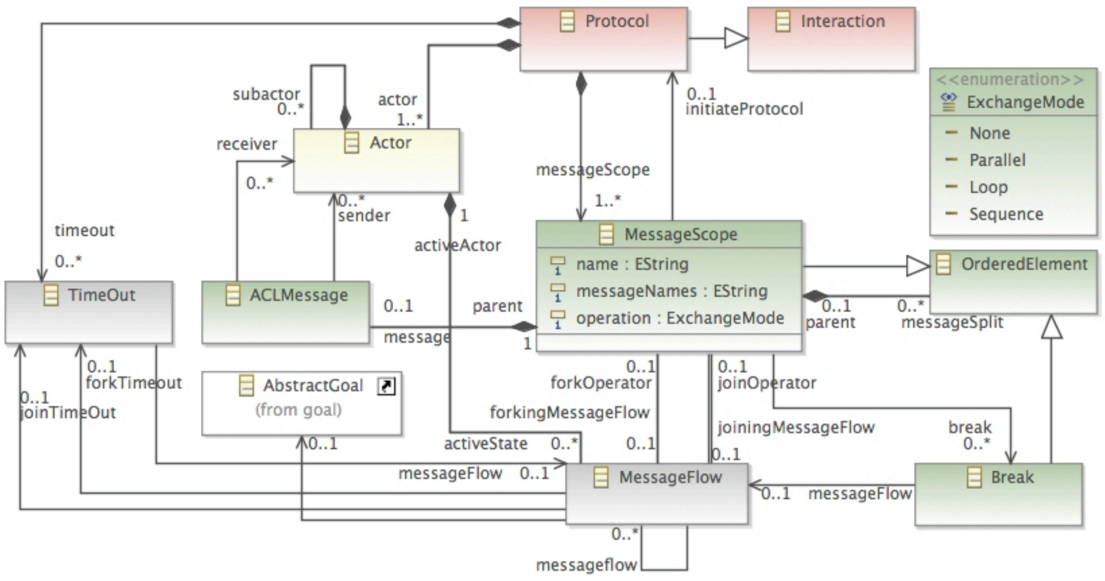


Figure 4.3: The Pim4Agents interaction metamodel.

overview of the Pim4Agents interaction metamodel we refer to [Hahn et al., 2009b] and [Hahn et al., 2011].

4.1.2 Microscopic Metamodel

The concepts of the microscopic layer provide the conceptual foundation for the specification of agent and capability types, the internals (role bindings) of organizational structures, and the behavior of agents. Section 3.3 already introduced the agent metamodel and parts of the behavior metamodel. In the following, the goal, behavior, and collaboration metamodels are presented.

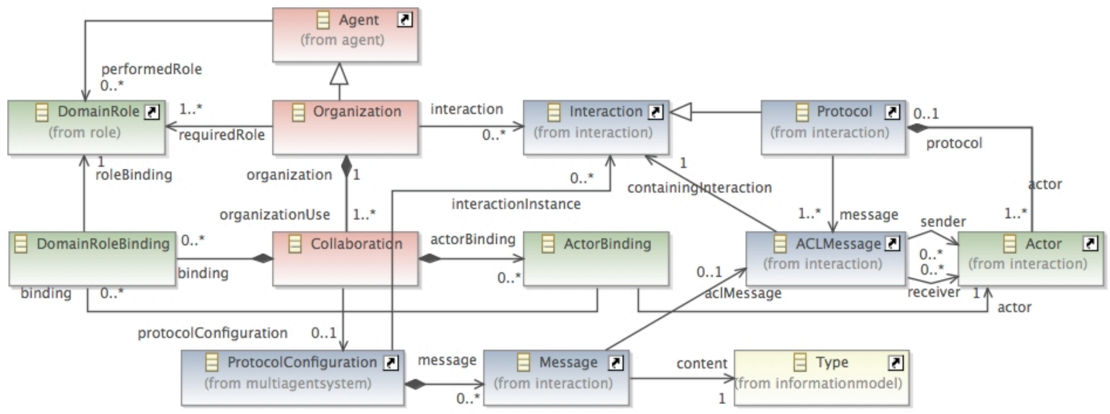


Figure 4.4: The PIM4AGENTS collaboration metamodel.

Agent and Organization Aspect

Figure 4.4 depicts the collaboration metamodel of PIM4AGENTS. The main purpose of `Organizations` in PIM4AGENTS is the definition of the overall structure of a MAS. However, `Organizations` can also be agentified – meaning they perform `DomainRoles` and execute `Behaviors` like `Agents`. The role of agentified `Organizations` in PIM4AGENTS has been discussed by Madrigal-Mora et al. [2008]. As part of this dissertation, `Organizations` are only considered as pure design artifacts for defining the overall structure of a SUC.

The microscopic layer defines the internal role bindings of an `Organization` in terms of `Collaborations`. A `Collaboration` uses `DomainRoleBindings` to bind a `DomainRole` of an `Organization` to an `Actor` (interaction role) of an utilized `Interaction`. This is necessary since `Interactions` are self-contained design artifacts that are independent of a concrete use case. A `Collaboration` in combination with a `ProtocolConfiguration` defines how an interaction protocol is utilized. Similar to the `DomainRoleBinding`, an `ActorBinding` binds an `Actor` to a `Collaboration`. The combination of `DomainRoleBindings` and `ActorBindings` enables an engineer to bind the same `DomainRole` to multiple `Actors` (e.g. of multiple `Interactions`) of the same `Collaboration`. For example, an agent might perform the requester role of a request response protocol and at the same time the participant role in a *Contract Net Protocol* (CNP) [FIPA, 2002c]. This mechanism also enables nesting of protocols. Finally, a `ProtocolConfiguration` defines concrete message and content types for a certain interaction protocol.

Goal Aspect

Figure 4.5 depicts the PIM4AGENTS event and goal metamodel. A `Signal` is an internal event of an agent. A `ConcreteGoal` inherits from `Event` and has

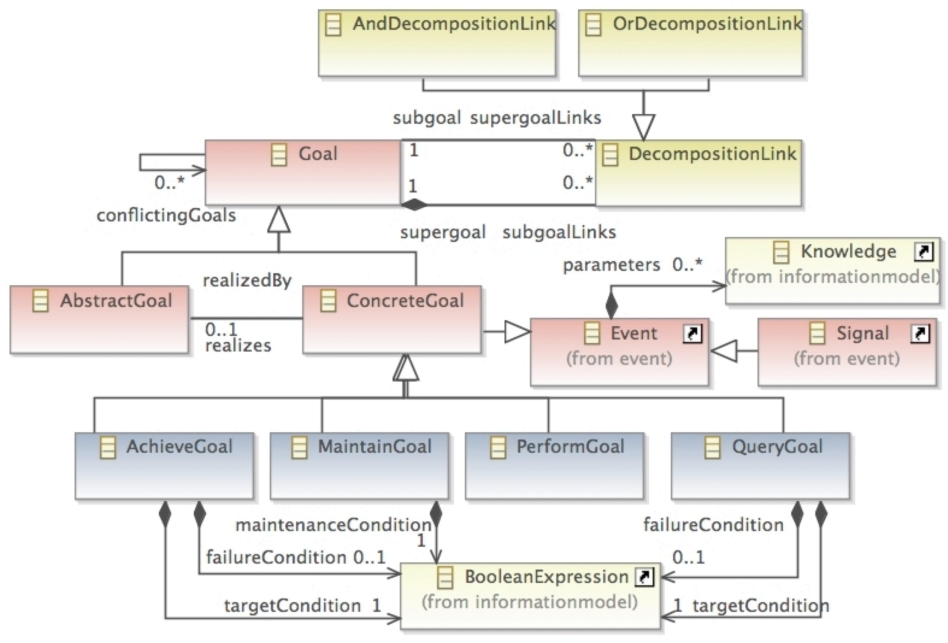


Figure 4.5: The Pim4Agents event and goal metamodel.

Knowledges as parameters. Relationships between Goals can be modeled using the conflictingGoals and subgoal relationships. The subgoal relationship can be either an AndDecompositionLink or an OrDecompositionLink. There exist four different goal types: AchieveGoal (achieve a target condition), MaintainGoal (maintain a state), PerformGoal (perform some action), and QueryGoal (provide an answer). The goal types make use of the BooleanExpression concept for defining the target, failure, and maintain conditions.

Behavior Aspect

As part of the Bochica extension interface specification, the basic infrastructure for behavior definitions was already introduced in Section 3.3.1. The Pim4Agents metamodel provides one realization of the abstract Behavior concept, called Plan. The Activities of a Plan are connected by ControlFlows and InformationFlows. Figure 4.6 depicts the structured activity metamodel. A StructuredActivity is an Activity that contains nested sub-Activities (see Figure 4.6). A Loop is used to (i) iterate over a set of values or (ii) to iterate until a BooleanExpression becomes false. The Parallel concept is used to model n parallel branches. Each branch b_i consists of a sequence of Activities that are connected by Control-Flows. Likewise, a Decision consists of multiple branches that are protected by a BooleanExpression as precondition (if-then-else). Depending on the pre-condition, the branches are activated. ParallelLoops are used for sending and

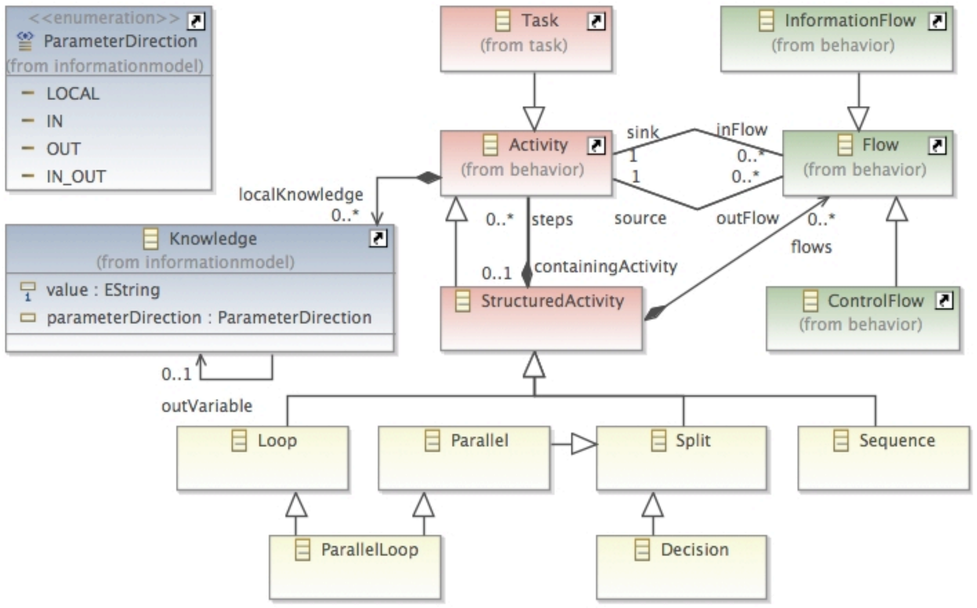


Figure 4.6: The Pim4Agents structured activity metamodel.

receiving `Messages` to/from a set of agent instances (represented by an `Actor` of a `Protocol`). The `ParallelLoop` performs the contained `Activities` for each role-filler of an `Actor` in parallel. A set of `Behaviors` and `Knowledges` with a close relationship can be aggregated by a `Capability` to a functional unit. Similar to the `PlanUse` concept introduced in Section 3.3, a `CapabilityUse` makes an agent's `Knowledges` available within a `Capability` using `KnowledgeBindings`.

Figure 4.7 depicts the Pim4Agents task metamodel. A `Task` is an atomic `Activity` that is not further decomposed. Every `StructuredActivity` has exactly one `Begin` and one `End` task. The `Begin` task has one outgoing `ControlFlow` and the `End` task one incoming `ControlFlow`. It is not allowed to model loops or branches using `ControlFlows` – this has to be done using `StructuredActivities`. Goals are posted by an agent using the `AssumeGoal` task. The `DelegateGoal` task is used to delegate goals to `DomainRoles` (e.g. inside an `Organization`). The `Wait` task makes the agent wait for a timeout. An `InternalTasks` is a placeholder for business logic that is not further detailed at the modeling level (e.g. an algorithm). Interaction protocols are instantiated using the concept `InitiateProtocol`. The `InitiateProtocol` task refers to a `ProtocolConfiguration` and takes the role-fillers of the `Actors` as parameters. `Messages` are exchanged using the `Send` and `Receive` tasks. The `InvokeWS` and `ReceiveWS` tasks are used to orchestrate Web services. The details are discussed in Xiaoqui Cao's master thesis [Cao, 2011]. Synchronous Web service calls use the `InvokeWS` task, whereas asynchronous calls use the additional `ReceiveWS` tasks to wait for the invocation result.

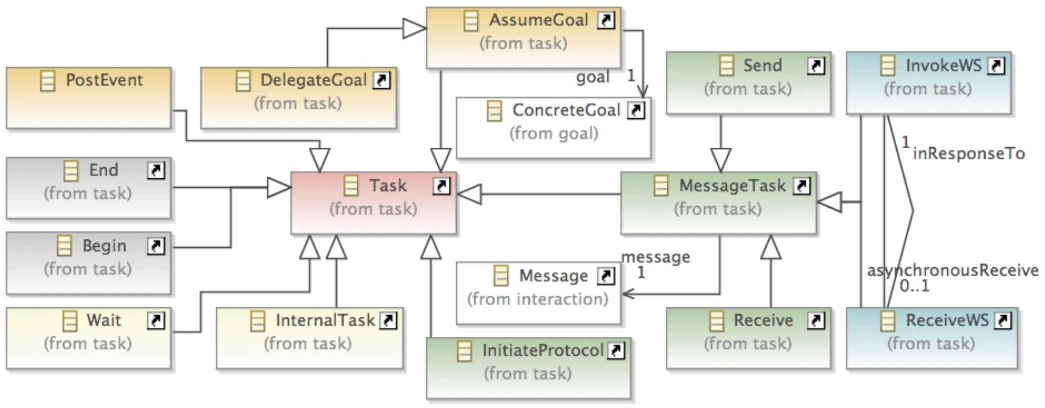


Figure 4.7: This figure depicts the Pim4Agents task metamodel.

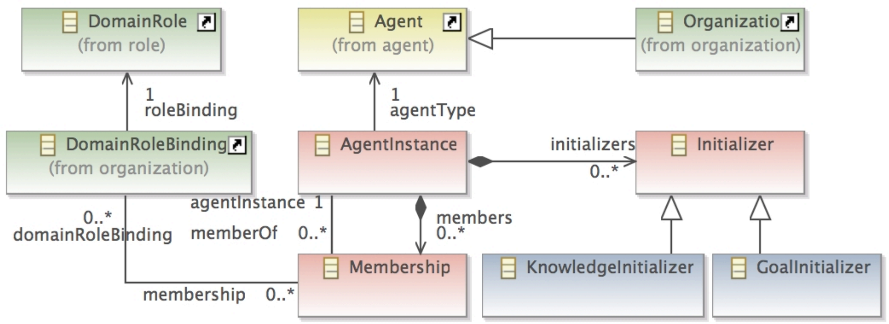


Figure 4.8: This figure depicts the Pim4Agents metamodel for the deployment layer. Green and yellow concepts are from the type layer, whereas red and blue ones represent instances of those types.

4.1.3 Deployment Metamodel

The macro- and microscopic layers specify the entities of a SUC on the type level. Deployment configurations are defined by the concepts of the deployment layer (see Figure 4.8). An `AgentInstance` represents an instance of an `Agent` or `Organization`. The `Membership` concept is used to specify the `DomainRole` that is performed by an `AgentInstance` in an `Organization`. Of course, the bindings have to meet the definitions of the according `Agent` and `Organization` types. Those restrictions are enforced using OCL invariants. Finally, the initial set of beliefs and goals of an `AgentInstance` is configured using the `Initializer` concept. For example, the `KnowledgeInitializer` and `GoalInitializer` concepts are used to set the initial values of `Knowledges` and to configure the initial `ConcreteGoals`.

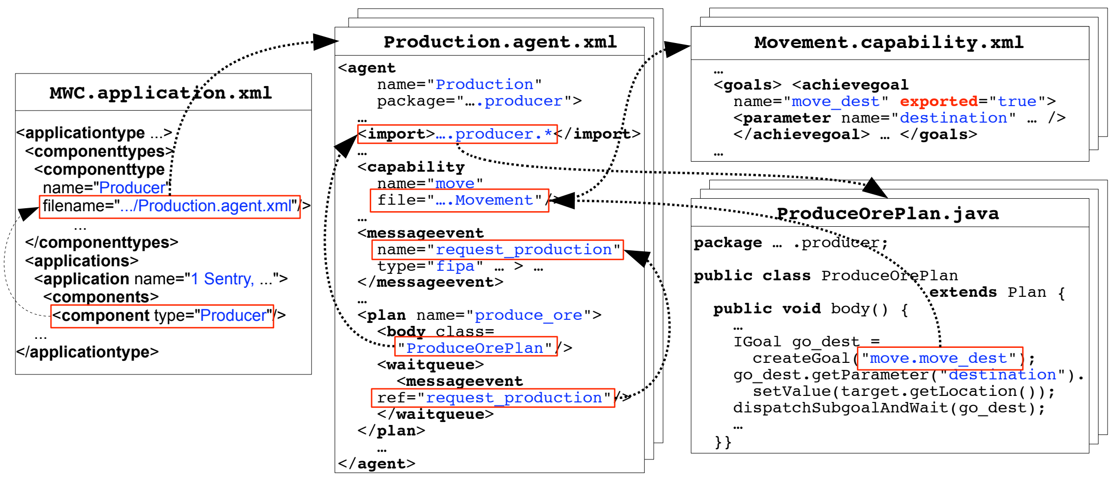
4.2 Platform-specific Metamodel for Jadex

As introduced in Section 2.2.1, Jadex is a BDI agent execution platform. In order to enable Jadex for MDSD according to MDA, a platform-specific metamodel for Jadex is required. The Jadex PSM is necessary to make the Jadex concepts and resources accessible for MDA tools. A Jadex application consists of a set of XML-based configuration files and Java-based behaviors. In order to better understand where the pitfalls for the creation of a Jadex PSM are, Figure 4.9 depicts a typical Jadex application (a part of the official Jadex *Mars World Classic* application). The arrows visualize the inter-dependencies between the XML and Java files. For example, the application XML file on the left hand side declares an application (name "`1 Sentri, ...`") with one component (agent instance) of type Producer. The component type declaration of Producer is resolved to the `Production.agent.xml` file. The production agent XML file imports the ProduceOrePlan Java class. The Java behavior is used as body of the produce_ore plan declaration. Finally, one can see how the ProduceOrePlan posts the move_dest goal. The move_dest goal is declared by the Movement capability and imported to the production agent XML file. It is important to note that the references are plain strings - meaning, there exists no support for resolving those strings automatically. To process those files, one has to resolve the references manually (e.g. by analyzing the imports). The remainder of this section introduces the Jadex project metamodel (Section 4.2.1), the application metamodel (Section 4.2.2), the BDI metamodel (Section 4.2.3), and the behavior metamodel (Section 4.2.4). Together, they make up the Jadex PSM[58].

4.2.1 Project Metamodel.

A PSM for Jadex has to cover the XML and Java-based resource of a Jadex application. This requires metamodels for the XML and Java-based platform artifacts. Jadex provides XML Schema definitions for the XML-based platform artifacts. This encompasses the definition of Jadex applications[59] and BDI[60] resources (e.g. agent and capability definitions). The XML Schema files build the basis for the Jadex application and BDI metamodels. EMF provides the possibility to derive Ecore metamodels from XML Schema files. This has the advantage that the Ecore-based metamodel corresponds one-to-one to the XML Schema files. Moreover, EMF provides automatic de/serialization from/to XML. The Java

[58]The Jadex PSM is based on Jadex version 2.0 RC6.
[59]`http://jadex.sourceforge.net/jadex-application-2.0.xsd`
[60]`http://jadex.sourceforge.net/jadex-bdi-2.0.xsd`



MWC.application.xml

```
<applicationtype ...>
 <componenttypes>
  <componenttype
   name="Producer"
   filename=".../Production.agent.xml"/>
      ...
 </componenttypes>
 <applications>
  <application name="1 Sentry, ...">
   <components>
    <component type="Producer"/>
  ...
</applicationtype>
```

Production.agent.xml

```
<agent
    name="Production"
    package="….producer">
  …
  <import>….producer.*</import>
  …
  <capability
    name="move"
    file="….Movement"/>
  …
  <messageevent
    name="request_production"
    type="fipa" … > …
  </messageevent>
  …
  <plan name="produce_ore">
    <body class=
     "ProduceOrePlan"/>
    <waitqueue>
      <messageevent
    ref="request_production"/>
    </waitqueue>
  </plan>
    …
</agent>
```

Movement.capability.xml

```
  …
  <goals> <achievegoal
    name="move_dest" exported="true">
    <parameter name="destination" … />
    </achievegoal> … </goals>
  …
```

ProduceOrePlan.java

```
package … .producer;

public class ProduceOrePlan
                          extends Plan {
  public void body() {
    …
    IGoal go_dest =
      createGoal("move.move_dest");
    go_dest.getParameter("destination").
      setValue(target.getLocation());
    dispatchSubgoalAndWait(go_dest);
    …
  }}
```

Figure 4.9: This figure depicts a typical Jadex application. It consists of an application file, agent configurations, capabilities, and Java-based plans. The arrows visualize references.

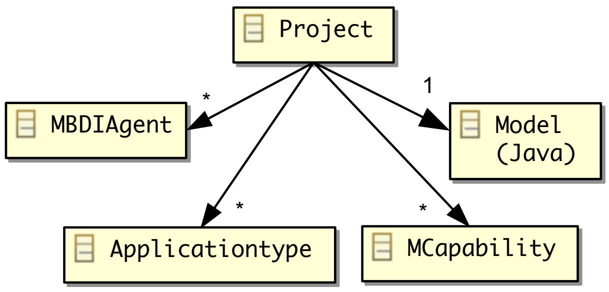


Figure 4.10: This figure depicts the Jadex project metamodel. The `Project` concept is the top-most container and aggregates all resources of a Jadex application.

metamodel of the Eclipse MoDisco[61] project is used as Jadex behavior metamodel. The Jadex project metamodel's task is to aggregate the model artifacts of a Jadex application to one model (see Figure 4.10). The `Project` concept of the Jadex project metamodel imports the root elements of the other metamodels, such as `Applicationtype` (application metamodel), `MBDIAgent` (BDI metamodel), and `Model` (behavior/Java metamodel).

4.2.2 Application Metamodel

The Jadex application metamodel covers concepts for configuring the components of a Jadex application. This encompasses deployment configurations of agent instances as well as the configuration of the Jadex platform. Figure 4.11 depicts an overview of the created Jadex application metamodel. The concept

[61]`http://www.eclipse.org/MoDisco/`

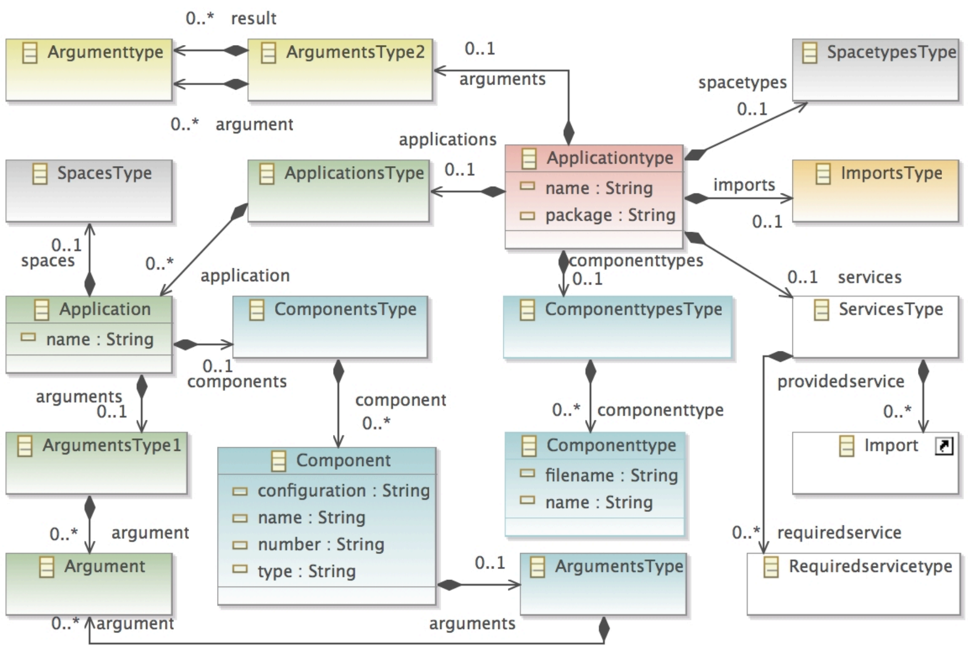


Figure 4.11: This figure depicts the Jadex application metamodel (root concept `Applicationtype`). It encompasses concepts for configuring applications (green, yellow, orange), platform services (white), agents (blue), and the environment (gray).

`Applicationtype` is the root concept of a Jadex application and possesses a `name` and `package` attribute. Resources like Java classes that are not in the current scope (defined by `package`) are imported by the `ImportsType` concept. All components (such as agents) that are part of the application are declared by the concept `Componenttype`. A `Componenttype` has a `filename` attribute. The attribute references a Jadex agent XML file. The concept `Application` defines a single Jadex application configuration (there can be several) and contains a set of `Component` (agent instance) declarations. A `Component` is an instance of a declared `Componenttype` (identified by the `type` attribute). Moreover, the `Component's` `configuration` attribute refers to an `MConfiguration` defined by the agent model (referenced by the `Componenttype`). An `MConfiguration` defines the initial beliefs and goals of a `Component`. Jadex platform services, such as messaging or directory services, are configured using the `ServicesType` concept. The concept `SpacetypesType` is used to import environment definitions based on the Jadex *envspaces*[62] metamodel. For example, it is used to specify data structures that are shared between agents. The envspace metamodel is covered by the Jadex PSM but it is not further detailed in this dissertation.

[62] `http://jadex.sourceforge.net/jadex-envspace-2.0.xsd`

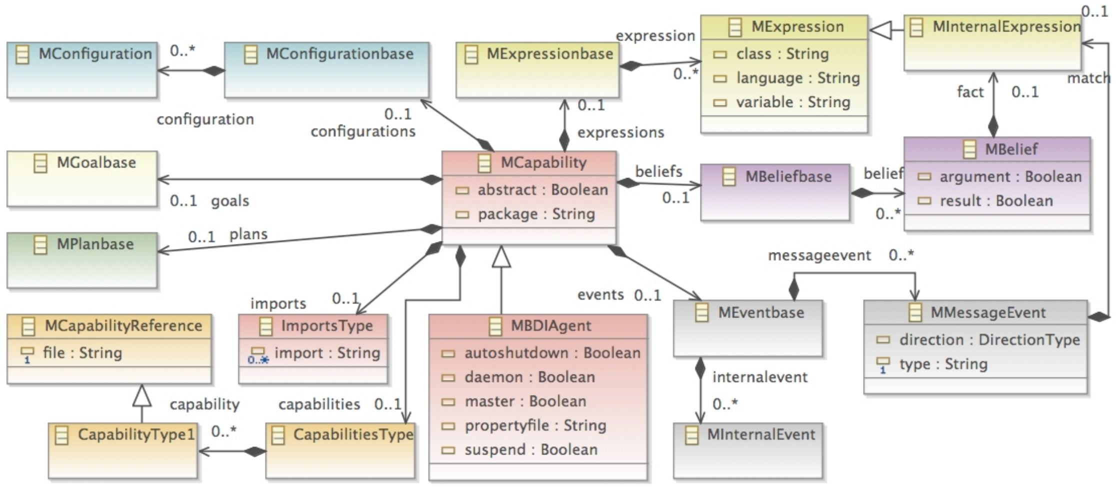


Figure 4.12: This figure depicts the Jadex BDI metamodel. The core concepts are `MCapability` and `MBDIAgent`. The metamodel covers concepts like beliefs (magenta), capability imports (orange), event declarations (gray), goal declarations (yellow), used plans (green), agent instance configurations (blue), and expressions (dark yellow).

4.2.3 BDI Metamodel.

The Jadex BDI metamodel specifies the concepts for defining Jadex BDI agents and capabilities (see Figure 4.12). The central concepts are `MBDIAgent` and `MCapability`. The main difference between Jadex agents and capabilities is that agents possess an own execution thread. Capabilities are self-contained modules and extend an agent with additional beliefs, events, and behaviors. In Jadex, the concept `MBDIAgent` inherits from `MCapability`. An `MCapability` imports resources (e.g. Java classes) by the concept `ImportsType`. The `CapabilitiesType` container imports external `MCapabilities` using the `MCapabilityReference` concept. The `file` attribute references the according XML file. The `MBeliefbase` is a container for belief declarations (concept `MBelief`). The event types of an `MCapability` are defined by the `MEventbase`. For example, `MInternalEvents` are internal events of an agent and `MMessageEvents` are used for the communication between agents. The `MGoalbase` contains goal declarations (discussed later). The behaviors of an `MCapability` are defined by the `MPlanbase` (see Section 4.2.4). `MExpressions` are used in Jadex to initialize `MBeliefs`. Finally, an `MConfiguration` defines the initial events, beliefs, and plans of an agent. It is important to note that beliefs, events, goals, etc. can be exported/imported to/from other `MCapabilities`. For this purpose, each of those concepts has an abstract equivalent. The abstract element serves as placeholder for an entity that is declared elsewhere. For example, the `MBeliefReference` concept is an abstract placeholder

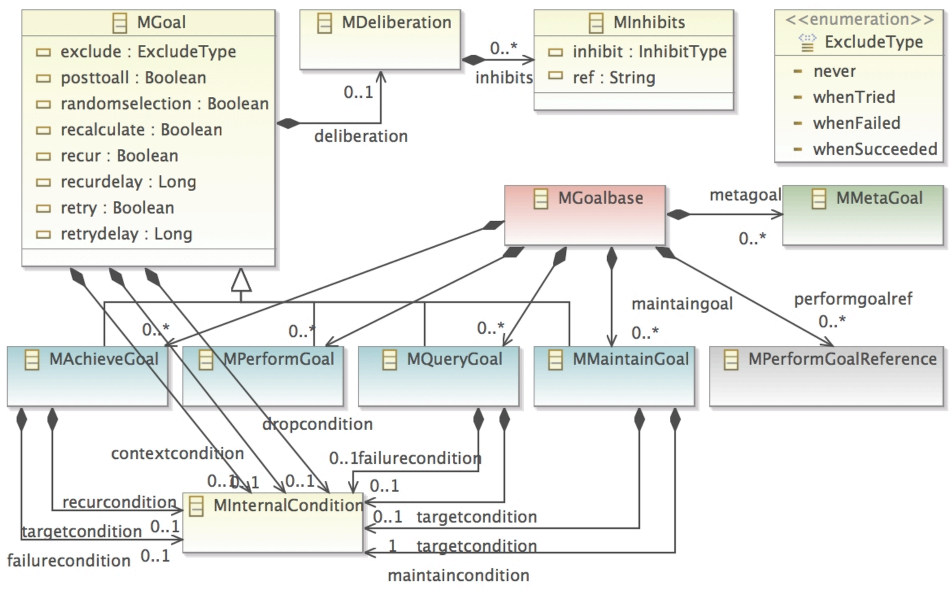


Figure 4.13: This figure depicts the Jadex goal metamodel. There exist four goal types (blue).

for an `MBelief` that is imported from an external `MCapability`. Likewise, there exist `MInternalEventReference` and `MGoalReference` concepts.

Goals. The goal metamodel is part of the Jadex BDI metamodel (see Figure 4.13). The `MGoalbase` contains the goal declarations of an `MCapability`. `MGoal` is the abstract base concept for all goal types. Its attributes control the execution behavior of a goal. For example, the `retry` attribute defines whether the goal is retried after a plan failed. Moreover, the `MInhibits` concept is used to specify that one goal suppresses the execution of another one. There exist four goal types in Jadex: `MAchieveGoal` (achieve some goal state), `MPerformGoal` (perform an action), `MMaintainGoal` (maintain a state), and `MQueryGoal` (provide an answer). The `MInternalCondition` concept is used to define the target, maintain, and failure conditions of goals. Metalevel reasoning is supported by the `MMetaGoal` concept. Goals can be imported and exported by `MCapabilities`. For example, the `MPerformGoalReference` concept is used as placeholder for importing an `MPerformGoal` from an external capability.

4.2.4 Behavior Metamodel

The behavior specification of a Jadex agent consists of two parts: (i) the behavior meta information defined by the Jadex BDI metamodel and (ii) the actual behavior implementation by a Java class. Figure 4.14 depicts the behavior metamodel

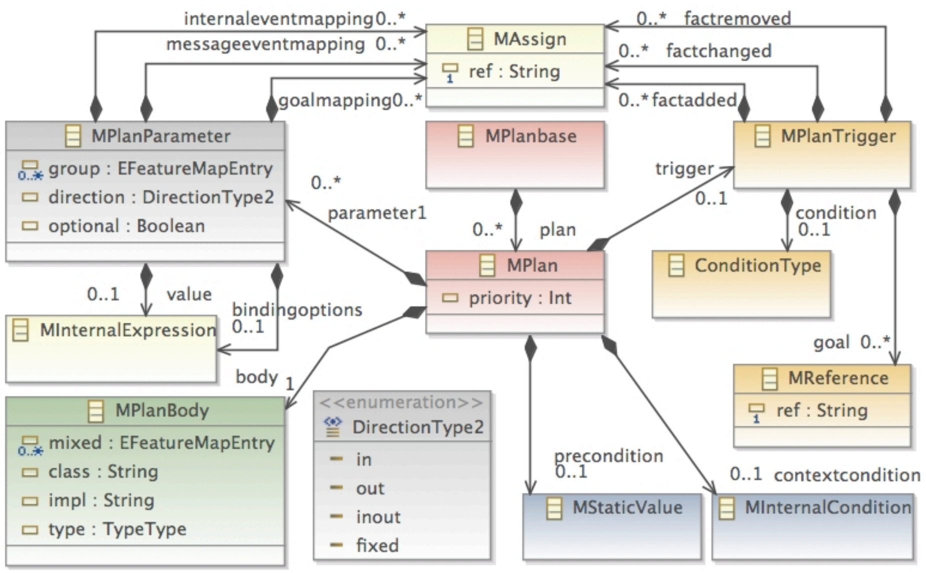


Figure 4.14: This figure depicts the behavior metamodel. It encompasses the reference to the plan body (green), the definition of context and preconditions (blue), the declaration of the triggering event (orange), and a plan's parameters (gray).

as defined by the Jadex BDI metamodel. The `MPlanbase` concept contains the behavior declarations of an `MCapability`. An `MPlan` defines an `MPlanBody`. The `MPlanBody` refers to a Java-based Jadex plan implementation (`impl` attribute). The `MPlanTrigger` concept specifies the triggering event that activates the `MPlan`. This can be an event, goal, or belief change. An event or goal type is referenced by the `ref` attribute of the `MReference` concept. The `MAssign` concept is used to define a belief change as trigger. An `MPlan` can also declare `MPlanParameters`. An `MPlanParameter` is a parameter of a plan and can be initialized with beliefs or the triggering event's parameters (referenced by the `MAssign` concept). Moreover, an `MInternalExpression` can be used to define a set of *binding options* (possible values) for an `MPlanParameter`. Finally, the precondition of an `MPlan` is defined by the `MStaticValue` concept and the context condition by the `MInternalCondition` concept. It is important to note that context conditions in Jadex are invariants for the execution of a plan. In Jack, context conditions are used to compute the plan candidates (similar to the binding options in Jadex). Thus, the semantics is different.

Jadex behaviors are implemented as Java classes. Because the creation of a Java metamodel is a big endeavour on its own, we rely on the Java metamodel and tool support provided by the Eclipse MoDisco project. The top-most container

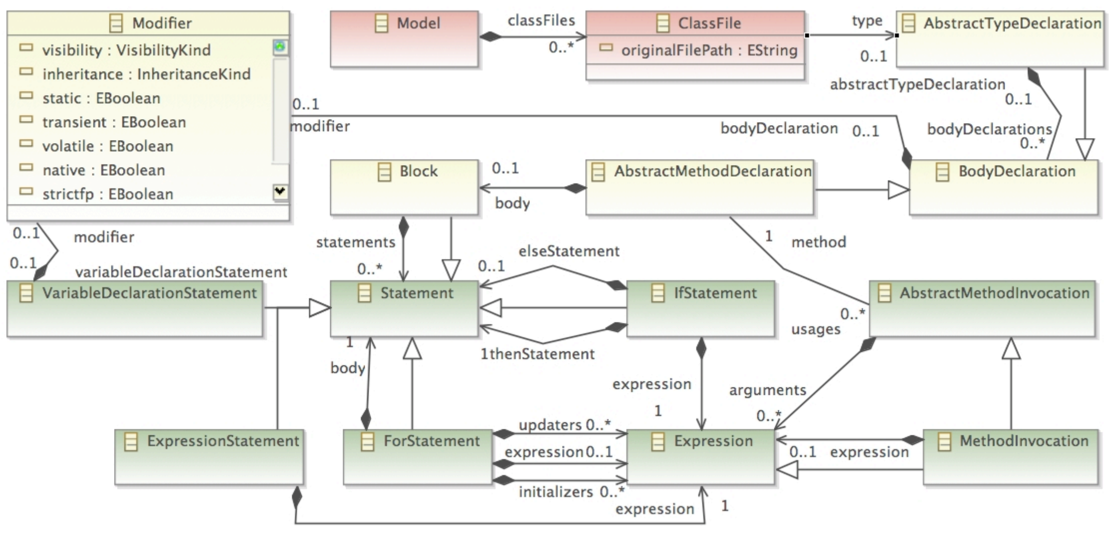


Figure 4.15: This figure depicts a part of the Java metamodel which is used as Jadex behavior metamodel. The root concept is `Model`. The metamodel defines different kinds of statements (green).

for a set of Java files is the concept `Model` (see Figure 4.15). A `ClassFile` has an `originalFilePath` which points to the location of the original Java file. Furthermore, its type is specified by an `AbstractTypeDeclaration`. An `AbstractTypeDeclaration` contains a set of `AbstractMethodDeclarations` (methods of that class). The body of a method declaration is specified by the (code-) `Block` concept. It contains a sequence of `Statements`. There are different kinds of `Statements`. For example, a `ForStatement` has a (condition-) `Expression` (relation `expression`), an initializer `Expression` (relation `initializers`), and an iterator `Expression` (`updaters` relation). A `MethodInvocation` is an `Expression` that invokes a method. `VariableDeclarationStatements` are used to declare variables and have a `Modifier` (e.g. visibility or whether a variable is static).

It is important to note that the Java metamodel does not contain Jadex-specific concepts (e.g. a concept that represents a Jadex API call for sending a message). In order to make use of the Java metamodel for Jadex, one has to take the Jadex API into consideration. For example, an instance of type `AbstractTypeDeclaration` that extends the `jadex.bdi.runtime.Plan` class of the Jadex API represents a Jadex behavior. Other `AbstractTypeDeclarations` are no Jadex plan declarations. A further example is an instance of type `MethodInvocation` that invokes the `jadex.bdi.runtime.Plan.dispatchSubgoal()` method of the Jadex API for posting a sub-goal from inside a plan. Table 4.1 depicts an overview of important instantiations of the Java metamodel for the Jadex API. Those types will be used by the reverse transformation in Chapter 6.

Java MM Concept	**Property**	**Value**	**Description**
AbstractTypeDeclaration	superclass	jadex.bdi.runtime.Plan	Jadex behavior implementation
AbstractTypeDeclaration	–	–	Data type declaration
MethodDeclaration	name	jadex.bdi.runtime.Plan.body()	Main method of a Jadex behavior
MethodInvocation	name	jadex.bdi.runtime.Plan.dispatchSubgoal[AndWait]()	Post sub-goal (async./sync.)
MethodInvocation	name	jadex.bdi.runtime.Plan.sendMessage[AndWait]()	Send message (async./sync.)
MethodInvocation	name	jadex.bdi.runtime.Plan.waitForMessageEvent()	Receive message
MethodInvocation	name	jadex.bdi.runtime.Plan.waitFor()	Delay execution

Table 4.1: This table depicts the relation between the Java metamodel and the Jadex API. For example, an instance of a Java `MethodInvocation` that invokes the `jadex.bdi.runtime.Plan.sendMessage()` method is a call to the Jadex API for sending a message asynchronously.

4.3 Summary

Metamodels build the foundation for MDSD. In order to build-up the MDA stack for the Jadex platform, this section introduced a PSM for Jadex. Moreover, the Pim4Agents metamodel has been introduced. It defines the abstract syntax of Dsml4Mas which is used as Bochica core DSL. Existing metamodels and tools were reused where possible. By comparing the Pim4Agents and Jadex metamodels one can see that the Jadex PSM is more fine-grained than Pim4Agents. For example, it defines platform services, fine-grained goal execution semantics, and Java-based behaviors. On the other side, Pim4Agents offers high-level model artifacts such as organizational structures and interaction protocols. The presented metamodels build the foundation for the conceptual mappings presented in Chapters 5 and 6.

Chapter 5

Model-driven Forward Engineering of BDI Agents

Based on the transformation architecture introduced in Section 3.4, this Chapter defines a base transformation for BDI agents founded on the Jadex platform. For this purpose, conceptual mappings between the Bochica core DSL and the Jadex PSM are defined. Although Jadex as well as Bochica make use of typical BDI concepts, there exists a conceptual gap which has to be overcome. The Jadex platform provides low-level infrastructure for implementing BDI agent systems (e.g. platform services, capability infrastructure, and Java-based behaviors) but it has no explicit representation of high-level artifacts like interaction protocols or organizational structures. Moreover, the platform leaves room for different "flavours" for implementing a MAS. How an agent engineer makes use of this toolbox depends on his experience with AOSE and Jadex. The specification of a base transformation implies to anticipate several design decisions by using design patterns. Design patterns help to (i) guarantee a certain quality independent of the developer's experience, and (ii) ease the understanding of the generated artifacts. The generated code should be (i) modular, (ii) extensible, (iii) complete, and (iv) scalable. Figure 5.1 depicts an overview of the proposed transformation architecture for Jadex. The base transformation consists of the five modules (i) application, (ii) BDI, (iii) interaction, (iv) behavior, and (v) data model. The application, BDI, and interaction modules are mapped to the Jadex PSM and serialized to XML. Bahaviors are directly mapped by a model-to-text transformation to Java code. The decision to map behaviors directly to code is based on our experience with model transformations to Jack and Jade. Java-based behaviors are very fine-grained so that writing model-to-model transformations becomes tedious. As Bochica gets extended by a custom extension model for an execution environment or application domain, the additional concepts are covered by an extension transformation. Chapter 7

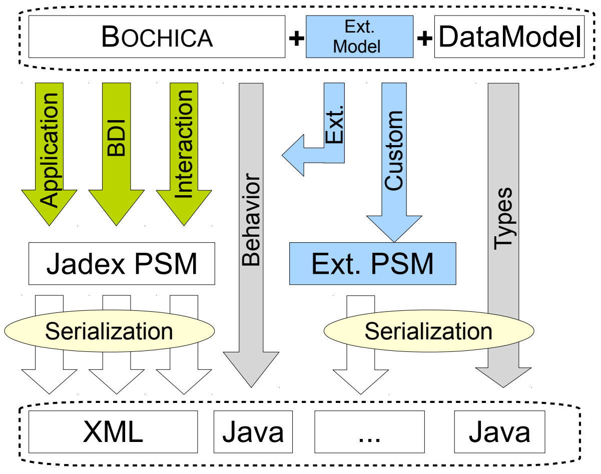


Figure 5.1: This figure depicts the forward transformation architecture for Jadex. It consists of model-to-model mappings (green and blue) and model-to-text mappings (gray and white). The green and gray parts represent the base transformation and the blue parts an extension transformation.

introduces an extension transformation for agents in semantically-enhanced virtual worlds which extends the Jadex base transformation. The remainder of this chapter presents the mapping rules and code generation patterns for the application (Section 5.1), BDI (Section 5.2), behavior (Section 5.3), interaction (Section 5.4), and data model (Section 5.5) aspects. Following abbreviations for the metamodels are used: PIM4AGENTS (P4A), Jadex BDI (BDI), and Jadex Application (APP). Moreover, the prefix `CODE` indicates that the target of a mapping rule is a Java code template. `CODE` is used for model-to-text mappings (Java concepts). Each mapping rule consists of (i) an identifier, (ii) the signature (head) of the rule (source and target concepts), (iii) an optional OCL-based precondition, and (iv) the body of the rule. Additionally, tables, figures, and listings are used to provide an overview of the interdependencies of the rules. The variable *self* is used by OCL conditions to refer to the object the rule is being applied to. Some rules are separated into sub-rules (e.g. MR-11a/b, etc.) in order to indicate that a concept is mapped to two target concepts with a close relationship (e.g. passing parameters between an agent and a capability).

5.1 Application Mappings

The following conceptual mapping rules project PIM4AGENTS concepts onto concepts of the Jadex application metamodel. This covers aspects like the deployment configuration and platform services.

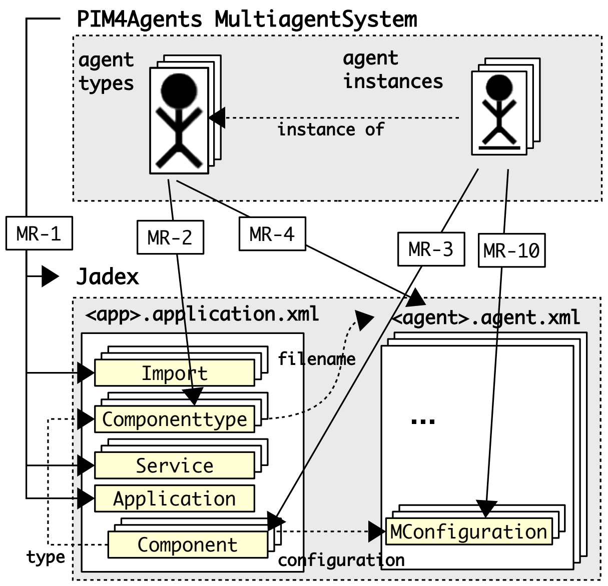


Figure 5.2: This figure depicts the relationship between agents of a Pim4Agents model and the corresponding artifacts in Jadex. Every agent type is transformed by MR-4 to an according Jadex agent model. The agent model is imported by the Jadex application model (MR-2). Likewise, an instance of a certain agent type is declared in the Jadex application model (MR-3) and configured in the Jadex agent model (MR-10).

MR-1: *P4A* : *MultiagentSystem* → *APP* : *Applicationtype*

Body: The concept `MultiagentSystem` is the root concept of a Pim4Agents model. It is mapped to a Jadex application model (root concept `Application-type`). Figure 5.2 provides an overview of the structural interdependencies of the related mapping rules. Table 5.1 depicts the details of this mapping rule.

MR-2: *P4A* : *Agent* → *APP* : *Componenttype*

Pre: $self.oclIsTypeOf(P4A:Agent)

Body: A Jadex `Applicationtype` imports the `Componenttypes` (agent types) which are used by the application. The precondition ensures that no `Organiza-tions` are mapped by this rule. Every Pim4Agents `Agent` of the source model is transformed to a Jadex `Componenttype` declaration in the application model. The `Componenttype` has a `filename` attribute that references the according agent model (MR-4; see Figure 5.2).

MR-1: *P4A* : *MultiagentSystem* → *APP* : *Applicationtype*		
target	source	MR
name	The name of the Pim4Agents `MultiagentSystem`.	–
imports	The import section adds the required Java packages to the classpath. For example, classes implementing platform services used in the `services` section are imported.	
componenttypes	For each `Agent` of the source model an according Jadex `Componenttype` is declared which references the according Jadex agent model generated by MR-4.	2, 4
services	A Jadex application requires the initialization of platform services, such as (i) a directory service, (ii) a component management service, and (iii) a messaging service. The mapping rule initializes the required Jadex platform services.	–
applications	A `MultiagentSystem` is mapped to a Jadex `Application`. All `AgentInstances` of the source model are mapped to `Components` of this `Application`. The application has the name of the Pim4Agents `MultiagentSystem`.	3

Table 5.1: Details of the application mapping (MR-1). The first column depicts the target attributes of the `Applicationtype`, the second column summarizes the mapping, and the third column shows the related mapping rules.

MR-3: *P4A* : *AgentInstance* → *APP* : *Component*

Pre: $self.agentType.oclIsTypeOf(P4A:Agent)

Body: A Pim4Agents `AgentInstance` is mapped to a Jadex `Component`. The precondition ensures that no organization instances are mapped by this rule. The initializers of a Pim4Agents `AgentInstance` (e.g. `GoalInitializer` or `BeliefInitializer`) are mapped to an `MConfiguration` in the agent model (see MR-10). The `MConfiguration` is referenced by the `configuration` attribute of the `Component`. Moreover, the `type` attribute references a `Componenttype` (agent type; see MR-2).

5.2 BDI Mappings

The mapping rules related to the BDI aspect cover the internals of agents like beliefs, events, or goals. As mentioned in Chapter 4, the main difference between agents and capabilities in Jadex is that agents possess an own execution thread.

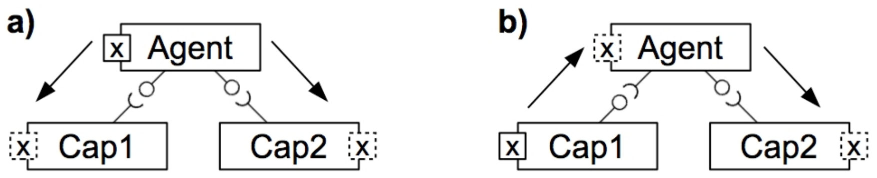


Figure 5.3: This figure depicts two examples of how scoping can be realized in Jadex. a) shows a top-down assignment of an element x of an agent to some used capabilities. b) shows how a capability defines an element x that is imported by the agent and assigned to another capability. X can be beliefs, events, or goals.

As the Jadex `MBDIAgent` inherits from `MCapability`, the mapping rules for agents and capabilities are very similar. Before we go into detail, it is important to note that Jadex offers different mechanisms for defining the scope between an agent and its capabilities (see Figure 5.3). Case a) makes an element x available in `Cap1` and `Cap2` by assigning it top-down, whereas b) imports x form `Cap1` and assigns it to a placeholder in `Cap2`. The design decision is left open to the developer. However, goals and events in Pim4Agents have a global scope inside the agent and are not explicitly assigned by the developer (in contrast to Jadex). Instead, the used events and goals are derived from an agent's plan templates. The transformation moves the goal and event declarations to the outmost scope (e.g. the agent) and assigns the entities top-down to abstract placeholders in the capabilities that make use of it (like case a). Thus, the designer does not have to think about event and goal scoping in Pim4Agents. The assignment of beliefs is solved in Pim4Agents similar to Jadex by the `PlanUse` and `CapabilityUse` concepts.

MR-4: *P4A* : *Agent* → *BDI* : *MBDIAgent*

Body: The Pim4Agents `Agent` concepts is projected onto the Jadex `MBDIAgent` concept. Since Bochica and Jadex are both based on the BDI model of agency, the basic structure of agents is similar. Table 5.2 summarizes the details of this mapping rule.

MR-5: *P4A* : *Capability* → *BDI* : *MCapability*

Body: A Pim4Agents `Capability` is transformed to an `MCapability` in Jadex. Since agents and capabilities in Jadex have the same properties, the capability mapping rule is very similar to MR-4 (see Table 5.2). The main difference is that the `MCapability` mapping uses abstract placeholder concepts for beliefs, events, and goals. For example, MR-8a is replaced by MR-8b, and 38a by 38b. MR-9a is

MR-4: *P4A* : *Agent* → *BDI* : *MBDIAgent*		
target	source	MR
name	The name of the PIM4AGENTS `Agent`.	–
imports	The import section adds Java classes to the agent's classpath. This is done for data types of `MBeliefs` and `MPlanParameters`.	–
beliefs	Each local `Knowledge` that is declared by the PIM4AGENTS `Agent` is mapped to an `MBelief`. Section 5.3 provides details on how beliefs are passed as parameters to plans and capabilities.	38a
capabilities	Each `CapabilityUse` of the PIM4AGENTS `Agent` is mapped to a Jadex `MCapabilityReference`. The `file` attribute refers to the according capability model. A PIM4AGENTS `Plan` is also mapped to an `MCapability` (see MR-11). For every `PlanUse` of the agent, an `MCapabilityReference` is generated. It imports the according `MCapability`.	6, 7
goals	Every PIM4AGENTS `ConcreteGoal` that is used by the agent is mapped to a Jadex `MGoal` and added to the agent's `MGoalbase`. `MGoals` are passed to capabilities using the `assignto` relationship (see Section 5.3).	8a
events	Each `Signal` of a PIM4AGENTS `Agent` is mapped to an `MInternalEvent` and added to the `MBDIAgent's MEventbase`.	9a
configurations	Every `AgentInstance` of the PIM4AGENTS `Agent` is mapped to an `MConfiguration` (see Figure 5.2). The `MConfiguration` is referenced by the application mappings (see MR-3). The `MConfiguration's` name is set to the name of the `AgentInstance`.	10

Table 5.2: Details of the agent mapping (MR-4).

replaced by MR-9b for `Knowledges` with direction `IN` or `INOUT` (the `Knowledges` are mapped to placeholders). `Knowledges` with direction `LOCAL` or `OUT` are mapped by MR-9a.

MR-6: *P4A* : *CapabilityUse* → *BDI* : *MCapabilityReference*

Body: A `CapabilityUse` assigns a `Knowledge` of a PIM4AGENTS `Agent` or `Capability` to an imported `Capability`. In Jadex, it is mapped to an `MCapabilityReference` that imports the referenced capability model into an agent or capability model using the `file` attribute. The scoping of beliefs, events, and goals is done by MR-8, MR-9, and MR-38.

MR-7: *P4A* : *PlanUse* → *BDI* : *MCapabilityReference*

Body: Similar to a `CapabilityUse`, a `PlanUse` assigns beliefs of an agent or capability to an used `Plan`. As Pim4Agents `Plans` are mapped to `MCapabilities` (see Section 5.3), the corresponding `MCapability` has to be imported to an agent or capability that makes use of the plan. This is done by the `file` attribute of the `MCapabilityReference`. The scoping of beliefs, events, and goals is done by MR-8, MR-9, and MR-38.

MR-8a/b: *P4A* : *ConcreteGoal* → *BDI* : *MGoal*[*Reference*]

Body: Pim4Agents and Jadex use the same four goal types: perform goal, achieve goal, maintain goal, and query goal. The concepts are mapped accordingly. Goals in Pim4Agents have a global scope inside an agent. In contrast to Jadex, the developer does not have to do explicit scoping between agents and capabilities. Instead, the used goals are automatically derived from the agent's plan templates. The transformation moves the goal declaration to the outmost scope (e.g. the agent) and then assigns the goal top-down to abstract placeholders (concept `MGoalReference`) in the used (plan-) capabilities using the `assignto` relationship of `MGoal`. MR-8a maps a `ConcreteGoal` to an `MGoal` declaration, whereas MR-8b creates the abstract goal placeholder in a sub-capability or plan (see Section 5.3). A Pim4Agents `AchieveGoal's` target condition (concept `BooleanExpression`) is mapped to a target condition (concept `MInternalCondition`), and the `MaintainGoal's` maintain condition (concept `BooleanExpression`) to a maintain condition (concept `MInternalCondition`; see MR-39b). The `conflictingGoals` relationship is mapped to the `inhibits` relationship in Jadex.

MR-9a/b: *P4A* : *Signal* → *BDI* : *MInternalEvent*[*Reference*]

Body: This mapping rule maps a Pim4Agents `Signal` to an `MInternalEvent` in Jadex. The scoping is solved in the same way as for goals in MR-8a/b. MR-9a maps the `Signal` to an `MInternalEvent` declaration, whereas MR-9b maps it to an abstract placeholder. The `assignto` relation is used by an `MInternalEvent` to assign it to a sub-capability.

MR-10: *P4A* : *AgentInstance* → *BDI* : *MConfiguration*

Body: This mapping rule is the counterpart for MR-3 in the application section (see Figure 5.2). It is responsible to setup the initial goals, events, and beliefs of a certain `AgentInstance`. Therefore, the Pim4Agents concepts `GoalInitializer` and `KnowledgeInitializer` are mapped to `MConfigGoal` and `MConfigBelief` of the `MConfiguration`. The `ref` relation is used to refer to the according goal type or belief declaration. The `MConfiguration` has the name of the `AgentInstance`.

5.3 Behavior Mappings

The mapping of a Pim4Agents `Plan` to Jadex consists of three parts (see Figure 5.4): (i) the mapping of the plan to a plan capability (MR-11), (ii) the plan's head (meta data) inside the plan capability (MR-12), and (iii) the actual Java-based plan template (MR-13). The plan is mapped to a capability since Pim4Agents plans might contain complex structures that are split-up during model transformation into several parts. The capability groups those parts into one self-contained entity. Moreover, the mapping to the plan capability is responsible for configuring the plan's interface to the outer scope (e.g. placeholders for beliefs, events, and goals). The business logic (encapsulated by `InternalTasks`) gets separated during the mapping process from the plan body into factory classes (MR-21a/b). The factories can be customized at code level. This avoids customizations of the generated code. The mapping rules presented in the remainder of this section have many structural interdependencies. Figure 5.4 depicts a high-level overview of the three target artifacts. Section 5.3.1 introduces the `Task` mappings, and Section 5.3.2 the mappings for `StructuredActivities`.

Pre- and Context Conditions. There exists an important difference between Bochica and Jadex in how the agent engineer (and thus the model transformation) has to initialize the parameters of a plan. Usually, a plan has a set of plan parameters $P = \{p_1, \ldots, p_n\}$. In Jadex, for each $p_i \in P$ a set of values (variable bindings) can be assigned. In order to generate all possible plan candidates for handling a goal, Jadex computes the cartesian product $p_1 \times \ldots \times p_n$. For each computed tuple an according plan candidate is created. In Bochica, the engineer initializes the plan parameters using a *context condition* (similar to Jack). A context condition contains unbound variables $p_1, \ldots, p_n$ (also called *logicals* in Jack) and is a query to the agent's knowledge base. The computed tuples are used to create the plan candidates (like in Jadex). The difference for the agent engineer is that Jadex expects a separate initialization of each plan parameter, while Bochica uses a single query per plan that is evaluated in the agent's knowledge base. One solution to make this difference transparent for the agent engineer dur-

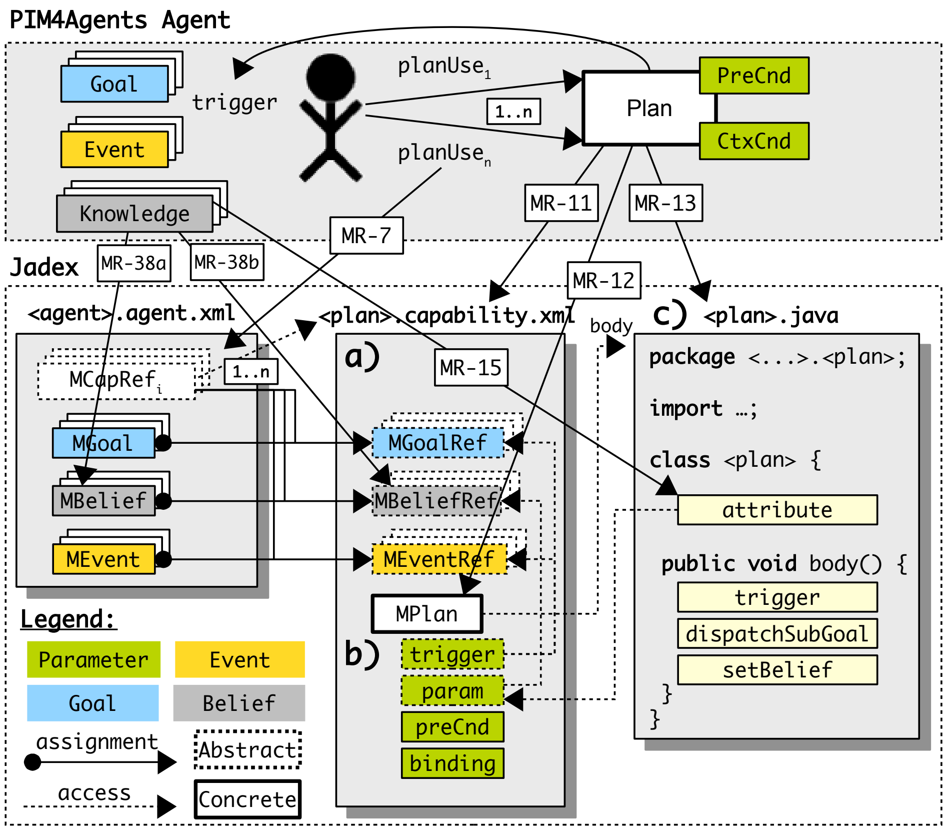


Figure 5.4: This figure depicts an overview of the structural dependencies between the generated artifacts for one PIM4AGENTS `Plan` which is used in n configurations by one agent. For each PIM4AGENTS plan, an according Jadex capability model is generated (MR-11). All goals, beliefs, and events which are declared by the outer scope (here the agent) and accessed by the plan are passed to the plan capability. MR-38a/b and MR-15 exemplify this relationship for `Knowledges`. MR-38a generates the belief declaration and MR-38b the abstract belief placeholder. Finally, the belief is accessed from the plan body to initialize the class attribute (MR-15).

ing model transformation is to wrap a BOCHICA context condition into a single Jadex `MPlanParameter` and generate code that uses the computed tuples for initializing the single plan parameters. However, the used workaround also depends on the utilized knowledge base. The workaround has to be realized as part of an extension transformation that is aware of the used knowledge base. Moreover, the concept of context condition is used in Jadex as an invariant for plans. Whenever a Jadex plan's context condition is violated the according plan instance is dropped. Preconditions in BOCHICA and Jadex have the same semantics.

MR-11: *P4A* : *Plan* → *BDI* : *MCapability*		
target	source	MR
name	The name of the PIM4AGENTS `Plan`.	–
imports	For every data type (Java class) that is used by a Jadex `MBelief` or `MPlanParameter` declaration an import is generated.	–
beliefs	Every PIM4AGENTS `Knowledge` of the source plan which has the direction `IN`, `OUT`, or `INOUT` is mapped to an abstract `MBeliefReference` in the Jadex plan capability model (MR-38b). The belief is initialized by the outer scope using the binding information provided by a PIM4AGENTS `PlanUse`.	38b
plans	The plan capability's `MPlanbase` is initialized with exactly one `MPlan` declaration for the current PIM4AGENTS `Plan`.	12
goals	Every goal that is used inside the PIM4AGENTS `Plan's` body (including the triggering goal) is mapped to an abstract `MGoalReference` declaration in the Jadex plan capability. The `MGoalReference` is initialized by the outer scope (e.g. the agent).	8b
events	Every event which is used inside the PIM4AGENTS plan's body (including the triggering event) is mapped to an abstract `MInternalEventReference` declaration in the Jadex plan capability. It is initialized by the enclosing scope (e.g. the agent).	9b

Table 5.3: Details of the plan capability mapping (MR-11).

MR11: *P4A* : *Plan* → *BDI* : *MCapability*

Body: As already mentioned, a PIM4AGENTS `Plan` is mapped to an `MCapability`. It aggregates all plan-related artifacts and configures the capability's interface to the surrounding scope (e.g. beliefs, events, and goals). Table 5.3 summarizes the details of this mapping rule and Figure 5.4 depicts an overview of the structural dependencies between an agent and the plan capability.

MR12: *P4A* : *Plan* → *BDI* : *MPlan*

Body: This rule declares a plan's head inside the according plan capability model (see Figure 5.4). The meta information encompasses (i) the declaration of the plan trigger (concept `MPlanTrigger`), (ii) the declaration of in/out parameters of the plan (type `MPlanParameter`), and (iii) the precondition and binding options (context condition). Moreover, the `body` attribute (concept `MPlanBody`) points to the Java-based plan implementation. The plan trigger is mapped from the `triggeringEvent` attribute of the PIM4AGENTS `Plan`. `Knowledges` of the

Pim4Agents `Plan` with directions `IN`, `OUT`, and `INOUT` are mapped to `MPlanParameters`. The `MPlanParameters` are initialized by the according abstract `MBeliefReferences` mapped by MR-38b (see Figure 5.4). The precondition of type `BooleanExpression` is mapped to an `MStaticValue` of the `precondition` property of the `MPlan` (MR-39a). The `ContextCondition` of the source plan is mapped by default to one `MPlanParameter` (MR-39b). The parameter's `bindingoptions` relations is initialized by an `MInternalExpression`. The `MInternalExpression` has the value of the source context condition.

MR-13: *P4A* : *Plan* → *CODE* : *PlanClass*

Body: This mapping rule projects a Pim4Agents `Plan` to a Jadex plan class (model-to-text). Listing 5.1 depicts the basic structure of the created Jadex plan template and the cross references to related mapping rules. The Java class inherits from `jadex.bdi.runtime.Plan` (provided by the Jadex API). MR-15 maps the Pim4Agents `Plan's Knowledges` to attributes. The class overloads the `body()` method defined by the Jadex API. The body consists of (i) getting the triggering event (MR-16), (ii) the actual behavior code (MR-18), and (iii) setting the out-variables (MR-17). The `passed()`, `failed()`, and `aborted()` methods of the Jadex API provide the possibility to react to plan abortions (MR-20).

MR-14: *P4A* : *Plan* → *CODE* : *ImportSection*

Body: This mapping rule is responsible for generating the imports required by the Java-based plan class. This encompasses imports for (i) types of the Jadex API (e.g. `IMessageEvent`, `IGoal`), (ii) classes defined by the data model, (iii) native Java types (e.g. `java.util.ArrayList`), and (iv) types from additional libraries used by the mapping rules.

MR-15: *P4A* : *Knowledge* → *CODE* : *Attribute*

Body: A local `Knowledge` of a Pim4Agents `Plan` is mapped to an object attribute of the Jadex plan class (see Listing 5.2). If the current `Knowledge` has the parameter direction IN or INOUT, the attribute is initialized with the value of the `MPlanParameter` mapped by MR-12 (see Figure 5.4). Otherwise, the `Knowledge's value` attribute is used. The variable type mapping is discussed in Section 5.5.

```
package <$basepackage>.<$self.name>; <imports: MR-14>
  public class <$self.name> extends jadex.bdi.runtime.Plan {
  <plan variables: MR-15>
  public <$self.name> () {
    <constructor: MR-20d>
  }
  @Overrides
  public void body() {
    <trigger: MR-16>
    <body: MR-18>
    <out-parameters: MR-17>
  }
  @Overrides
  public passed() {
    <passed: MR-20a>
  }
  @Overrides
  public failed() {
    <failed: MR-20b>
  }
  @Overrides
  public aborted() {
    <aborted: MR-20c>
  }}
```

Listing 5.1: This listing depicts the code template for MR-13 ($self refers to a PIM4AGENTS `Plan`; $basepackage is the location of the Java code). Angle brackets define code generation instructions (e.g. an invocation of a mapping rule).

```
<IF ($self.parameterDirection = P4A:ParameterDirection:LOCAL
   || $self.parameterDirection = P4A:ParameterDirection:OUT)>
  private <$self.type> <$self.name> = <$self.value>;
<ELSE-IF>
  private <$self.type> <$self.name> =
     (<$self.type>)this.getParameter("<$self.name>").getValue();
<END-IF>
```

Listing 5.2: Code template for `Knowledges` of a `Plan` (MR-15; $self refers to a PIM4AGENTS `Knowledge`).

```
<IF($self.triggeringEvent.oclIsTypeOf(P4A:ConcreteGoal)>
 IGoal trigger = (IGoal)this.getReason();
<ELSE-IF ($self.triggeringEvent.oclIsTypeOf(P4A:Signal)>
 IInternalEvent trigger = (IInternalEvent)this.getReason();
<END-IF>
```

Listing 5.3: This listing depicts the code template for MR-16. Depending on the type of the trigger, a local variable called *trigger* is initialized ($self refers to a Pim4Agents `Plan`).

```
this.getParameter("<$self.name>").setValue(<$self.name>);
```

Listing 5.4: This template sets an out parameter of a plan (MR-17; $self refers to a `Knowledge` with direction `OUT` or `INOUT`).

MR-16: *P4A* : *Plan* → *CODE* : *PlanTrigger*

Body: This mapping rule generates the Java code that is required to access the triggering event or goal of a plan in the plan's body (see Listing 5.3). The rule is related to the trigger parameter set in the Jadex plan capability (see MR-12; Figure 5.4).

MR-17: *P4A* : *Knowledge* → *CODE* : *OutParameter*

Pre: $self.parameterDirection = P4A:ParameterDirection:OUT or
$self.parameterDirection = P4A:ParameterDirection:INOUT

Body: At the end of a plan body, this mapping rule updates the value of each `Knowledge` of a Pim4Agents `Plan` with parameter direction `OUT` or `INOUT`. Thus, the outer scope (e.g. an agent's beliefs) gets updated (see Listing 5.4).

MR-18: *P4A* : *Plan* → *CODE* : *PlanBody*

Body: A Pim4Agents `Plan` consists of an ordered sequences of `Activities` $(a_0, \ldots, a_n)$, where a_0 is of type `Begin` and a_n of type `End`. The sequencing is enforced by `ControlFlows` that connect two succeeding `Activities`. Cycles and branches are not allowed. `Activities` can be atomic `Tasks` or `StructuredActivities` with sub-`Activities` (see Section 4.1). MR-19 generates the actual code for the `Activity` sequence.

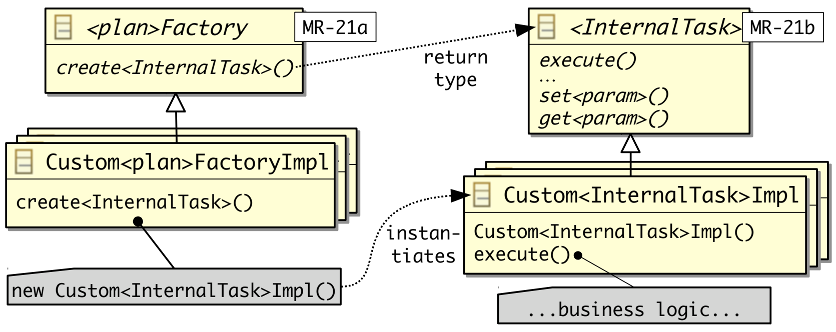


Figure 5.5: This figure depicts the design pattern which is used to generate an `InternalTask` factory for every PIM4AGENTS `Plan`. The factory can be exchanged by providing a custom factory implementation which instantiates custom `InternalTask` implementations. The abstract `InternalTask` class provides get/set methods for every input/output parameter. Moreover, the `execute()` method is used to invoke the business logic.

MR-19: $Sequence(P4A : Activity) \rightarrow CODE : PlanBody$

Body: This mapping rule iterates over a sequence of `Activities` $(a_0, \ldots, a_n)$ and invokes the according mapping rules. Sections 5.3.1 and 5.3.2 present the relevant mapping rules.

MR-20a/b/c/d: $P4A : Plan \rightarrow CODE : Passed/Failed/Aborted/Constructor$

Body: The three methods and the constructor are placeholders that can be used by extension transformations to perform tasks on the initialization, termination, or failure of a plan. The methods are defined by the Jadex API.

MR-21a: $P4A : Plan \rightarrow CODE : FactoryInterface$

Body: As introduced in Section 4.1, an `InternalTask` is used at the modeling level as black box that represents business logic. In order to separate the custom code from the generated one, this mapping rule generates a factory interface for the source plan (see Figure 5.5). The factory provides one `create()` method for each `InternalTask` of the current plan. A `create()` method is responsible for instantiating the custom `InternalTask` implementation.

```
// instantiate custom InternalTask implementation
<$self.name> my<$self.name> =
  <$plan.name>Factory.create<$self.name>();
// set input parameters
<FOREACH $self.inparameter AS p>
  my<$self.name>.set<p.name>(<p.value>);
<END-FOREACH >
// execute business logic
my<$self.name>.execute();
// get output parameters
<FOREACH $self.outparameter AS p>
  <p.outvariable.name> = my<$self.name>.get<p.name>();
<END-FOREACH >
```

Listing 5.5: This listing depicts a template for invoking an `InternalTask` implementation (MR-22). $self refers to a Pim4Agents `InternalTask` and $plan to the `Plan` that contains the `InternalTask`.

MR-21b: *P4A* : *InternalTask* → *CODE* : *InternalTaskInterface*

Body: This rule maps an `InternalTask` to a Java interface class (see Figure 5.5). The interface provides an `execute()` method to run the encapsulated business logic. The method is invoked by the plan body (see MR-22). Moreover, the task's `Knowledge` parameters are mapped to get-/set-methods for accessing the input/output parameters.

5.3.1 Tasks

This section introduces a representative selection of mapping rules for Pim4Agents `Tasks`.

MR-22: *P4A* : *InternalTask* → *CODE* : *Invocation*

Body: Mapping rules MR-21a/b already introduced the `InternalTask` factory and the `InternalTask` interface (see Figure 5.5). This mapping rule is responsible for invoking a custom `InternalTask` implementation from a Jadex plan body. For this purpose, the `InternalTask` is mapped to the code template depicted in Listing 5.5. Before and after the `execute()` method is invoked, the input and output parameters are set according to the input and output `Knowledges` specified in Pim4Agents.

```
IGoal <$self.name> = createGoal("<$self.goal.name>");
<FOREACH $self.inparameter AS p>
 <$self.name>.getParameter("<p.name>").setValue("<p.value>");
<END-FOREACH >
this.dispatchSubgoal[AndWait](<$self.name>);
```

Listing 5.6: This listing depicts a code template for the `AssumeGoal` task (see MR-23; $self refers to an `AssumeGoal` task).

```
this.waitFor(<$self.timeout * 1000>);
```

Listing 5.7: This listing depicts a code template for the `Wait` task (MR-24; $self refers to a PIM4AGENTS `Wait` task).

MR-23a/b: $P4A : AssumeGoal[AndWait] \rightarrow CODE : DispatchGoal[AndWait]$

Body: Dispatching goals in PIM4AGENTS is done using the `AssumeGoal` (asynchronous) and `AssumeGoalAndWait` (synchronous) tasks. Both tasks are mapped to the corresponding API calls `dispatchSubgoal()` and `dispatchSubgoalAndWait()` (see Listing 5.6). The listing also depicts how the parameters (attribute `inparameter`) are assigned to the goal. The invocation of `createGoal()` demands that the referred `MGoal` is declared in the plan capability (see MR-8a/b and Figure 5.4).

MR-24: $P4A : Wait \rightarrow CODE : WaitFor$

Body: The PIM4AGENTS `Wait` task is used to wait for a timeout in seconds. Listing 5.7 depicts the according code template for Jadex.

MR-25: $P4A : InvokeWS \rightarrow CODE : WSInvocation$

Body: This mapping rule maps a PIM4AGENTS `InvokeWS` task to code for invoking a Web service. The details of orchestrating Web services with PIM4AGENTS and the according code generation have been presented in Xiaoqi Cao's master thesis [Cao, 2011] and are not detailed in this dissertation. Chapter 7 discusses how to use the `InvokeWS` concept for orchestrating object services in semantically-enriched virtual 3D worlds.

```
for(<$self.iterator.type> <$self.iterator.name>;
    <$self.condition>; <$self.iterator>++) {
 <$self.steps -> apply MR-19>}
```

Listing 5.8: Code template for a PIM4AGENTS `Loop` ($self refers to a PIM4AGENTS `Loop`).

MR-26: *P4A* : *InitiateProtocol* → *CODE* : *InitiateProtocolEvent*

Body: The initialization of an interaction protocol is done by posting an `MInternalEvent`. The `MInternalEvent` is handled by the according protocol capability. The details of the protocol transformation are discussed in Section 5.4

5.3.2 StructuredActivities

In contrast to atomic `Tasks`, `StructuredActivities` contain nested `Actvities`. In the following, the mapping rules for the `Loop` and `Decision StructuredActivities` are presented.

MR-27: *P4A* : *Loop* → *CODE* : *ForLoop*

Body: A PIM4AGENTS `Loop` is mapped to a Java *for-loop* (see Listing 5.8). There exist two different modes of how the `Loop` concept can be used: (i) using a counter variable and a termination condition or (ii) for iterating over a collection of values (e.g. a set of received messages). Listing 5.8 depicts the code template for the first case. The `Loop's` body consists of a sequence $(a_0, \ldots, a_n)$ of `Activities`. The `Activities` are mapped by MR-19 to code.

MR-28: *P4A* : *Decision* → *CODE* : *If − Then − Else*

Body: A PIM4AGENTS `Decision` is mapped to a Java *if-statement*. The `Decision's` body contains one `Begin` task, one `End` task, and a set of branches $(b_0, \ldots, b_n)$. The branches fork at the `Begin` and join at the `End` using `ControlFlows`. Each branch consists of a sequence $(a_0, \ldots, a_n)$ of `Activities` which are connected by `ControlFlows`. For each branch, the `Begin` task has an outgoing `ControlFlow` to the first `Activity` of a branch. The `ControlFlow` defines a `BooleanExpression` that is the precondition of that branch. Listing 5.9 depicts the code template for generating the according source code. The branches are mapped by MR-19.

```
<VAR branches = getBeginTask($self).outFlow>
<FOREACH branches AS b>
 <IF branches -> indexOf(b) = 0>
  if(<b.preCondition>) {
    <b.sink -> apply MR-19>
  }
 <ELSE-IF >
  else if(<b.preCondition>) {
    <b.sink -> apply MR-19>
  }
 <END-IF >
<END-FOREACH >
```

Listing 5.9: This listing depicts a code template for a PIM4AGENTS `Decision` (MR-28). The helper function `getBeginTask()` returns the `Begin` task of a `StructuredActivity`.

5.4 Interaction Mappings

This section presents the mapping rules related to the BOCHICA interaction metamodel. Figure 5.6 depicts an overview of the developed design pattern for Jadex. The pattern maps the BOCHICA interaction interface introduced in Section 3.3.1 to Jadex. It consists of three parts: (i) the initialization of the protocol by a user-defined plan using the PIM4AGENTS `InitiateProtocol` task (MR-26), (ii) the automatically generated capability(s) for managing the protocol execution (MR-29a/b, MR-30a/b, MR-31a/b), and (iii) the user-defined behaviors for executing the business logic. The protocol execution and management module consists of three sub-parts. The *protocol execution plan* implements the message handling of an `Actor` of a PIM4AGENTS interaction protocol (MR-31a/b). For example, it checks whether a certain state of a conversation has been reached using Jadex `MExpressions`, manages timeouts, and posts the user-defined goals for invoking the business logic. The message types, timeouts, and goals are defined in the *protocol execution capability* where the protocol execution plan is part of (MR-29a/b). The protocol execution capability defines the goals, message types, and timeouts as abstract placeholders. The placeholders are initialized by the *protocol configuration capability* (MR-30a/b). The separation leverages the reuse of the protocol execution plans and capabilities. For example, the same protocol might be initialized with different timeouts (e.g. 10s vs. 1h) and different content types of messages. For each of the `Actors` of a PIM4AGENTS interaction protocol, an own

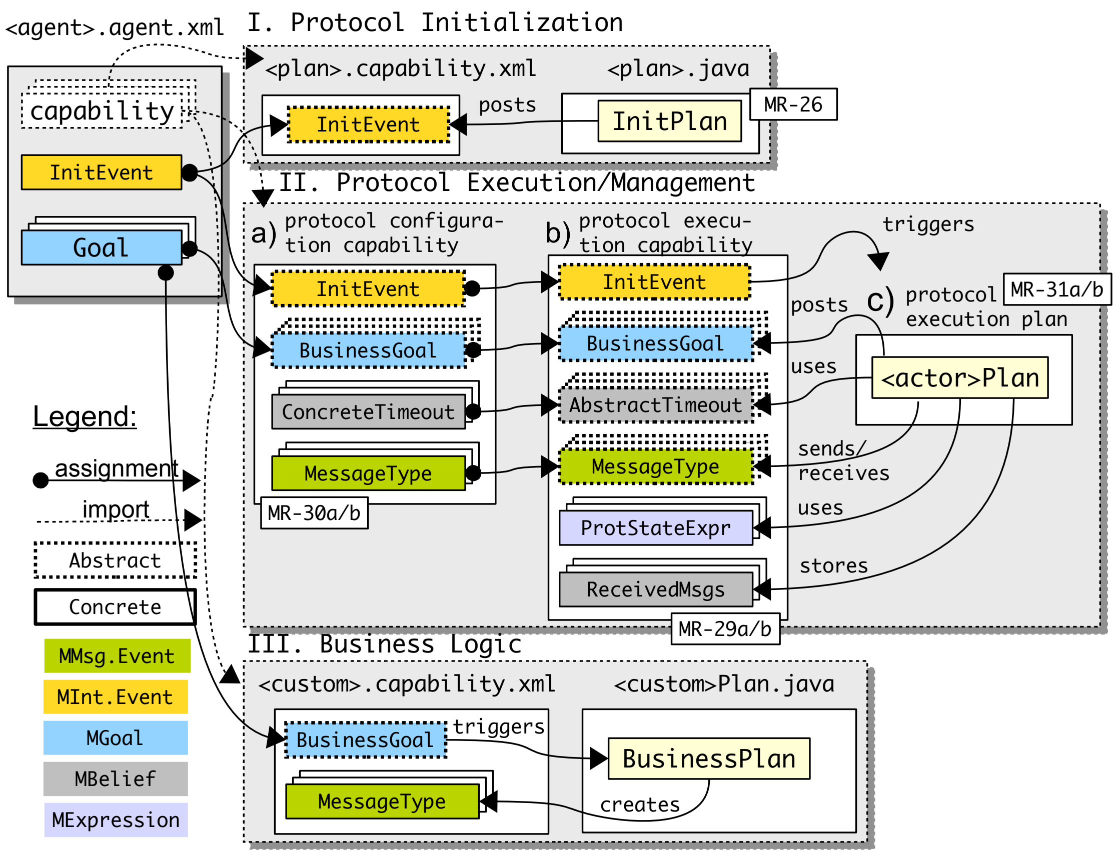


Figure 5.6: This figure depicts an overview of the design pattern used for realizing Bochica interaction protocols in Jadex. It consists of (I) the plan that initiates a protocol (see MR-26), (II) the protocol management capabilities and plans, and (III) the business logic capabilities for handling the messages (see Section 5.3). The protocol management (II) is separated into c) the protocol execution plan that sends and receives the messages (see MR-31a/b), b) the protocol execution capability (see MR-29a/b), and a) the protocol configuration capability (see MR-30a/b).

set of the just introduced plans and capabilities is generated. The remainder of this section introduces the involved mapping rules.

MR-29a/b: *P4A* : *Actor* → *BDI* : *MCapability*

Pre: a) $self.isInitiatorActor(); b) $self.isParticipantActor()

Body: This rule maps an `Actor` of a Pim4Agents `Protocol` to an according `MCapability` in Jadex. The capability enables an agent to participate in the protocol. The resulting protocol execution capabilities for the initiator (a) and the participant actor (b) are slightly different. For example, the participant capability

MR-29a/b: *P4A* : *Actor* → *BDI* : *MCapability*		
target	source	MR
name	The name of the source `Actor`.	–
imports	An import is generated for every data type (Java class) that is used by a Jadex `MBelief` or `MParameter`.	–
beliefs	Every PIM4AGENTS `TimeOut` of the current `Actor` is mapped to an abstract `MBeliefReference` in Jadex. The `MBeliefReference` is set from the protocol configuration capability and is accessed by the protocol execution plan. Moreover, for every received `ACLMessage` of the `Actor`, an own `MBelief` is created. It is used for collecting the received messages. The beliefs are used for evaluating state expressions (see MR-37).	33b, 34b
plans	The protocol execution capability declares exactly one plan. It is responsible for executing the protocol.	31a, 31b
goals	Every `AbstractGoal` of the PIM4AGENTS `Protocol` that is referenced by one of the current `Actor's MessageFlows` is mapped to an abstract `MPerformGoalReference`. The concrete goal is set by the protocol configuration capability (see Figure 5.6). The protocol execution plan posts the goals to invoke the business logic.	32
events	Every `ACLMessage` that is sent or received by the current actor is mapped to an abstract `MMessageReference`. The concrete message is set by the protocol configuration capability. The protocol execution plan uses the messages as templates for sending and receiving messages. Moreover, the `InitiateProtocol` task is mapped to an `MInternalEventReference` which initiates the protocol.	34b, 36b
expression	Every PIM4AGENTS `MessageFlow` of the current `Actor` is mapped to a Jadex `MExpression` which is used to compute whether a certain state (concept `MessageFlow`) in the communication has been reached.	37

Table 5.4: Details of mapping rules 29a/b.

declares no `MInternalEvent` to initiate a new conversation. Instead, the participant protocol execution plan is triggered by the first `MMessageEvent` sent by the initiator. The concrete message types, goal types, and timeout values are set by the protocol configuration capability. Table 5.4 depicts the details for this mapping rule.

MR-30a/b: *P4A* : *ProtocolConfiguration* → *BDI* : *MCapability*

Pre: a) *self.isInitiatorActor*(); b) *self.isParticipantActor*()

Body: This mapping rule maps a PIM4AGENTS `ProtocolConfiguration` to a protocol configuration `MCapability`. The `MCapability` initializes a protocol execution capability with concrete goals (MR-32), message events (MR-34a), and timeouts (MR-33a). The source concept `ProtocolConfiguration` provides the specification of the concrete artifacts. Analogous to MR-29a/b, this mapping rule distinguishes between initiator and participant protocol configurations. The main differences are: (i) a participant configuration declares no `MInternalEvent` to initialize the protocol and (ii) the message events have inverse sent/receive directions (a message which is sent from one party is received by the other).

MR-31a/b: $P4A : Actor \rightarrow CODE : Initiator - /ParticipantPlan$

Pre: a) $self.isInitiatorActor(); b) $self.isParticipantActor()

Body: This mapping rule is responsible for generating the protocol execution plan for the initiator (a) and participant (b) actors. The resulting code accesses the message events, beliefs, timeouts, etc. that are declared by the protocol execution capability (see MR-29a/b). How the plan is actually realized depends on (i) the used protocol specification approach at the platform-independent layer and (ii) whether the protocol model is interpreted or according executable code is generated. Since we want to stay independent of a concrete approach, we leave it open at this point.

MR-32: $P4A : AbstractGoal \rightarrow BDI : MPerformGoalReference$

Body: `AbstractGoals` are used in BOCHICA to represents business logic that has to be performed when a certain state of a conversation is reached (see Section 4.1). This rule maps an `AbstractGoal` to an `MPerformGoalReference` in Jadex. The concrete goals which realize an `AbstractGoal` are defined by the agent model. The BOCHICA `realizedBy` relationship is mapped to an `assignto` relationship in Jadex.

MR-33a/b: $P4A : TimeOut \rightarrow BDI : MBelief[Reference]$

Body: The `TimeOut` concept is used by a BOCHICA `Protocol` to specify a time constraint between two states (concept `MessageFlow`). It is mapped by MR-33a to an `MBelief` of a protocol configuration capability that holds the timeout value. MR-33b maps the `TimeOut` to an abstract `MBeliefReference`. The `MBeliefReference` is a placeholder for the concrete timeout value in the protocol execution capability (see Figure 5.6). The assignment of the `MBelief` to the `MBeliefReference` is done using the `assignto` attribute.

MR-34a/b: $P4A : Message(\$actor) \rightarrow BDI : MMessageEvent[Reference]$

Pre: $actor.sentMessages -> includes($self.aclMessage)

Body: MR-34a maps a BOCHICA `Message` to a Jadex `MMessageEvent`. The precondition ensures that the `Actor` (passed as parameter), for which this rule is invoked, is the sender of the current `Message`. The direction of the message is set to *sent* and the according FIPA parameters are set as specified by the BOCHICA model. MR-34b maps a PIM4AGENTS `Message` to an `MMessageEventReference`. It is used by the protocol execution capability as placeholder for the concrete message event mapped by MR-34a.

MR-35a/b: $P4A : Message(\$actor) \rightarrow BDI : MMessageEvent[Reference]$

Pre: $actor.receivedMessages -> includes($self.aclMessage)

Body: MR-35a maps a BOCHICA `Message` to a Jadex `MMessageEvent`. The precondition ensures that the `Actor` (passed as parameter), for which this rule is invoked, is the receiver of the current `Message`. The direction is set to *received* and the according FIPA parameters are set as specified by the BOCHICA model. MR-35b maps a PIM4AGENTS `Message` to an `MMessageEventReference`. It is used by the protocol execution capability as placeholder for the concrete message event mapped by MR-35a.

MR-36a/b: $P4A : ProtocolConfiguration \rightarrow BDI : MInternalEvent[Reference]$

Body: Protocols in BOCHICA are initialized using the `InitiateProtocol` task. The task is mapped by MR-26 to code. The code posts an `MInternalEvent` that triggers the protocol execution plan (see Figure 5.6). Mapping rule MR-36a maps a `ProtocolConfiguration` to an `MInternalEvent` that initializes a certain protocol configuration. The `assignto` attribute is used to pass the event to (i) all plan capabilities which contain an `InitiateProtocol` task that references the current `ProtocolConfiguration ($self)` and (ii) the protocol capability generated for `$self`. Mapping rule MR-36b maps the `ProtocolConfiguration` to an abstract `MInternalEventReference`. The `MInternalEventReference` is a placeholder for the concrete event (assigned by the agent).

MR-37: *P4A* : *MessageFlow* → *BDI* : *MExpression*

Body: As introduced in Section 4.1, a `MessageFlow` in BOCHICA represents a state in a conversation. The code generation pattern depicted in Figure 5.6 uses a Jadex `MExpression` to check whether a certain state has been reached. This mapping rule is responsible for generating such an expression. How the expression is mapped to Jadex depends on the protocol specification approach used at the platform-independent layer and is left open at this point.

5.5 Data Model Mappings

As introduced in Section 3.3.3, BOCHICA data models are based on Ecore. The remainder of this section defines mapping rules related to the BOCHICA data model.

MR-38a/b: *P4A* : *Knowledge* → *BDI* : *MBelief*[*Reference*]

Pre: $self.eContainer.oclIsTypeOf(P4A:Agent) or
$self.eContainer.oclIsTypeOf(P4A:Capability)

Body: This rule maps a BOCHICA agent's or capability's `Knowledges` to `MBeliefs` in Jadex. The `MBelief` has the name of the `Knowledge` and the `value` attribute is used to set the initial value. For every `PlanUse` and `CapabilityUse` of an agent or capability that references the `Knowledge`, the according Jadex `assignto` property is set for passing the `MBelief` to the `MBeliefReference` (MR-38b).

MR-39a/b/c: *P4A* : *Expression* → *BDI* : *MStaticValue*, *BDI* : *MInternalCondition*, *CODE* : *Expression*

Body: This rule maps a BOCHICA `Expression` to code. Since the expression language depends on the used language extension, this rule has to be specialized by an extension transformation (see Section 3.3.2). One has to distinguish between (i) expressions of Jadex XML files (concepts `MStaticValue` and `MInternalCondition`) and (ii) expressions of Java behaviors (e.g. an if-then-else condition). By default, the `Expression's text` attribute is mapped one-to-one to code.

Bochica Type	Java Type
EString	java.lang.String
EBoolean	boolean
EChar	char
EInt	int
ELong	long
EByte	byte
EClass	java.lang.Class
P4A:Sequence	java.util.Vector
P4A:HashMap	java.util.HashMap
P4A:Set	java.util.Set
P4A:Message	jadex.bdi.runtime.IMessageEvent
P4A:ConcreteGoal	jadex.bdi.runtime.IGoal
P4A:Signal	jadex.bdi.runtime.IInternalEvent

Table 5.5: Mapping of Bochica data types to Java types.

MR-40: *P4A* : *EType* → *CODE* : *Type*

Body: This rule maps a Bochica `EType` to concrete code. Table 5.5 depicts an overview of the type mappings to Java and the Jadex API.

MR-41: *Ecore* : *EClass* → *CODE* : *JavaClass*

Pre: $self.instanceClassName = OclUndefined

Body: The generation of Java code from Ecore models is supported by EMF. However, Listing 5.10 depicts an alternative code generation pattern which supports the Jadex BDI reasoning engine. The precondition ensures that no code is generated for `EClasses` with an `instanceClassName`. The `instanceClassName` attribute is used when there already exists Java code at the platform (e.g. as part of a library). Jadex makes use of the *Java Beans* design pattern. The Java Beans pattern provides a notification mechanism when attributes of a Java class are modified. This enables the agent to react to belief changes. For example, the `set()` method invokes the `firePropertyChange()` method to notify listeners about a changed property value.

```
package <$self.package>;

import ...

public class <$self.name>Impl extends <$self.name> {

 private PropertyChangeSupport p;
 <FOREACH $self.eAttributes AS a>
  private <a.eAttributeType> <a.name>;
 <END-FOREACH >

 public class() {
  this.p = new PropertyChangeSupport(this);
 }

 <FOREACH $self.eAttributes AS a>
  public set<a.name>(<a.eAttributeType> <a.name>) {
   <a.eAttributeType> old = this.<a.name>;
   this.<a.name> = <a.name>;
   this.pcs.firePropertyChange("<a.name>", old, this.<a.name>);
  }
  public <a.eAttributeType> get<a.name>() {
   return <a.name>;
  }
 <END-FOREACH >
 ...
 public void addPropertyChangeListener
  (PropertyChangeListener l) {
    p.addPropertyChangeListener(l);
 }
 public void removePropertyChangeListener
  (PropertyChangeListener l) {
    p.removePropertyChangeListener(l);
 }}
```

Listing 5.10: Java Beans code template for an Ecore `EClass` in Jadex.

```
mapping P4A::agent::Agent::toBeliefBase() : JADEX::MBeliefbase{
    belief := self.knowledge.map toBelief();
    map adjustBeliefBase(result);
}

mapping inout adjustBeliefBase(inout beliefBase : JADEX::MBeliefbase) {
}

mapping pim4agents::informationmodel::Knowledge::toBelief() : JADEX::MBelief {
 name := toJavaVariableName(self.name);
 _class := getTypeForKnowledge(self);
 var uzez : Set(pim4agents::behavior::PlanUse) :=
  this.planUses -> select(d | d.bindings
   -> select(e | e.source -> includes(self)) -> size() > 0) -> asSet();
 assignto := uzez.map toPlanBeliefAssignment(self) -> flatten();
 var initExp : String := "";
 if(self.value = null or self.value.length() = 0) then {
  initExp := self.type.getInitializerExpression();
 } else {
  initExp := self.value.map toInitializerExpression();
 } endif;
 fact := object MInternalExpression {text := initExp};
 map adjustBelief(result);
}

mapping inout adjustBelief(inout belief : JADEX::MBelief) {
}
```

Figure 5.7: Example QVT-based implementation of MR-38a.

5.6 Summary

This chapter presented conceptual mappings for bridging the gap between Bochica and the Jadex BDI agent platform. The conceptual mappings build the basis for the implementation of the Jadex base transformation. The code quality of manual implementations of a Jadex application depends highly on the agent engineer's experience. The design patterns presented in this chapter anticipate design decisions (e.g. for implementing interaction protocols) and support engineers to guarantee a certain quality level. By further developing the base transformation over time, all applications that make use of it directly benefit from improvements. The conceptual mappings have been implemented with QVT and XPand as a Bochica base transformation. The mapping rules build the infrastructure for extension transformations. Figure 5.7 depicts an example implementation of MR-38a in QVT. It is responsible for configuring the `MBeliefbase` with `MBeliefs`. The first mapping rule prepares the `MBeliefbase` (see Section 4.2.3) of a Jadex agent. For every `Knowledge` of a Pim4Agents agent, the `toBelief` mapping rule is invoked. The `toBelief` mapping rule configures the name and data type of the `MBelief`. Moreover, it passes the belief to behaviors that make use of it (concept `PlanUse`). The mapping rule uses additional helper functions. For example, the `getTypeForKnowledge()` function converts the Ecore-based data type specification of Bochica to a concrete Java type. The variables `self` and `result` are predefined variables of QVT which represent the source and target objects of the mapping rule. The `adjustBeliefbase` and `adjustBelief` mapping rules are overloaded by an extension transformation for modifying the existing base transformation. For example, an extension transformation can add additional `MBeliefs` to the `MBeliefbase`. The XPand-based mapping rules are similar to the code templates presented in this chapter.

Chapter 6

Model-driven Reverse Engineering of BDI Agents

Agent-Oriented Reverse Engineering (AORE) is concerned with extracting the underlying design of concrete implemented MAS. The extracted artifacts can be used (i) to visualize and analyze an existing MAS implementation, (ii) to extract the design for later reuse, and (iii) to migrate a SUC from traditional AOSE to model-driven AOSE. Model-driven AORE uses model transformations for bridging the gap between the platform-specific and the platform-independent layer (in upward-direction). The remainder of this chapter introduces a model-driven agent-oriented MDRE approach for the BDI platform Jadex. Figure 6.1 a) depicts an overview of the developed reverse engineering architecture. The process consists of (i) lifting the platform artifacts to the platform-specific layer and (ii) extracting the underlying design. The design extraction encompasses model artifacts like (i) the problem decomposition in terms of organizational structures, agent types, and means-end decompositions, (ii) behavior templates and interaction patterns, and (iii) deployment configurations as well as data models. The extracted artifacts are manually refined where necessary.

Before we define the actual conceptual mappings, we provide an overview of the technical infrastructure that builds the foundation of the mapping rules. The first task is to lift the Jadex source code (e.g. Jadex Java and XML files) to the Jadex PSM presented in Section 4.2. This task is already supported by EMF (XML files) and the MoDisco project (Java files). A fully automatic *model extractor* integrates both tools and assembles the Jadex PSM based on the platform artifacts. The lifting is followed by the design extraction step which maps the underlying design to Bochica. As discussed in Section 4.2, references between Jadex model artifacts (e.g. Jadex application and BDI models) are only represented as plain strings which cannot be directly processed by the model transformation implementation.

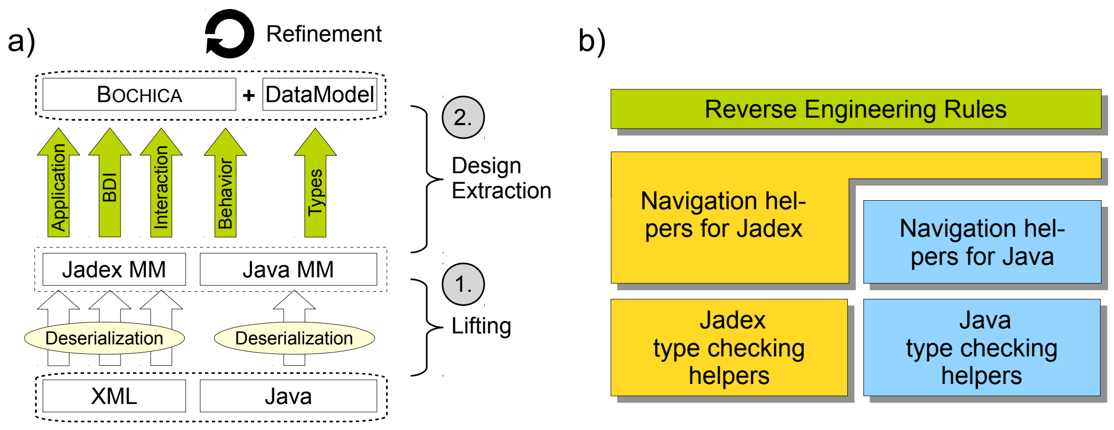


Figure 6.1: a) depicts the transformation architecture. b) depicts the technical infrastructure of the design extraction step.

Moreover, the generic Java metamodel does not define Jadex-specific types. For this purpose, Table 4.1 defines additional constraints which enable the generic Java metamodel for the use with Jadex. In order to address those technical problems, Figure 6.1 b) depicts an overview of the technical infrastructure of the model transformation. The lowest layer consists of helper functions for checking Jadex and Java types (e.g. whether a Java `MethodInvocation` is an invocation of the Jadex API's `dispatchSubgoal()` method). Based on the type checking helpers, additional helpers for resolving string references between XML and Java files are defined. Based on this infrastructure, the actual mapping rules are defined. One general problem of the design extraction step is that there exist different flavours of how to use the infrastructure provided by the Jadex platform for implementing a MAS. For example, there exist no fixed patterns for implementing interaction protocols or organizational structures in Jadex. Moreover, Java-based behaviors provide numerous possibilities to implement a system (e.g. by using nested classes and complex inheritance relationships). Thus, it is difficult to specify generic rules that cover all cases. Due to those problems, the extracted structures have to be manually refined (e.g. by using existing platform-independent artifacts like interaction protocols).

In the following, we specify the *Mapping Rules* (MR) for extracting the underlying design of Jadex applications and representing it in PIM4AGENTS. The mappings are structured into application mappings (Section 6.1), BDI mappings (Section 6.2), behavior mappings (Section 6.3), interaction mappings (Section 6.4), and data model mappings (Section 6.5). Following metamodel abbreviations are used: Jadex Project (PROJ), Jadex Application (APP), Jadex BDI (BDI), Java (JAVA), and PIM4AGENTS (P4A). The mapping rules presented in the remainder

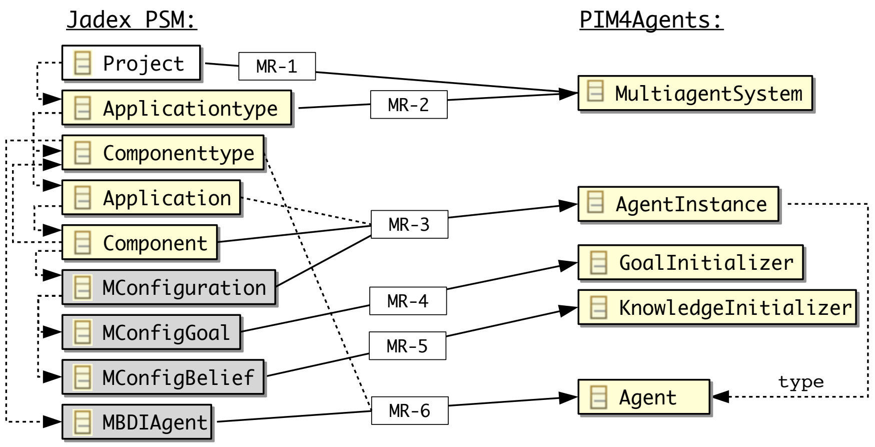


Figure 6.2: The left hand side depicts concepts of the Jadex project metamodel (white), the application metamodel (yellow), and BDI metamodel (gray). The right hand side depicts the target concepts in PIM4AGENTS. The dotted lines highlight dependencies.

of this section consist of (i) the *head* of the mapping rule, (ii) the OCL *precondition* that has to be fulfilled in order to apply the mapping rule, and (iii) the *body* that specifies the actual mapping. Finally, Section 6.6 summarizes and discusses the approach.

6.1 Application Mappings

The `Project` concept is the main container of the Jadex PSM presented in Section 4.2. It aggregates a set of Jadex application models (root concept `Application-type`). Each application model declares the utilized agent types (concept `Compo-nenttype`) and one or more deployment configurations (concept `Application`). A deployment configuration specifies a number of agent instances (concept `Compo-nent`). In order to identify the application model and configuration to be mapped, the model transformation requires the two parameters `$applicationName` and `$configurationName`. Figure 6.2 depicts an overview of the application mappings.

MR-1: *PROJ* : *Project* → *P4A* : *MultiagentSystem*

Body: The entry point of the reverse transformation is the mapping from `Project` to `MultiagentSystem`. The parameter `$applicationName` identifies the application model to be transformed (concept `Applicationtype`). MR-2 is applied to map the user-specified application to a PIM4AGENTS `MultiagentSystem`.

MR-2: *APP* : *Applicationtype* → *P4A* : *MultiagentSystem*		
target	source	MR
name	The name of the Jadex `Applicationtype` is mapped to the name of the Pim4Agents `MultiagentSystem`.	–
agent	Each `Componenttype` declaration of a Jadex `Applicationtype` is resolved to the actual `MBDIAgent` model. The resolved `MBDIAgent` is mapped to a Pim4Agents `Agent`.	6
behavior	Each Jadex Java plan class (concept `AbstractTypeDeclaration`) that is used by an agent (concept `MBDIAgent`) or capability (concept `MCapability`) of the Jadex application is mapped to a Pim4Agents `Plan`.	13
capability	Each capability (concept `MCapability`) used by agents of the Jadex application is mapped to a Pim4Agents `Capability`.	7
protocolConfiguration	Jadex has no explicit representation of interaction protocols. Thus, all `MMessageEvents` declared by Jadex agents or capabilities are transformed to Pim4Agents `Messages`. All messages of the same agent or capability are grouped by a Pim4Agents `ProtocolConfiguration` (for later manual refinement).	22, 23
goal	Each goal (concept `MGoal`) used by agents of the Jadex application is mapped to a Pim4Agents `ConcreteGoal`.	10
instance	Each `Component` that is contained by the user-specified deployment configuration (concept `Application`) is mapped to a Pim4Agents `AgentInstance`.	3
event	Every Jadex `MInternalEvent` that is declared by an agent or capability of the application is mapped to a Pim4Agents `Signal`.	9
type	The data types used within a Jadex application are lifted to an external data model based on Ecore. The lifted data types are imported by the concept `EType` into Pim4Agents.	26, 27

Table 6.1: Details of the application mapping (MR-2).

MR-2: *APP* : *Applicationtype* → *P4A* : *MultiagentSystem*

Pre: \$self.name = \$applicationName

Body: The `Applicationtype` concept is the top-most container of a Jadex application and is mapped to a Pim4Agents `MultiagentSystem`. The `MultiagentSystem` is the root element of a Pim4Agents model. Table 6.1 depicts the details of this mapping rule. The parameter `$configurationName` is used to identify the deployment configuration to be mapped (concept `Application`).

MR-3: *APP* : *Component*, *BDI* : *MConfiguration* → *P4A* : *AgentInstance*

Body: The agent instances of a Jadex application are specified by the concept `Component`. A `Component's type` attribute refers to an agent type (concept `Componenttype`). The `filename` attribute of the `Componenttype` is used to resolve the concrete Jadex agent model (concept `MBDIAgent`). The resolved `MBDIAgent` is mapped by MR-6 and used as type of the PIM4AGENTS `AgentInstance`. Moreover, a Jadex `Component` refers to a configuration (concept `MConfiguration`) that is part of the resolved Jadex `MBDIAgent`. The `MConfiguration` defines the initial beliefs and goals of the `AgentInstance` (see MR-4 and MR-5).

MR-4: *BDI* : *MConfigGoal* → *P4A* : *GoalInitializer*

Body: The Jadex `MConfigGoal` concept is used to assign an initial goal to an agent instance. It corresponds to the PIM4AGENTS `GoalInitializer` concept. The referenced Jadex goal type (concept `MGoal`) is resolved and assigned to the `GoalInitializer` (after it has been mapped by MR-10).

MR-5: *BDI* : *MConfigBelief* → *P4A* : *KnowledgeInitializer*

Body: The Jadex `MConfigBelief` concept initializes an agent instance's `MBelief`. It corresponds to the PIM4AGENTS `KnowledgeInitializer` concept. The `MBelief` declaration is resolved and assigned to the PIM4AGENTS `KnowledgeInitializer` (after it has been mapped by MR-8).

6.2 BDI Mappings

The mapping rules related to the Jadex BDI metamodel extract the agent types of the MAS. This includes capabilities, internal events, and the goal hierarchy. One further important aspect is the visibility (scoping) of beliefs between an agent and its capabilities and plans. Jadex has no explicit representation of organizational structures. How an agent manages its relations to other agents depends on the engineer who designed the MAS. Thus, there exists no generic way for extracting the organizational structure (see discussion in Section 6.6). Figure 6.3 depicts an overview of the mappings of the BDI, behavior, and interaction mappings.

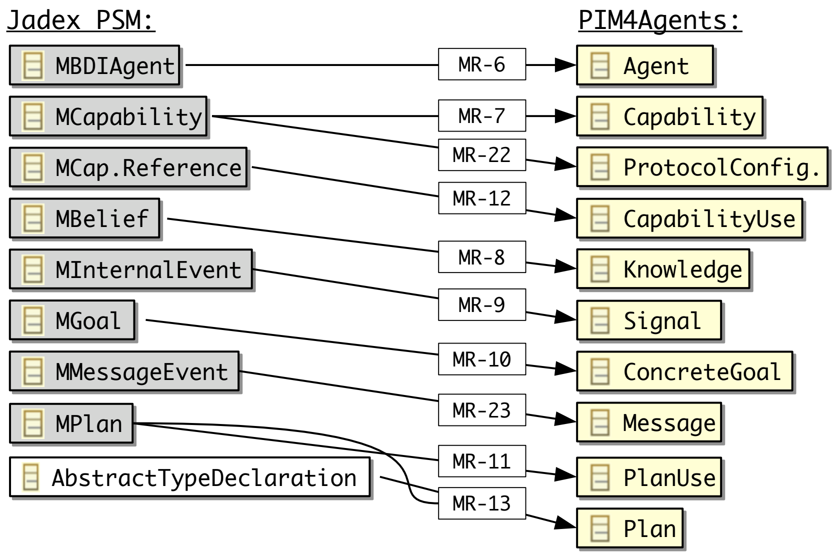


Figure 6.3: The left hand side depicts concepts of the Jadex BDI metamodel (gray) and Java metamodel (white). The right hand side depicts the target concepts in Pim4Agents.

MR-6: *BDI* : *MBDIAgent* → *P4A* : *Agent*

Body: This rule maps a Jadex `MBDIAgent` to a Pim4Agents `Agent`. Table 6.2 summarizes the details. Each `MPlan` declaration of the `MBDIAgent` is mapped by MR-11 to a Pim4Agents `PlanUse`. Additionally, a default `DomainRole` with the name of the agent is created. The `DomainRole` is used during the manual refinement step for modeling missing organizational structures.

MR-7: *BDI* : *MCapability* → *P4A* : *Capability*

Body: As discussed in Section 4.2, the concept `MBDIAgent` inherits from `MCapbility`. Thus, the mapping rules for both concepts are very similar and we refer to Table 6.2 for the details. The major difference is that no `DomainRole` is created for the `MCapability` (as it has no own execution thread).

MR8: *BDI* : *MBelief* → *P4A* : *Knowledge*

Body: The concept `MBelief` is used by a Jadex agent or capability to store information. The name of the `MBelief` is assigned as name of the Pim4Agents `Knowledge`. The data type of the `MBelief` is mapped by MR-27 and assigned to the `Knowledge's type` attribute.

MR-6: *BDI* : *MBDIAgent* → *P4A* : *Agent*		
target	source	MR
name	The name of the Jadex `MBDIAgent` is set as the name of the Pim4Agents `Agent`.	–
know-ledge	Each `MBelief` that is declared by the `MBDIAgent` is mapped to a Pim4Agents `Knowledge`.	8
capa-bility-Use	Each `MCapabilityReference` of the `MBDIAgent` is mapped to a `CapabilityUse` of the Pim4Agents `Agent`. An `MCapability-Reference` represents an imported `MCapablity`.	12
plan-Use	Each `MPlan` that is declared by the `MBDIAgent` is mapped to a Pim4Agents `PlanUse`. The `PlanUse` contains a reference to the Pim4Agents `Plan` which is generated from the Java plan class (concept `AbstractTypeDeclaration`).	11, 13
domain-role	A default Pim4Agents `DomainRole` is created. It has the name of the agent.	

Table 6.2: Details of the agent mapping (MR-6).

MR-9: *BDI* : *MInternalEvent* → *P4A* : *Signal*

Body: This rule maps a Jadex `MInternalEvent` to a `Signal` in Pim4Agents. The `MInternalEvent's MParameter` elements are mapped by MR-24 to Pim4-Agents `Knowledges`.

MR-10: *BDI* : *MGoal* → *P4A* : *ConcreteGoal*

Body: This rule defines the mapping of a Jadex `MGoal` to a Pim4Agents `ConcreteGoal`. Pim4Agents as well as Jadex distinguish between perform goals, achieve goals, maintain goals, and query goals (see Chapter 4). The goal types are mapped accordingly. An `MGoal's name` is set as the name of the Pim4Agents `ConcreteGoal`. The Jadex `MGoal's MParameter` elements are mapped by MR-24 to Pim4Agents `Knowledges`. Conflicts between Jadex `MGoals` defined by the `inhibits` relationship are mapped to the `conflictingGoals` relationship in Pim4Agents. The target and maintain conditions are assigned to the corresponding attributes in Pim4Agents.

Goal Hierarchy Extraction. Goals and goal hierarchies play a central role in AOSE. During the design of a SUC, agent engineers decompose complex goals into sub-goals. The goal analysis is usually performed using AND/OR-decomposition trees. A means-end analysis step is used to specify how goals are achieved by means (behaviors). Usually, this process is done top-down. The model-driven AORE approach for Jadex starts with an already existing means-end decomposition and extracts the goal hierarchy as far as possible (bottom-up). Goal hierarchies are not explicitly represented in Jadex. The design extraction transformation analyzes the `dispatchSubgoal()` Jadex API invocations inside the `body()` method of the Java behaviors. Moreover, the `trigger` properties of the `MPlans` are used to compute the goal hierarchy. Figure 6.4 depicts the patterns in Jadex and their mappings to PIM4AGENTS. The *relevant plan set* Π_g represents the set of plan templates that are triggered by an `MGoal` g. The computed sub-goals of an `MGoal` are assigned to a PIM4AGENTS goal's `subgoalLinks` property. It is assumed that plans post goals sequentially. Basically, we distinguish three cases:

1. $\mid \Pi_g \mid = 1$: There exists exactly one relevant plan $\pi \in \Pi_g$ for goal g. The relevant plan π posts a sequence of sub-goals $(g_1, \ldots, g_n)$. Since we assume that goals are posted sequentially, an AND-decomposition from g to $(g_1, \ldots, g_n)$ is defined (case a). If π does not post sub-goals, g is mapped without any decompositions (case b).

2. $\mid \Pi_g \mid > 1$: If there exists more than one relevant plan for g, it depends on the plans' preconditions which one is applicable in a certain situation. Thus, g and its sub-goals $(g_{11}, \ldots, g_{1x}, \ldots, g_{n1}, \ldots, g_{ny})$ are mapped to an OR-decomposition.

3. $\mid \Pi_g \mid = 0$: Every goal g should have at least one relevant plan. If this is not the case, g is mapped by MR-10 without any decomposition information.

MR-11: *BDI* : *MPlan* $\rightarrow$ *P4A* : *PlanUse*

Body: An `MPlan` defines the header information of a Jadex behavior specification (e.g. triggering event and precondition). Moreover, it references the Java class which implements the behavior and defines the parameters that are passed by the agent to the plan (concept `MPlanParameter`). This rule maps a Jadex `MPlan` to a PIM4AGENTS `PlanUse`. The `PlanUse` concept specifies the parameter interface using the PIM4AGENTS `KnowledgeBinding` concept. One `KnowledgeBinding` is created for every `MPlanParameter` of the `MPlan`. All other behaviors related aspects (e.g. the behavior implementation) are handled by MR-13.

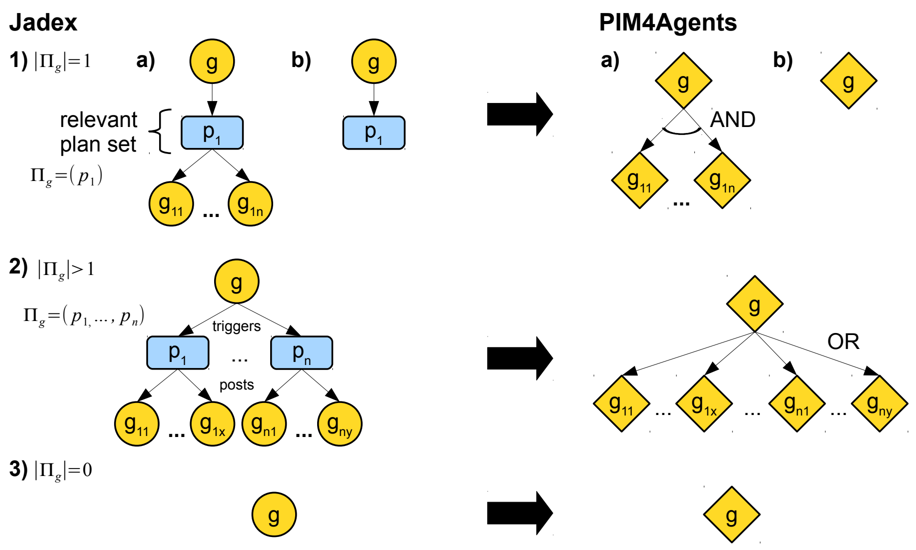


Figure 6.4: This figure shows how the goal hierarchy is extracted from Jadex. It is assumed that plans post goals sequentially.

MR-12: *BDI* : *MCapabilityReference* → *P4A* : *CapabilityUse*

Body: The `MCapabilityReference` concept imports an `MCapability` to a Jadex `MBDIAgent` (similar to a `PlanUse` for plans). This rule maps a Jadex `MCapability-Reference` to a PIM4AGENTS `CapabilityUse`. The `CapabilityUse` specifies the parameter interface using the PIM4AGENTS `KnowledgeBinding` concept. For every `MBelief` of the `MBDIAgent` that defines an `assignto` relationship for passing the `MBelief` to the imported `MCapability`, an according PIM4AGENTS `Knowl-edgeBinding` is created.

6.3 Behavior Mappings

As already discussed, the Java classes of a Jadex application are lifted to a Java model. Since the underlying Java metamodel only defines generic Java concepts (not specific to Jadex), the Jadex API is additionally taken into consideration to identify the relevant artifacts (see Table 4.1). In the following, the preconditions of the mapping rules define Jadex-specific constraints to the Java concepts (e.g. whether a Java `MethodInvocation` represents an invocation of the Jadex `dispatchSubgoal()` method). Figure 6.5 depicts an overview of the mappings related to the Jadex behavior metamodel. Every Java class that implements a Jadex plan has a `body()` method which implements the actual behavior. The `body()` method consists of a sequence of Java statements $(s_0, \ldots, s_n)$ of concept

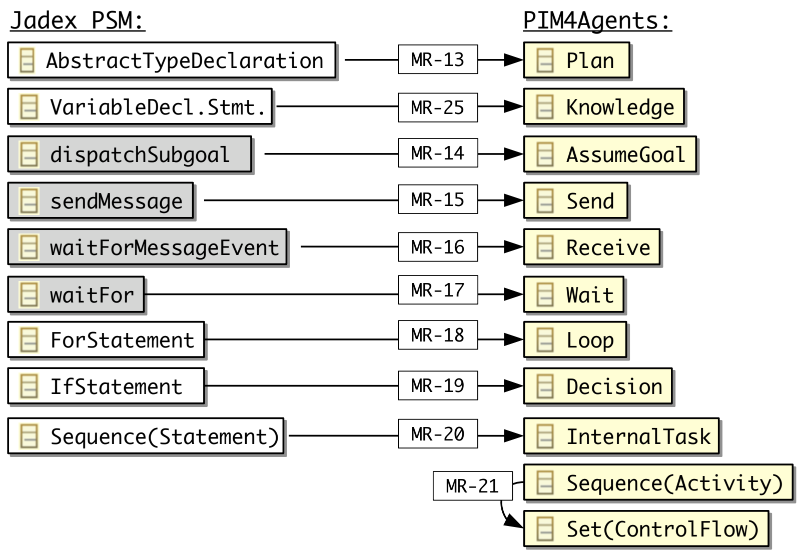


Figure 6.5: The left hand side depicts concepts of the Java metamodel. The gray concepts are instances of the Java concept `MethodInvocation`. The right hand side depicts the target concepts in PIM4AGENTS.

`Statement`. In order to map the a Java-based behavior to a PIM4AGENTS `Plan`, the transformation iterates over all statements of the behavior and applies the according mapping rules. Algorithm 6.1 depicts the according process. Java concepts that are mapped to PIM4AGENTS `StructuredActivities` contain nested code blocks with additional `Statements` (e.g. an if-then-else or a for-loop). The sub-sequences are recursively transformed by Algorithm 6.1. Concepts for which no mapping rule applies are aggregated and mapped to an `InternalTask` (see MR-20). An `InternalTask` in PIM4AGENTS is used to encapsulate business logic or aspects that are not modeled in PIM4AGENTS (e.g. algorithms). The variables that are accessed by the aggregated statements are used as interface for the `InternalTask`. Besides the mapped `Activities`, Algorithm 6.1 also returns a set of `ControlFlows`. The `ControlFlows` link the mapped `Activities` according to their execution order.

MR-13: $JAVA : AbstractTypeDeclaration, BDI : MPlan \rightarrow P4A : Plan$

Pre: \$self.superclass = “jadex.bdi.runtime.Plan”

Body: This rule maps a Java class (concept `AbstractTypeDeclaration`), that inherits from the Jadex API class `jadex.bdi.runtime.Plan`, to a PIM4AGENTS `Plan`. Additionally, the plan’s meta information defined by the `MPlan` is taken into consideration. Table 6.3 depicts the details of this mapping rule.

Algorithm 6.1 This algorithm controls the behavior extraction. The *stmts* parameter is a sequence of Java `Statements` $(s_1, \ldots, s_n)$. The algorithm returns a sequence of PIM4AGENTS `Activities` that are connected by `ControlFlows`. The function `relevantStatement()` returns true if the parameter is an instance of a type that is handled by one of the mapping rules defined by this section. "MR-X" stands for the rule that is applicable to a Java `Statement`.

```
processStatementSequence($stmts$) {
$activities := Sequence\{\}$
$flows := Set\{\}$
$unknown := Sequence\{\}$
for $i = 1$ to $| stmts |$ do
    $s := stmts[i]$
    if $relevantStatement(s) \wedge | unknown | = 0$ then
        $activities := activities.append($MR-X$(s))$
    else if $relevantStatement(s) \wedge | unknown | > 0$ then
        $activities := activities.append($MR-20$(unknown))$
        $activities := activities.append($MR-X$(s))$
        $unknown := Sequence\{\}$
    else
        $unknown := unknown.append(s)$
        if $i = | body |$ then
            $activities := activities.append($MR-20$(stmts))$
        end if
    end if
end for
$flows :=$ MR-21$(activities)$
return $Sequence\{activities, flows\}$
}
```

MR-14a/b: *JAVA* : *MethodInvocation* → *P4A* : *AssumeGoal*

Pre: a) $self.name = "jadex.bdi.runtime.Plan.dispatchSubgoal";
b) $self.name = "jadex.bdi.runtime.Plan.dispatchSubgoalAndWait"

Body: Goals in Jadex are posted by calling the (a) `dispatchSubgoal()` or (b) `dispatchSubgoalAndWait()` methods (asynchronous and synchronous). Both methods get an instance of an `MGoal` declaration as input. The method invocation is mapped to a PIM4AGENTS `AssumeGoal` task. The declaration of the posted `MGoal` is resolved within an agent or capability model and assigned to the PIM4AGENTS `AssumeGoal` task (see MR-10).

MR-13 *JAVA* : *AbstractTypeDeclaration*, *BDI* : *MPlan* → *P4A* : *Plan*		
target	source	MR
name	The name of the Java `AbstractTypeDeclaration` is assigned to the Pim4Agents `Plan`.	–
local-Know-ledge	Each `VariableTypeDeclarationSatement` is mapped to a Pim4Agents `Knowledge`.	25
steps	The plan class's `body()` method is resolved (concept `AbstractMethodDeclaration`) and all contained Java `Statements` are mapped by Algorithm 6.1 to corresponding Pim4Agents `Activities`. The resulting task sequence is started by a `Begin` and terminated by an `End` task.	14 to 20
control-Flow	Algorithm 6.1 also returns a set of Pim4Agents `ControlFlows` which define the execution order of the mapped `Activities`.	21

Table 6.3: Details of the behavior mapping (MR-13).

MR-15a/b: *JAVA* : *MethodInvocation* → *P4A* : *Send*

Pre: a) \$self.name = "jadex.bdi.runtime.Plan.sendMessage";
b) \$self.name = "jadex.bdi.runtime.Plan.sendMessageAndWait"

Body: Messages in Jadex are sent using the (a) `sendMessage()` and (b) `sendMessageAndWait()` methods. The methods get an `MMessageEvent` as parameter. The referenced `MMessageEvent` declaration is resolved inside an agent or capability model. The resolved message is mapped by MR-23 and assigned to the Pim4Agents `Send` task's `message` attribute (see MR-23). Resolving the message declaration is not always possible since Jadex message events can also be assembled in Java code which makes the extraction difficult.

MR-16a/b: *JAVA* : *MethodInvocation* → *P4A* : *Receive*

Pre: a) \$self.name = "jadex.bdi.runtime.Plan.waitForMessageEvent";
b) \$self.name = "jadex.bdi.runtime.Plan.waitForReply"

Body: The Jadex (a) `waitForMessageEvent()` and (b) `waitForReply()` methods are mapped to a Pim4Agents `Receive` task. The methods get an `MMessageEvent` as parameter. The `MMessageEvent` defines a template for the event to be received. The parameter is resolved to an `MMessageEvent` declaration in an agent or capability model. This is not always possible since Jadex message events can also be assembled in Java code which makes a generic extraction difficult.

MR-17: *JAVA* : *MethodInvocation* $\rightarrow$ *P4A* : *Wait*

Pre: \$self.name = “jadex.bdi.runtime.Plan.waitFor”

Body: A call to the Jadex `waitFor()` method is mapped to a PIM4AGENTS `Wait` task. The method’s timeout parameter is assigned to the `timeout` attribute of the `Wait` task.

MR-18: *JAVA* : *ForStatement* $\rightarrow$ *P4A* : *Loop*

Body: A `ForStatement` is directly mapped to a `Loop` in PIM4AGENTS. The declared variables (concept `VariableDeclarationStatement`) are mapped to `Knowledges` (see MR-25). The contained Java `Statements` are transformed by Algorithm 6.1 to PIM4AGENTS `Activities`. Additionally, a PIM4AGENTS `Begin` task is prepended to the resulting sequence and an `End` tasks is appended. The according `ControlFlows` are created and assigned to the `Loop’s flows` attribute.

MR-19: *JAVA* : *IfStatement* $\rightarrow$ *P4A* : *Decision*

This rule maps a Java `IfStatement` to a PIM4AGENTS `Decision`. Each branch $b_1, \ldots, b_n$ of the if-then-else statement consists of a sequence of Java `Statements`. Algorithm 6.1 is applied to each of the branches. One `Begin` task is added to the `Decision` to fork the n branches and one `End` task to join them. If $n = 1$, an additional empty branch from `Begin` to `End` is inserted using a `ControlFlow`. The created `ControlFlows` are assigned to the `Decision’s flows` attribute.

MR-20: *Sequence*(*JAVA* : *Statement*) $\rightarrow$ *P4A* : *InternalTask*

Body: A sequence of Java `Statements` that cannot be mapped to a certain PIM4AGENTS `Task` or `StructuredActivity` is mapped to an `InternalTask`. An `InternalTask` is a black box that encapsulates business logic. All variables that are accessed by the enclosed code are added as input/output parameters to the `InternalTask` (see MR-25).

MR-21: $\mathit{Sequence}(\mathit{P4A} : \mathit{Activity}) \rightarrow \mathit{Set}(\mathit{P4A} : \mathit{ControlFlow})$

Pre: $self -> size() > 1

Body: The execution order of `Activities` in PIM4AGENTS is defined by `Control-Flows`. This mapping rule links the passed `Activities` in $self according to their order in the source `Sequence` with `ControlFlows`. The result of this mapping rule is the set of created `ControlFlows`.

6.4 Interaction Mappings

Jadex message events (concept `MMessageEvent`) are usually declared by an agent or capability model (they can also be assembled programmatically using Java code). The actual sending/receiving is performed by Java-based behaviors. As Jadex has no explicit representation of interaction protocols, PIM4AGENTS protocols can only be partially extracted. Thus, the protocols have to be manually refined afterwards.

MR-22: $\mathit{BDI} : \mathit{MCapability} \rightarrow \mathit{P4A} : \mathit{ProtocolConfiguration}$

Pre: $self.messageevent -> size() > 0

Body: This rule maps a Jadex `MCapability` to a PIM4AGENTS `ProtocolConfi-guration`. It is important to note that this mapping rule also applies to the concept `MBDIAgent` (`MBDIAgent` inherits from `MCapability`). Each declared `MMessageEvent` is mapped by MR-23 to a PIM4AGENTS `Message` and assigned to the `ProtocolCon-figuration`.

MR-23: $\mathit{BDI} : \mathit{MMessageEvent} \rightarrow \mathit{P4A} : \mathit{Message}$

Body: The name of the `MMessageEvent` is assigned to the PIM4AGENTS `Message`. If the `MMessageEvent` conforms to a FIPA message, the `content` parameter is used to determine the content type. We propose to use an already existing PIM4AGENTS interaction protocol from a model repository to manually refine the extracted PIM4AGENTS `Messages` (see discussion in Section 6.6).

6.5 Data Model Mappings

As discussed in Section 3.3.3, the Bochica data model is based on Ecore. There are two possibilities how to use the data model in the reverse engineering approach: (i) the data model can be automatically extracted using the mapping rules presented in this section and (ii) it can be manually created for small projects using the Ecore modeling tool (see also [Steinberg et al., 2008]). In the following, the mapping rules for the data model aspect are introduced. As part of Section 5.5, Table 5.5 already defined basic type mappings from Bochica to Java which also apply to the reverse transformation (in inverse direction). If no data type definition can be found in the Java model (e.g. because it is part of an external library that is not accessible to the model transformation), the type strings that are used within the Jadex PSM are mapped to placeholders (see MR-27).

MR-24: $BDI : MParameter \rightarrow P4A : Knowledge$

Body: The Jadex `MParameter` concept specifies the parameters of events and plans. In Pim4Agents, an `MParameter` is mapped to a Pim4Agents `Knowledge`. The name and type of the `MParameter` are mapped one-to-one to the `Knowledge's` name and type (see MR-27). The parameter direction (`in`, `out`, `inout`) is assigned to the `Knowledge` accordingly.

MR-25: $JAVA : VariableDeclarationStatement \rightarrow P4A : Knowledge$

Body: A `VariableDeclarationStatement` is used by a Java-based plan body to declare a local variable. The `VariableDeclarationStatement` is mapped to a Pim4Agents `Knowledge`. The variable name is assigned as the `Knowledge's` name. The data type is mapped by MR-26.

MR-26a/b: $JAVA : AbstractTypeDeclaration \rightarrow ECORE : EClass, P4A : EType$

Body: This rule maps a Java class to (a) an Ecore `EClass` and (b) an according Pim4Agents `EType` that makes the `EClass` available within the Pim4Agents model. The `EClass's instanceClassName` attribute holds the fully qualified name of the Java class. Moreover, the attributes (concept `VariableDeclarationStatement`) and methods (concept `AbstractMethodDeclaration`) are mapped to `EAttributes` and `EOperations` of the Ecore `EClass`.

MR-27a/b: $String \rightarrow ECORE : EClass, P4A : EType$

Pre: fully qualified Java class name

Body: If a data type used by the source model cannot be resolved within the Jadex Java model (see MR-26), this mapping rule is responsible for mapping the fully qualified Java type string to a placeholder `EClass`. The `EClass` uses its `instanceClassName` attribute to store the type information. For every `EClass`, an according PIM4AGENTS `EType` is created that imports the `EClass` into the PIM4AGENTS model.

6.6 Summary

This chapter introduced a novel reverse engineering approach for BDI agents. A model-driven approach was chosen to extract the underlying design of an already implemented Jadex application to the platform-independent BOCHICA core DSL. In order to extract the underlying design, we specified conceptual mappings from the Jadex PSM to the BOCHICA core DSL. The design extraction is done in two steps. First, the platform artifacts are lifted by a model extractor to the Jadex PSM. Afterwards, the conceptual mappings to BOCHICA are applied. It is important to note that the forward transformation presented in Chapter 5 and the reverse transformation of this chapter are not isomorphic – meaning, the reverse transformation followed by the forward transformation (or vice versa) does not produce the exact same model/code. One of the main problems is that there exist many different “flavours” of how to implement a Jadex application. Thus, it is hard to specify generic mapping rules which cover all variants. Moreover, there also exist conceptual mismatches (see Table 6.4). The mismatches are caused by (i) high-level concepts that are not represented explicitly at the platform, (ii) concepts that have a slightly different semantics, and (iii) low-level details that are abstracted by BOCHICA. For example, Jadex has no explicit representation of organizational structures and interaction protocols. Moreover, there exists no concept for knowledge bases (the integration is left open to the developer). Jadex provides the possibility to configure platform services (e.g. messaging or directory services). Those services are not explicitly considered by BOCHICA and setup by the Jadex forward transformation. Furthermore, business logic (e.g. an algorithm implemented in Java) is abstracted by the BOCHICA core DSL as black box. Likewise, expressions (e.g. Java conditions) are currently mapped as plain strings. An

Jadex	Pim4Agents
–	Organization
–	DomainRole
–	Interaction
–	KnowledgeBase
Platform Service	–
Business Logic	–
Expressions	–
Fine-grained Goal Semantics	–

Table 6.4: Conceptual mismatches between Jadex and the Bochica core DSL.

ideal solution would be to create a Jadex extension for Bochica that enables Bochica for native Jadex expressions. Finally, Jadex and Bochica share the same four goal types. However, Jadex provides more fine-grained control over the goal life cycle. As agent technology is getting applied more widely in main stream software engineering and the number of legacy agent systems increases, reverse engineering of agent-based systems is getting increasingly important. The conceptual mappings presented in this chapter were implemented as a QVT-based model transformation. Section 8.2 evaluates the reverse engineering approach in the IRL case study. The presented approach drastically increases the number of available models for model-driven AOSE and Bochica. Moreover, it lays the foundation for model-driven refactoring of existing applications and roundtrip engineering.

Chapter 7

An Extension Model for Agents in Semantically-enhanced Virtual Worlds

Virtual worlds play an increasingly important role for many application domains. Besides entertainment, they are used for serious applications like digital engineering and for training employees in virtual environments before a product or plant has been actually built. As of today, realistic and flexible behavior of agents in virtual worlds is usually simulated by triggered script sequences which create the illusion of intelligent behavior for the user. In the research project *Intelligent Simulated Realities* (ISReal), the DFKI *Agents and Simulated Realities* (ASR) research group developed a simulation platform based on Semantic Web technology for bringing intelligent behavior into virtual worlds [Kapahnke et al., 2010]. The basic idea of ISReal was to use Semantic Web technology to extend geometric objects with ontological information and specify their functionality by semantic service descriptions, called *object services* (see Figure 7.1). Intelligent agents perceive this information, store it in their knowledge base, and use it for reasoning and planning. Thus, agents can interact much more flexible with their environment. The design of intelligent ISReal agents requires the combination of concepts and techniques developed by the Semantic Web community, computer graphics, and AOSE. In the following, an extension model for intelligent agents in semantically-enhanced virtual worlds is presented. The extension model complements Bochica with ISReal-specific concepts (e.g. for orchestrating object services and Semantic Web reasoning languages). The domain-specific extensions enable an engineer to address concepts specific to ISReal without considering low-level technical details. This also reduces the need for manual customizations at code level and prevents that design and code diverge over time. In ISReal, I was responsible for the con-

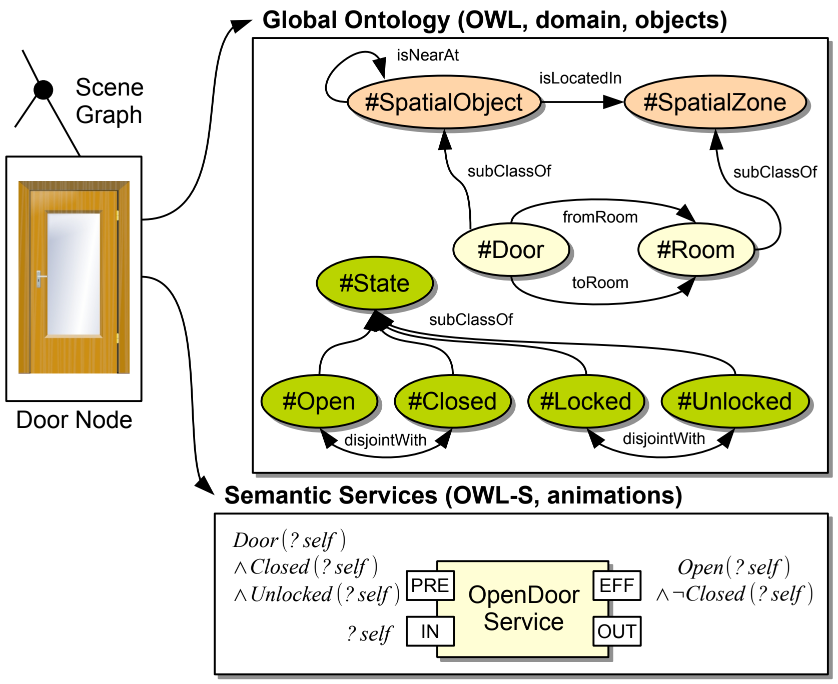


Figure 7.1: The left hand side depicts the geometry of a door as part of the scene graph. A scene graph object is annotated with (i) an ontological concept and object URI, as well as (ii) the offered object services. The right hand side depicts a part of the global ontology for the concept `Door`. The lower part shows the OpenDoorService object service. The precondition has to be entailed by the global world state in order to execute the service. The service is grounded in animation scripts. The effects are updated during service execution.

ceptual design and realization of the agent-specific aspects. In the remainder of this chapter, Section 7.1 provides an overview of the ISReal platform and Section 7.2 discusses the properties of intelligent ISReal agents. Afterwards, Section 7.3 specifies the ISReal extension model for Bochica and Section 7.4 discusses the integration of external reasoning languages. The ISReal extension transformation is presented in Section 7.5. Finally, Section 7.6 provides an overview of the notation of the ISReal concepts.

7.1 The ISReal Platform

The ISReal platform provides the infrastructure for combining different types of simulation components for deploying simulated realities. Currently, the ISReal platform provides interface definitions for four different types of simulation components:

Global Semantic Environment. The *Global Semantic Environment* (GSE) maintains the global ontological facts of the virtual world. It is responsible for (i) executing object services (e.g. checking the precondition and invoking the service grounding), (ii) updating facts (e.g. when a door gets closed), and (iii) handling queries to the global world state.

Agent Environment. The ISReal agent environment defines interfaces for connecting 3rd-party agent execution platforms to the ISReal platform. This includes infrastructure for perception handling and service execution. Every ISReal agent is equipped with a *Local Semantic Environment* (LSE) which is an agent's local knowledge base. The LSE stores the perceived information and enables the agent to reason about it. Moreover, the LSE is equipped with a *Service Composition Planner* (SCP).

Graphics Environment. The user interface of the ISReal platform is realized by an XML3D[63]-enabled standard Web browser [Sons et al., 2010]. The 3D scene graph is part of the browser's *Document Object Model* (DOM) and can be manipulated using Java Script. The semantic annotation of 3D objects is realized using RDFa [W3C, 2008a]. Moreover, the graphics environment runs the agent sensors. The sensors enable agents to perceive the annotated 3D objects.

Verification Environment. The verification environment is used to verify properties of the scene objects (e.g. whether there exists a failure state for a certain configuration of a machine). The verification environment is not considered by this dissertation.

7.2 Intelligent ISReal Agents

An intelligent ISReal agent consists of (i) the body geometry and animations, (ii) semantic annotations, (iii) a sensor component, (iv) the agent that processes the perceptions and controls the body, and (v) an OWL-based knowledge base (see Figure 7.2). An agent's body geometry is part of the scene graph like any other annotated object. The geometry and animations are developed using state-of-the-art 3D modeling tools like Cinema 4D[64] or Blender[65]. The *Semantic Annotation*[66] (SA) Tool is an XML3D-based tool for semantically annotating 3D objects. It has been developed as part of the ISReal project. The agent that controls the body is developed using BOCHICA and the ISReal extension model presented in

[63]`http://www.xml3d.org`
[64]`http://www.maxon.net/products/cinema-4d-studio/`
[65]`http://www.blender.org`
[66]`http://www.dfki.de/isreal/xml3dsat/xml3dAnnotation.xhtml`

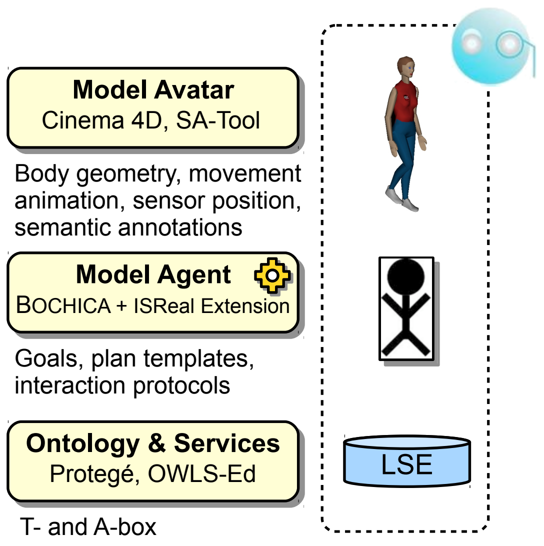

Model Avatar
Cinema 4D, SA-Tool
Body geometry, movement animation, sensor position, semantic annotations
Model Agent
BOCHICA + ISReal Extension
Goals, plan templates, interaction protocols
Ontology & Services
Protegé, OWLS-Ed
T- and A-box
LSE

Figure 7.2: This figure depicts the different artifacts that encompass an ISReal agent configuration and the according tools.

this chapter. Finally, Semantic Web tools like Protegé[67] are used to model the ontologies and semantic service descriptions. Before we introduce the extension model for ISReal agents, some more details about the ISReal agent architecture are discussed. The background information is required to derive the requirements for creating the ISReal extension model.

ISReal Agent Architecture

Figure 7.3 depicts an architectural pattern for ISReal agents. It is independent of a concrete agent platform. In order to enable a 3rd-party agent platform to host ISReal agents, the pattern has to be realized for that platform. As part of the ISReal project, this has been done for the BDI agent platforms Jack and Jadex. The LSE equips an agent with (i) an OWL-based knowledge base, (ii) a Semantic Web reasoner, and (iii) a *Service Composition Planner* (SCP). The remainder of this section provides an overview of the different components of the ISReal agent architecture.

Behavior Component. Intelligent ISReal agents are autonomous entities which are situated in a 3D scene, use perception-based service discovery, and orchestrate the object services of their environment in order to achieve their design objectives. The orchestration can be done (i) by an agent developer at design-time by modeling behavior templates or (ii) during runtime by the SCP. The behavior component provides interfaces for invoking the SCP and executing object

[67]`http://protege.stanford.edu`

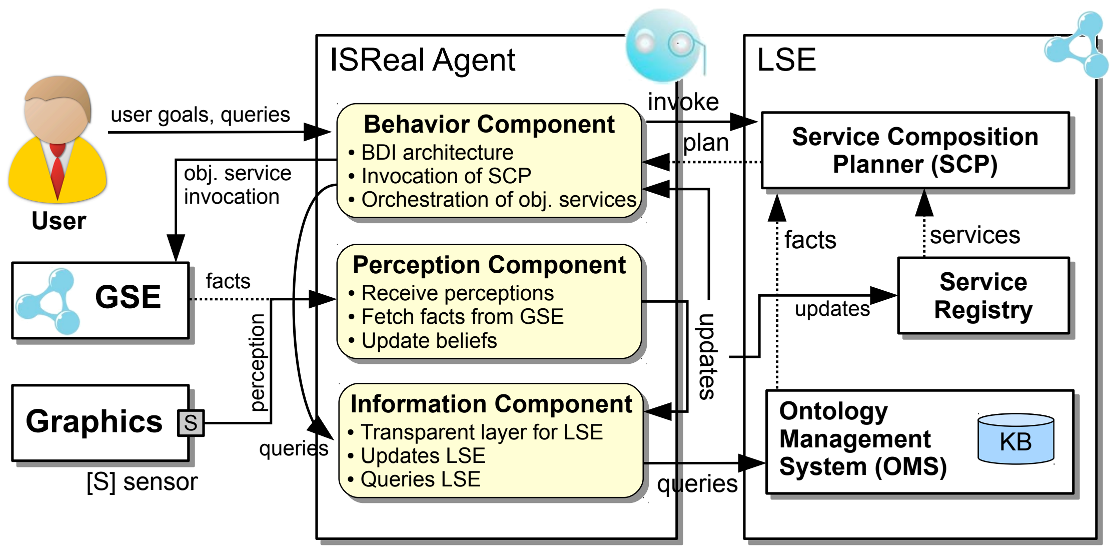


Figure 7.3: This figure depicts an architectural design pattern for ISReal agents. The components are connected by control flows (continuous lines) and information flows (dotted lines).

services. The sequence diagram depicted by Figure 7.4 visualizes an object service invocation. First, the agent checks whether the service's precondition holds in its LSE. Afterwards, the agent invokes the service by passing the service *Unified Resource Identifier* (URI) and according parameter bindings to the GSE. The GSE checks whether the precondition holds globally. If so, it invokes the according service grounding which performs simulation and animation calls. Finally, the effects of the service are updated by the GSE. The effects are *not* delivered to the agent. The agent uses its sensor to indirectly perceive the effects. After the agent perceived the effects, it requests the according facts from the GSE and updates its LSE. In order to prevent the agent from deadlocks, timeout t_e restricts the total execution time. If the effects are not perceived in time, the agent assumes the service call to be failed.

Perception Component. An intelligent ISReal agent's sensor is maintained by the graphics environment. As part of the ISReal project, XML3D has been extended with a sensor node that shoots rays into the scene to compute the line of sight (parameters: resolution, opening angle, and update rate). The sensor causes perception events which are handled by the perception component of the agent. A perception event contains the object's ID in the scene graph and the semantic annotations of that object. Figure 7.5 visualizes the perception delivery process of the ISReal platform. As an object o enters the sensor area s_a of an agent a, it causes a perception event for a. Moreover, the tuple (a, o) is added to the GSE's *visibility list*. The visibility list is necessary to notify a when the ontological facts

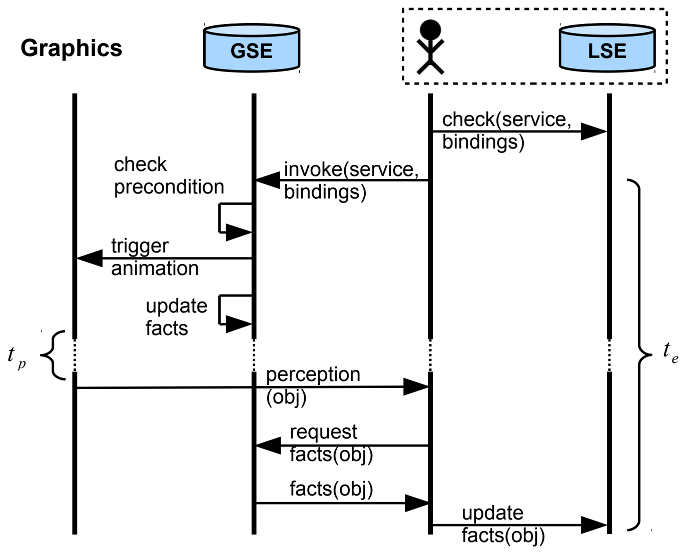


Figure 7.4: This sequence diagram visualizes the execution of object services. t_e is the total time between the service invocation and the perception of the effects and t_p is the time that passes between setting the facts in the GSE and perceiving the facts by the agent.

about o change while o is already in the line of sight. The underlying problem is that the global world state is distributed over the scene graph (geometry) and the GSE (semantic facts). Thus, semantic changes might only be recognized by the GSE – not by the sensor observing the geometry (e.g. switching a machine on). The agent's perception component proceeds with fetching the facts about o from the GSE and updating the agent's LSE. If the facts about an object change (e.g. as part of a service execution), the GSE notifies all agents that are registered at the visibility list for o. Finally, as an object moves out of the sensor area, the sensor notifies the agent and the GSE. The GSE removes the entry (a, o) from the visibility list. It is important to note that (i) agents can overlook effects when the affected object is not in the line of sight and (ii) an agent's LSE might get inconsistent with the GSE over time. Those properties are design decisions in order to simulate realistic perception. This causes high requirements towards the design of the agent who has to flexibly interact with the environment it is located in.

Information Component. In order to make rational decisions, intelligent ISReal agents reason about the beliefs stored in their LSE. For example, context conditions and preconditions of plans have to be evaluated in the LSE. Moreover, the target conditions of goals are evaluated in the LSE to decide whether an agent assumes its goals to be achieved. The information component provides the interfaces for integrating the LSE into a 3rd-party agent platform. Moreover,

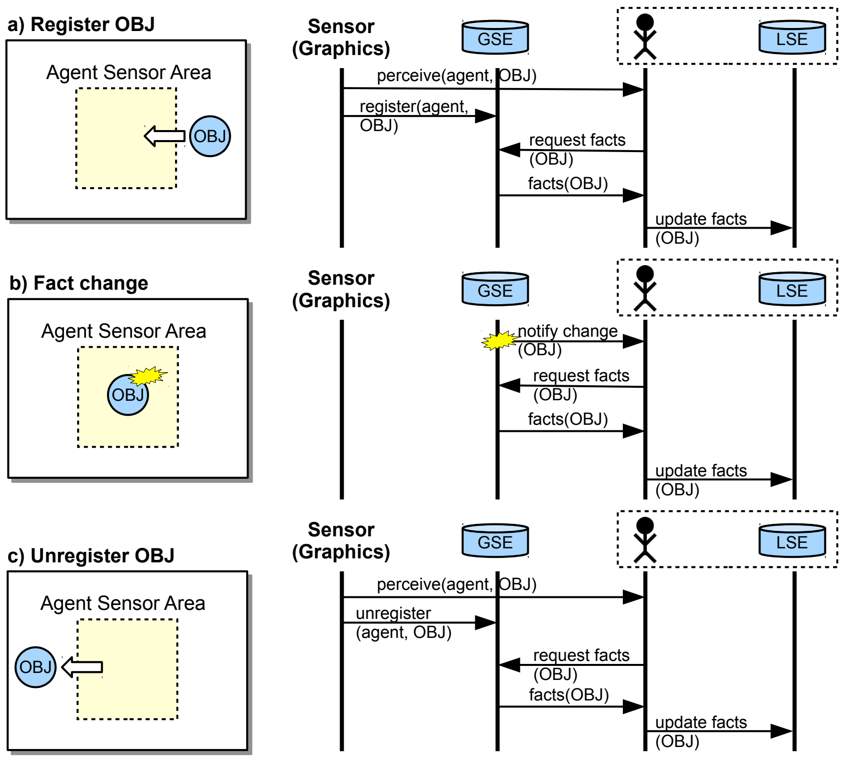


Figure 7.5: This figure visualizes the perception delivery mechanism of the ISReal platform.

it provides basic operations for processing perception data and handling queries to the LSE. In ISReal, the T-box (ontological types) is assumed to be globally known and static. However, an agent's A-box (object knowledge) evolves as new individuals are discovered.

7.3 Conceptual Extension

As introduced in Chapter 3, the core DSL underlying the Bochica framework covers the core concepts of MAS. Modeling ISReal agents using just the Bochica core DSL is possible, but many details specific to agents in semantically-enhanced virtual worlds cannot be captured since the according concepts are missing. For example, an ISReal agent's LSE is based on Semantic Web technology, the agent's body is a geometrical and annotated object in the scene graph, and the sensor has ISReal-specific properties. By extending Bochica, large parts of the infrastructure for modeling MAS can be reused. Moreover, Bochica enables the separation between MAS design and code. Figure 7.6 depicts the big picture of the Bochica framework for developing ISReal agents. In the following, we introduce the extension model which extends Bochica with ISReal-specific concepts.

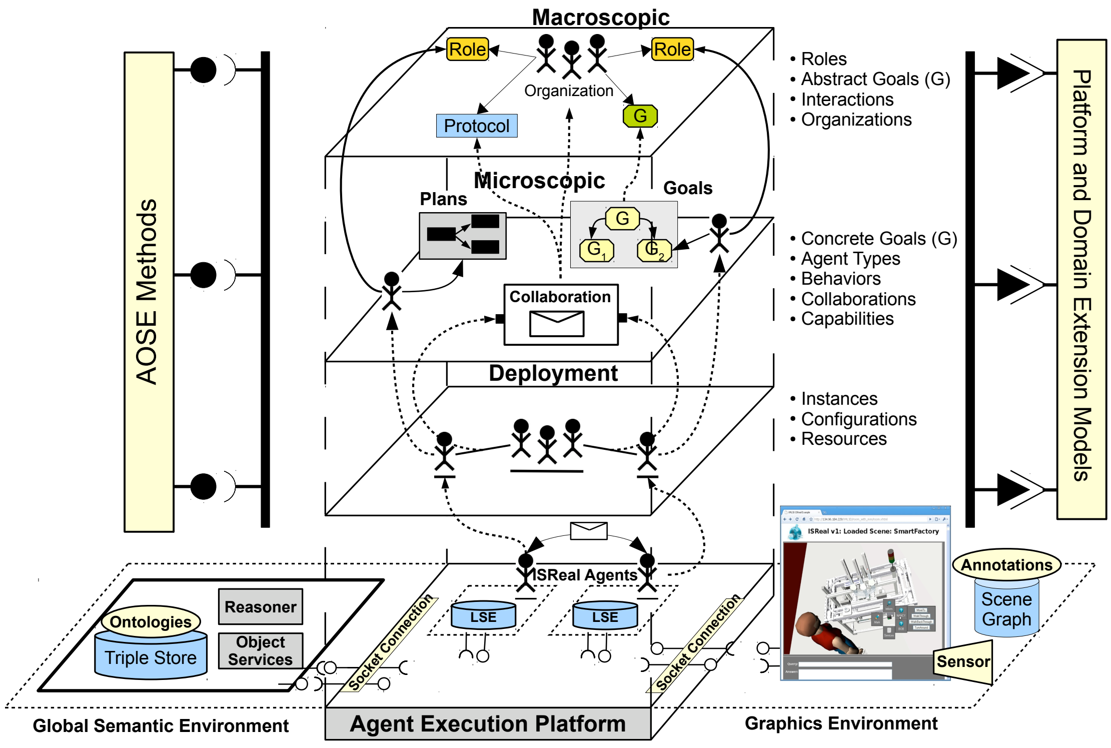


Figure 7.6: The bottom layer depicts the components of the ISReal platform. The upper central part shows the inherent degrees of abstraction of Bochica. The left an right hand sides represent the interfaces for extending the framework.

Section 7.2 already provided an overview of the components that encompass an ISReal agent. All parts have to be integrated into one consistent ISReal agent. Figure 7.7 depicts the ISReal extension model for Bochica. In order to link a Bochica `Agent` to its geometrical body, the `ISRealAgent` concept provides an URI. Moreover, the `ISRealAgent` has (like every 3D object in ISReal) an ontological concept defined by an URI. The `ISRealRaySensor` concept specializes the Bochica `Sensor` concept for ISReal. For example, it provides additional properties like resolution, opening angle, and update rate. An intelligent ISReal agent's knowledge base is based on Semantic Web technology. For this purpose, the `LocalSE` concept specializes the `KnowledgeBase` concept for ISReal. The ISReal platform already provides a metamodel for configuring the LSE, called *Ontology Management System Configuration* (OMSConfig). The `OMSConfig` metamodel is used for configuring ontologies, object services, and other parameters of the LSE. The extension model imports the `OMSConfig` concept of the ISReal platform. The `OntologyFile` concept is used to refer to ontologies (e.g. A- and T-box, as well as semantic service descriptions). Bochica already provides support for orchestrating traditional Web services by plans (see Section 4.1). Since ISReal object services

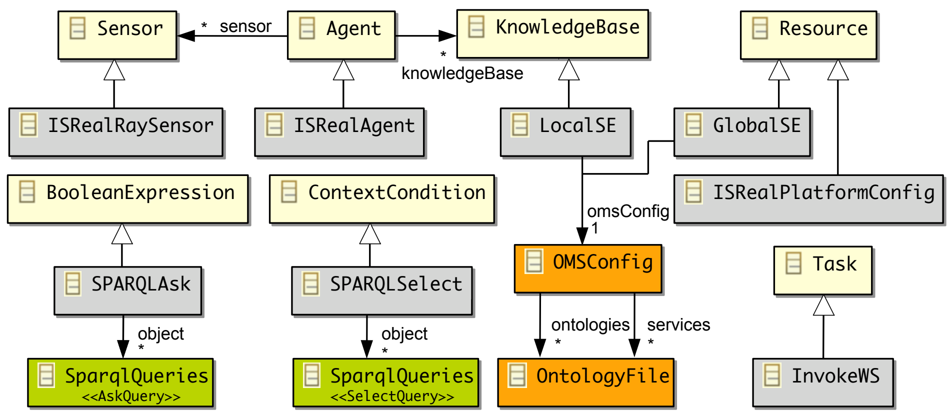


Figure 7.7: This figure depicts the main concepts of the ISReal extension model. Four types of concepts are shown: (i) interface concepts of Bochica (yellow), (ii) concepts of the ISReal extension model (gray), (iii) concepts of the ISReal platform (orange), and (iv) concepts of the SPARQL metamodel (green).

need similar parameters to ordinary Web services, the existing `InvokeWS` concept can be reused. However, the technical meaning of the concept changes during code generation. Instead of generating code for the invocation of a traditional Web service, the generated code invokes an ISReal object service of the GSE (see Section 7.5). Likewise, messaging between agents causes an additional logging event to the ISReal platform (e.g. for the visualization of negotiations). The integration of Semantic Web reasoning languages is discussed in the succeeding section.

7.4 Language Extension

In order to make rational decisions, it is essential for intelligent ISReal agents to reason about their beliefs. GSE and LSE use the *Resource Description Framework* (RDF) [W3C, 2004a] for knowledge representation. The interface to a knowledge base is usually defined by a query language. The *SPARQL Protocol And RDF Query Language* (SPARQL) [W3C, 2008b] is a widely supported query language for RDF graphs. GSE and LSE use SPARQL as query interface. There exist different kinds of SPARQL queries. A SPARQK-Ask query returns a boolean value. The result depends on whether a variable binding is found or not ($|\ result\ | > 0$). A SPARQL-Select query returns a set of variable bindings for the unbound variables of the query. In the following, we discuss the integration of SPARQL into Bochica.

As explained in Section 3.3.2, Bochica defines language interface concepts for integrating 3rd-party software languages. One requirement for integrating external software language is that the language specification is based on Ecore. The

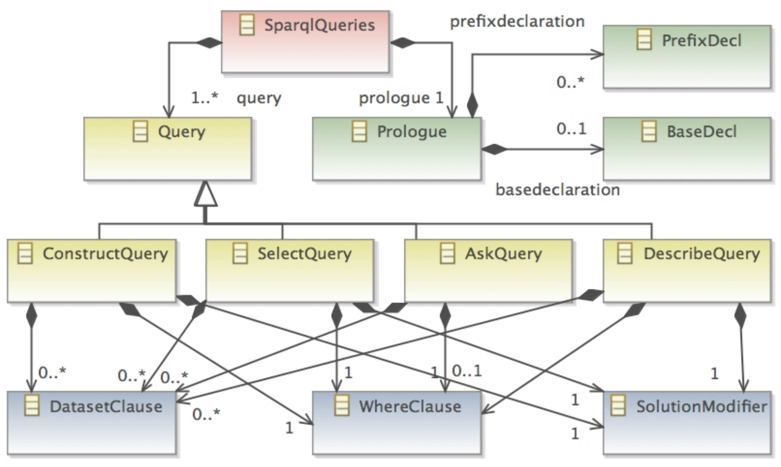


Figure 7.8: The main concepts of the EMFText SPARQL metamodel.

EMFText concrete syntax zoo already provides an Ecore-based SPARQL DSL[68] which we reused for the ISReal extension model. Figure 7.8 depicts the EMF-Text SPARQL metamodel. `SparqlQueries` is the root concept of the metamodel. It consists of a `Prologue` which defines name spaces and the actual query object. The four concepts `ConstructQuery`, `SelectQuery`, `AskQuery`, and `DescribeQuery` represent the four query types of SPARQL. The Bochica `BooleanExpression` has been extended with SPARQL-Ask and the `ContextCondition` with SPARQL-Select (see Figure 7.7). The automatically generated parser of EMFText is used for parsing textual SPARQL queries into SPARQL query models that are plugged into the Bochica extension slot (see Section 3.3.2). EMFText also provides a SPARQL editor with syntax highlighting for convenient query editing.

7.5 ISReal Extension Transformation

Section 3.4 introduced a forward transformation architecture which is tailored to the needs of Bochica. Base transformations map the concepts of the Bochica core DSL to platform artifacts. Extension transformations complement the base transformation with additional conceptual mappings for a certain execution environment. In the following, we consider Jadex as target platform for implementing intelligent ISReal agents. The conceptual mappings of the Jadex base transformation presented in Chapter 5 are taken as basis for the ISReal extension transformation. The remainder of this section introduces additional conceptual mappings for the ISReal extension model. Figure 7.9 depicts an overview of the ISReal-specific transformation architecture. The base transformation from Bochica to

[68]`http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_SPARQL`

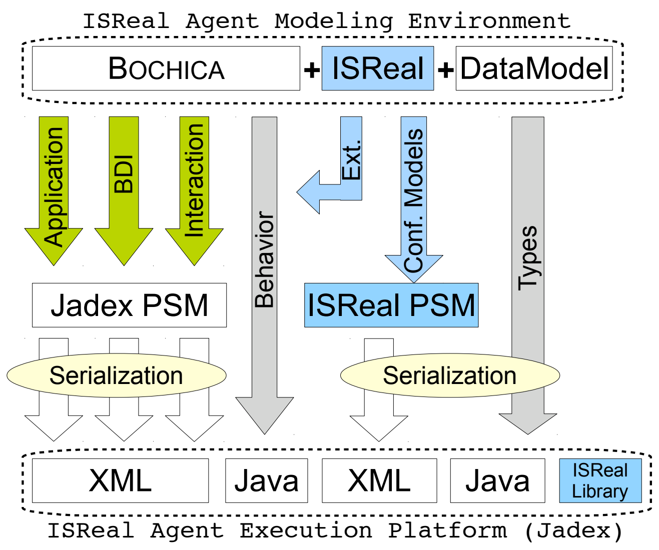


Figure 7.9: ISReal (Jadex) transformation overview. The left hand side depicts the Jadex base transformations. The blue part depicts the ISReal extension transformation.

Jadex consists of the four modules *application*, *BDI*, *interaction*, and *behavior* (see Chapter 5). The blue parts in Figure 7.9 depict (i) the ISReal extension model, (ii) the ISReal extension to the Jadex base transformation, (iii) the generation of ISReal configuration files (e.g. ISReal PSM), and (iv) an additional ISReal library that enables Jadex for the ISReal platform. The ISReal library implements the interfaces of the ISReal agent environment as presented in Section 7.2 for the Jadex platform (e.g. it equips a Jadex agent with an LSE). Moreover, it includes Jadex into the start-up procedure of the distributed ISReal platform. In the following, we provide an overview of the extension transformation. The mapping rules have the same structure as those of Chapters 5 and 6. Concepts of the ISReal extension model have the prefix `ISREAL`. The ISReal mapping rules use the two keywords **extends** and **replace** to refer to mapping rules of the Jadex base transformation. **Extends** means that the original mapping rule is modified, whereas **replace** substitutes the original mapping rule by the ISReal-specific one. Mapping rules of the Jadex base transformation are referenced by MR^{B}-X.

MR-1: *ISREAL* : *ISRealAgent* → *JADEX* : *MBDIAgent* **extends** MR^{B}-4

Body: This mapping rule extends MR^{B}-4 for mapping an `ISRealAgent` agent to Jadex. The generated code implements the ISReal agent architecture as depicted in Figure 7.3. For example, every Jadex ISReal agent is equipped by default with an ISReal capability that provides access to the ISReal library (e.g. the LSE).

Additional beliefs for storing the semantic concept and object URIs, the reference to the geometrical body, the sensor parameters, and information for connecting to the ISReal platform are generated. Moreover, for every `PlanUse` (MR^B-7) and `CapabilityUse` (MR^B-6), additional parameter bindings are generated that make the ISReal capability (e.g. the LSE) accessible within the used plans and capabilities of an agent (e.g. to evaluate the precondition of a plan in the LSE).

MR-2: *ISREAL* : *SPARQLAsk* → *JADEX* : *MStaticValue* **extends** MR^B-39a

Pre: $self.eContainer.oclIsTypeOf(P4A:Plan)

Body: The SPARQL-Ask precondition of a Bochica `Plan` is mapped to a precondition (concept `MStaticValue`) of an `MPlan` in Jadex. Wrapper code for evaluating the query in the LSE is generated (Section 8.1.2 provides an example). The Jadex ISReal library provides a transparent layer for evaluating the SPARQL-Ask query in the LSE. The model transformation also resolves the variable symbols of the query in the surrounding scope of the expression (see Figure 7.10 a). For example, assuming that *v1* is a variable defined in the surrounding scope (e.g. the belief base) that should be used as parameter in a plan's SPARQL-Ask precondition, the mapping rule first checks whether the `MPlan` declares such a parameter. If not, it is tried to resolve the symbol in the parameters of the triggering event (concept `MPlanTrigger`). Finally, the `MBeliefbase` is checked. If no match can be found, the variable stays unbound. It is also possible to explicitly define the scope by using variable prefixes as depicted in Figure 7.10.

MR-3: *ISREAL* : *SPARQLAsk* → *CODE* : *Expression* **extends** MR^B-39c

Pre: $self.eContainer.oclIsTypeOf(P4A:ControlFlow)

Body: A SPARQL-Ask condition contained by a `ControlFlow` is mapped to Java code that invokes the LSE. The ISReal library provides the according interface for evaluating the query in the LSE. For example, this mapping rule is used to generate the condition of an if-then-else expression (concept `Decision` in Pim4Agents). The variable symbols are resolved using the `getVariable()` method of concept `Activity` (discussed in Section 3.3.1).

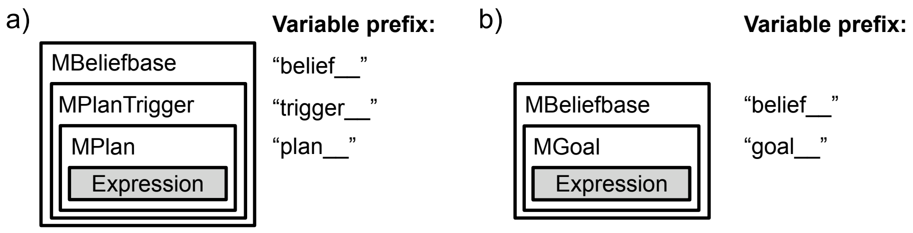


Figure 7.10: Scopes for resolving variable symbols of SPARQL expressions. a) depicts the scopes for a precondition or context condition of a plan and b) the scopes for a target condition of a goal. The variable prefixes can be used to explicitly address the variables of a certain scope.

MR-4: *ISREAL* : *SPARQLSelect* $\rightarrow$ *JADEX* : *MInternalCondition* **extends** MR^{B}-39b

Pre: \$self.eContainer.oclIsTypeOf(P4A:Plan)

Body: The SPARQL-Select context condition of a BOCHICA `Plan` is mapped to an `MPlanParameter` in Jadex. The differences between BOCHICA and Jadex have been discussed in Section 5.3. The SPARQL-Select query returns a collection of possible variable binding records. Each record contains an assignment for the query's unbound variables. The collection is used to initialize the binding options of the `MPlanParameter`. At runtime, Jadex generates for each record of the collection an own plan candidate. As a plan candidate gets selected for execution, the according record is used to access the values of the single variables (according code is generated automatically). Thus, the SPARQL-Select query seamlessly integrates into Jadex and can be used to initialize plan candidates in Jadex. To evaluate the query, the Jadex ISReal library provides a wrapper for passing SPARQL-Select to the LSE (similar to SPARQL-Ask queries). Figure 7.10 a) shows how the variable symbols are resolved in the surrounding scope.

MR-5: *ISREAL* : *SPARQLAsk* $\rightarrow$ *JADEX* : *MInternalCondition* **extends** MR^{B}-39b

Pre: \$self.eContainer.oclIsTypeOf(P4A:ConcreteGoal)

Body: SPARQL-Ask queries are used to define the target and maintain conditions of `ConcreteGoals` in ISReal. Similar to the previous mapping rules, the generated code passes the SPARQL-Ask query to the Jadex ISReal capability. Figure 7.10 b) depicts how the variable symbols are resolved in the surrounding scope.

MR-6: *P4A* : *InvokeWS* → *CODE* : *OSInvocation* **replace** MR^{B}-25

Body: This mapping rule replaces the code generated for the invocation of a traditional Web service with code for the invocation of an object service at the GSE. The service invocation uses the `serviceEndPoint` and `operation` attributes of the `InvokeWS` concept to address the service implementation. Moreover, the `Knowledge` parameters are used as parameters of the service call. The `timeout` attribute specifies the timeout t_e of Figure 7.4. The Jadex ISReal capability provides the interface for calling the object service invocation.

MR-7a/b/c: *ISReal* : *LocalSE*, *ISReal* : *GlobalSE*, *ISReal* : *ISRealPlatformConfig* → *ISREAL* : *PSM*

Body: The concepts `LocalSE`, `GlobalSE`, and `ISRealPlatformConfig` were imported to the ISReal extension model (e.g. for configuring the agent's LSE using OWL ontologies). This mapping rule is an one-to-one mapping of the concepts to the according artifacts of the ISReal PSM.

The mapping rules were implemented using XPand for model-to-text transformations and QVT for model-to-model transformations. Figure 7.11 depicts an XPand-based aspect-oriented mapping rule which replaces the original mapping rule of the Jadex base transformation for invoking a standard Web service by the invocation of an ISReal object service (MR-6). The first part sets the variable bindings of the object service and the second part does the actual invocation through a helper class provided by the ISReal library. Figure 7.12 depicts the QVT-based implementation of MR-5. The call to `self.resolveVariables()` resolves the variable tokens of a SPARQL-Ask query in the surrounding scope so that the proper parameters are passed to the invocation (see Figure 7.10). The `MInternalCondition's text` attribute encapsulates the call to the ISReal library. The LSE is accessed through the Jadex `$ beliefbase.lse` `MBelief`. The SPARQL query, including the resolved variable symbols, is passed to the LSE's `sparqlAsk` method.

7.6 ISReal Notation and Views

The technical details explained so far are (in the ideal case) not visible to the agent engineer. He follows an agent methodology and uses graphical diagrams to design a MAS for a certain use case (e.g. agents operating a virtual production line).

```
«AROUND template::invokews2code FOR pim4agents::InvokeWS»
  wsUri = "«this.serviceEndPoint»#«this.operationName»";
  wsBindings= new BindingListImpl();
  «FOREACH this.incomingParameters AS i-»
     wsBindings.addPair("«this.serviceEndPoint»#«i.name»",
     «resolveKnowledgeValueExpression(i.value.trim(), i, this)»);
  «ENDFOREACH-»

  OSInvocation w = new OSInvocation((ISRealAgent) this
    .getBeliefbase().getBelief("isrealAgent").getFact(), wsUri,
     wsBindings, (long) «this.timeout * 1000»);
  this.waitForExternalCondition(w);
  wsRes = w.getResult();
  if(!wsRes) this.fail();
«ENDAROUND»
```

Figure 7.11: This aspect-oriented mapping implements MR-6. It replaces a standard Web service invocation by the invocation of an ISReal object service.

```
transformation
  PIM4Agents2ISRealAgent(in p4a : P4A, out bdi : JADEX)
   extends PIM4Agents2JadexAgent;

main() {
  p4a.objects()[P4A::multiagentsystem::MultiagentSystem]
    .map toJadexAgentRoot(); }

mapping ISREAL::SparqlAsk::toISRealSPARQLAsk() :
  MInternalCondition {
    var cnd : String := self.resolveVariables();
    text := "$beliefbase.lse.sparqlAsk(" + cndr +
             ", $beliefbase.lseUpdateProxy)"; }
```

Figure 7.12: This figure depicts an example QVT-based extension transformation (agent module). The *transformation* and *extends* keywords refer to the Jadex base transformation. The `toISRealSPARQLAsk` rule implements MR-5.

The graphical front-end abstracts from technical details such as (i) the integration of Jadex into the ISReal platform, (ii) the invocation of ISReal object services in Jadex, or (iii) the evaluation of SPARQL queries in the LSE. Custom views are created to show new aspect or to show existing ones in a different context. Table 7.1 depicts an overview of the concrete syntax of the ISReal-specific concepts introduced in Section 7.3. Two new diagram types have been created:

- **ISReal Agent Diagram:** The ISReal agent diagram extends the Bochica agent diagram with ISReal-specific model elements like the `ISRealRaySensor` and the `LocalSE` (see Figure 7.13). Moreover, it uses the ISReal notation.

- **ISReal Platform Diagram:** The ISReal platform diagram enables the modeler to configure the ISReal platform. For example, the ontologies of the GSE and the components of the ISReal platform can be configured.

Figure 7.13 depicts the ISReal agent diagram. Placing an ISReal agent into the diagram implies, compared to a plain Bochica agent, the generation of an

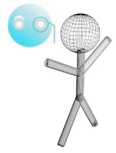
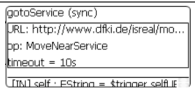
Notation	Concept	Description
	ISRealAgent	• Concept URI • Avatar URI
	LocalSE	• Local ontologies • Local object services • Configuration
	GlobalSE	• Global ontologies • Global object services • Configuration
	ISRealPlatformConf	• Configuration of platform components • Start-up configuration
✧ ISRealSensor [50 x 20] : 30ms	ISRealRaySensor	• Resoluation • Update rate
 gotoService (sync) URL: http://www.dfki.de/isreal/mo... op: MoveNearService timeout = 10s [IN] self : EString = $trigger.selfUF	InvokeWS	• Service URI • Parameters • Timeout

Table 7.1: This figure depicts the notation of the ISReal-specific concepts.

ISReal-enabled agent that integrates into the ISReal platform (e.g. the integration of the LSE and support for object service invocations).

7.7 Summary

This chapter introduced an extension model for agents in semantically-enriched virtual worlds. First, an overview of the ISReal platform and intelligent ISReal agents was provided. Afterwards, the conceptual extensions, the extension transformations, and the graphical notation have been introduced. The BOCHICA approach to AOSE has several advantages: (i) BOCHICA already provides the core concepts, diagrams, etc. for modeling agent systems, (ii) the existing base transformation to Jadex can be reused, (iii) only missing aspects have to be extended in order to create an individual development environment for agents in semantically-enhanced virtual worlds, and (iv) it enables the reuse of existing model artifacts (e.g. interaction protocols). The level of abstraction provided by BOCHICA enables an engineer to orchestrate object services without knowing how they are technically incorporated into the agent platform. Moreover, the extension model is not limited to the ISReal-enabled Jadex platform. Our experiences with Jack show that the ISReal models can be used for other ISReal-enabled agent platforms, too. The decision which methodology to apply is left open to the agent engineer who is the

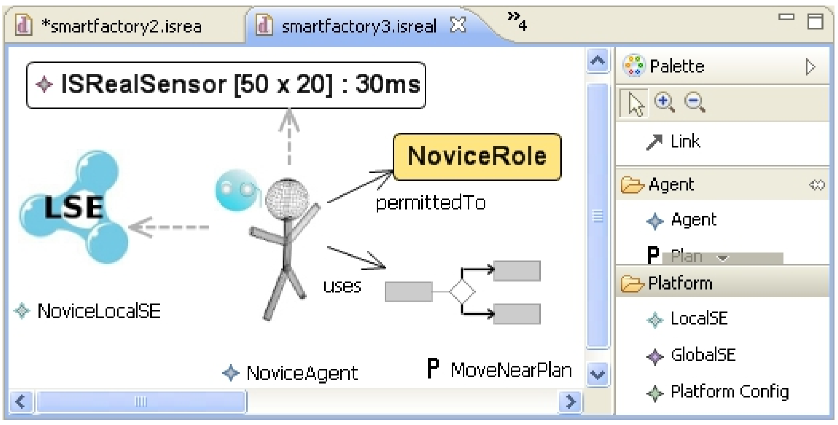


Figure 7.13: The customized ISReal agent diagram.

end-user of the framework. The succeeding chapter evaluates the ISReal-enabled BOCHICA framework in a virtual production line case study.

Chapter 8

Evaluation and Discussion

The previous chapters introduced the Bochica framework for model-driven AOSE. This included the underlying design principles, the alignment to existing development processes, an iterative extension mechanism, and a base transformation to a BDI platform. Moreover, an extension model for agents in semantically-enhanced virtual worlds and a method for model-driven reverse engineering was proposed. The remainder of this chapter evaluates the Bochica framework in two case studies. The SmartFactory case study (Section 8.1) evaluates Bochica and the ISReal extension for capturing the design decisions of a virtual production line scenario and the IRL case study (Section 8.2) evaluates the agent-oriented method for model-driven reverse engineering for extracting the underlying design of an already implemented MAS. Finally, Section 8.3 summarizes this chapter. Mapping rules introduced by previous chapters are referenced as MR^{B}-X for the Jadex base transformation (Chapter 5), MR^{R}-X for the Jadex reverse transformation (Chapter 6), and MR^{E}-X for the IReal extension transformation (Chapter 7).

8.1 SmartFactory Case Study

The aim of the DFKI SmartFactory[69] living lab in Kaiserslautern is to evaluate technology for the factory of the future. One goal of the collaboration between the ISReal project and the SmartFactory was to design intelligent agents that are able to operate a virtual representation of the SmartFactory and perform typical workflows. Such systems can be used to simulate operating sequences before a plant is actually built and also for training employees. Semantically-enriched 3D objects enable intelligent agents to flexibly react to their environment without the need of adapting hard-coded script sequences. Chapter 7 already discussed the

[69]`http://www.smartfactory-kl.de`

capabilities of the ISReal platform and introduced a Bochica extension model for agents in semantically-enhanced virtual worlds. The case study is structured into four parts. First, the ISReal-enabled Bochica framework is applied to the SmartFactory scenario for modeling the behavior of human-like entities (e.g. machines are not considered as intelligent agents in this case study). Afterwards, the Bochica approach is compared to (i) a methodology-oriented approach and (ii) a platform-specific approach. Finally, the results and experiences are summarized and set into context to the related work in Chapter 2.

The development of an ISReal scenario is a complex endeavour which involves different stakeholders such as computer graphics experts (geometry, animations), semantics experts (ontologies, semantic service descriptions), and agent experts (behavior simulation). The objective of this case study is not to go through the entire development process of an ISReal scenario. This is out of scope of this dissertation. Instead, the focus is on the aspects related to the design of intelligent ISReal agents. Following assumptions have been made:

- **Platform Integration.** It is assumed that the BDI agent platforms Jack and Jadex are already enabled for ISReal. In particular, this encompasses the implementation of the ISReal agent architecture depicted in Figure 7.3.
- **Scenario Implementation:** 3D objects, animations, ontologies, and object service implementations are given.
- **Extension Model:** It is assumed that the ISReal extension model presented in Chapter 7 already exists. This assumption is reasonable since the ISReal platform can be used for a wide range of applications and is not specific to the SmartFactory case study. The effort for creating the extension is discussed at the end of the case study.
- **Forward Transformation:** Likewise, it is assumed that the base transformation to Jadex and the ISReal extension transformation already exist.

As representative of the methodology-oriented approaches to AOSE we selected the Prometheus methodology and the according PDT modeling tool. Prometheus has been chosen because it is one of the most elaborated agent-oriented methodologies. It supports the development of BDI agents starting from requirements until implementation. Moreover, PDT provides according modeling support and offers code generation for Jack. INGENIAS and O-MaSE only provide tool support for the Jade platform (which is not a BDI platform). Tropos offers tool support for the BDI platform Jadex but is less mature than Prometheus and has a focus on the early phases of software development.

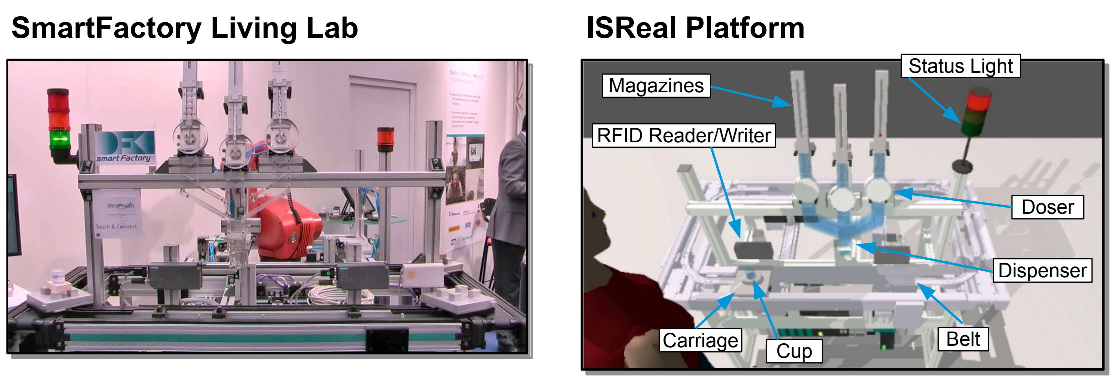


Figure 8.1: The *SmartFactory Fair Module* and its virtual counterpart.

The commercial BDI platform Jack and the according JDE workbench were chosen as representative of the platform-specific approaches. Jack is a very mature platform that is under continuous development for more than ten years. The other platform-specific approaches presented in Section 2.2 either do not support BDI agents or are not mature enough.

The SmartFactory case study is structured as follows: Section 8.1.1 provides an overview of the SmartFactory and the concrete scenario. The Bochica design artifacts of the macroscopic, microscopic, and deployment layer are discussed in Section 8.1.2. Moreover, an overview of the generated code is provided. Afterwards, Section 8.1.3 discusses how to apply the Prometheus methodology to the SmartFactory scenario. Section 8.1.4 shows how the case study has been realized in Jack. Finally, Section 8.1.5 discusses the results and the related work.

8.1.1 The SmartFactory

The virtual machine used in the following scenario is called *SmartFactory Fair Module* and was built by the DFKI SmartFactory living lab in Kaiserslautern (see Figure 8.1). The fair module is a portable version of the SmartFactory which is used for demonstration purposes on fairs. The machine's objective is to fill certain quantities of different types of pills into cups. Each cup carries an *Radio-Frequency IDentification* (RFID) tag which stores an individual order. The fair module is equipped with an RFID reader to process the next order. A stopper halts the carriage below the dispenser. The doser is responsible for retrieving the right quantity of each type of pill according to the current order. In total, the fair module has three pill magazines which store different types of pills. Finally, the dispenser fills the pills into the cup. The machine has a status light that can visualize four states: running (green), error (yellow and red), and off (no light). Figure 8.2 depicts the SmartFactory scenario as considered by this case

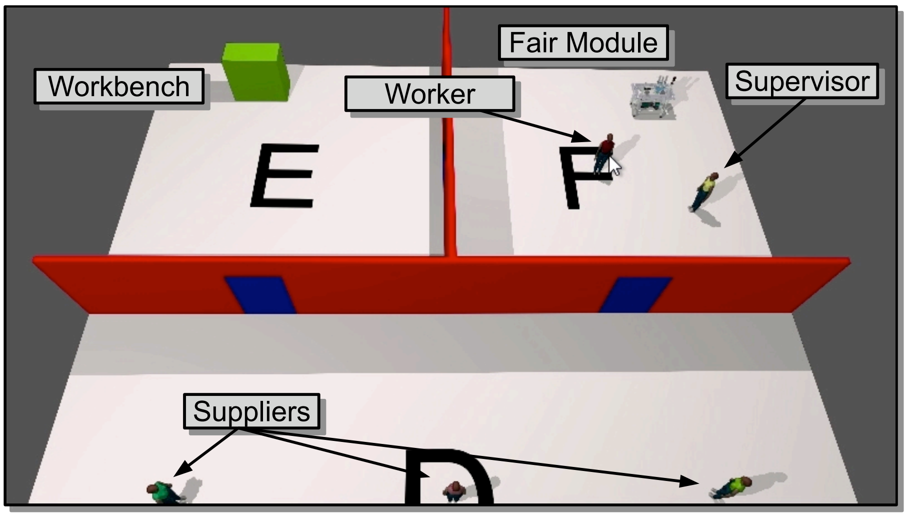


Figure 8.2: This figure depicts the SmartFactory scenario as considered by the case study.

study. It consists of three rooms that are connected by doors. The workbench is an additional machine for the production of new pills. Here follows a list of requirements towards the demonstrator to develop:

- **Production:** The worker is responsible for maintaining the production. This includes (i) to handle new orders and to start the production, (ii) to react on failures of the machine and request help from the supervisor if necessary, and (iii) to organize supplies by negotiating with the suppliers.

- **Supervisor:** The supervisor is responsible for providing help to the worker agent if a problem occurs that cannot be solved.

- **Supplies:** The pharmacy agents have no active role in the scenario. They represent external suppliers and provide the ingredients for the production of new pills on the workbench.

- **Movement:** The agents have to be able to navigate through the virtual environment. This includes (i) movement inside rooms, (ii) across rooms, and (iii) searching for unknown objects.

- **SmartFactory:** The agents need the capability to perform basic tasks at the SmartFactory like (i) switching the machine on/off, (ii) using the RFID writer for writing orders onto cups, (iii) refilling pill magazines, and (iv) using the workbench.

- **Negotiation:** The worker agent negotiates with the suppliers to get the ingredients for producing new pills on the workbench. Likewise, workers can request help from the supervisor.

8.1.2 Bochica

This section provides an overview of the model artifacts of the macroscopic, microscopic, and deployment layer created with Bochica. Moreover, the generated code and implementation-related aspects are discussed.

Macroscopic

According to Section 3.1, the macroscopic layer of Bochica structures the SUC in terms of organizational structures, domain roles, and abstract goals. The three domain roles Production, Supervision, and Supplies have been introduced to represent the responsibilities in the SmartFactory scenario (see Figure 8.3). The SmartFactoryOrganization specifies the organizational structure between the domain roles. The role-fillers of the Production role have the abstract goal to maintain the production, while the agents performing the Supplies role are responsible to provide supplies. Finally, the role-fillers of the Supervision role are responsible to assist the production. In order to specify the communication between the domain roles, the SmartFactoryOrganization makes use of two standard agent interaction protocols. The ContractNet and RequestPresonse protocols are independent of the context they are used in. They are imported from a model repository. The concrete role bindings inside the organization are defined by the microscopic layer. For example, the RequestResponse protocol is used by production agents to request help from agents performing the Supervision role. The ContractNet protocol is used by agents of the Production role to buy ingredients from the suppliers. Figure 8.4 depicts the *Contract Net Protocol* (CNP) [FIPA, 2002c] which is a frequently used negotiation protocol for MAS. The Initiator sends a request-for-proposal message to the role-fillers of the Participant actor. Every participant evaluates the request and either proposes or refuses. The Initiator collects the responses and chooses the best one(s) and sends according notifications to the Participants. Finally, the Participant confirms the completion of the task.

Microscopic

The microscopic layer encompasses collaborations inside organizations, concrete goals, agent types, capabilities, behaviors, and the data model.

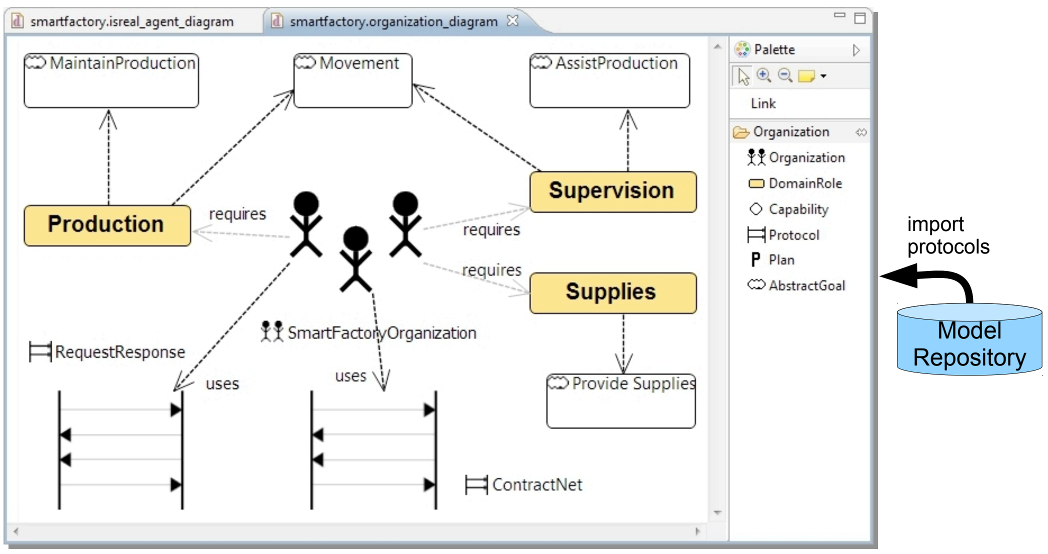


Figure 8.3: This diagram depicts the domain roles of the SmartFactory scenario and their dependencies and responsibilities. The SmartFactoryOrganization uses two interaction protocols to specify the communication between the involved roles.

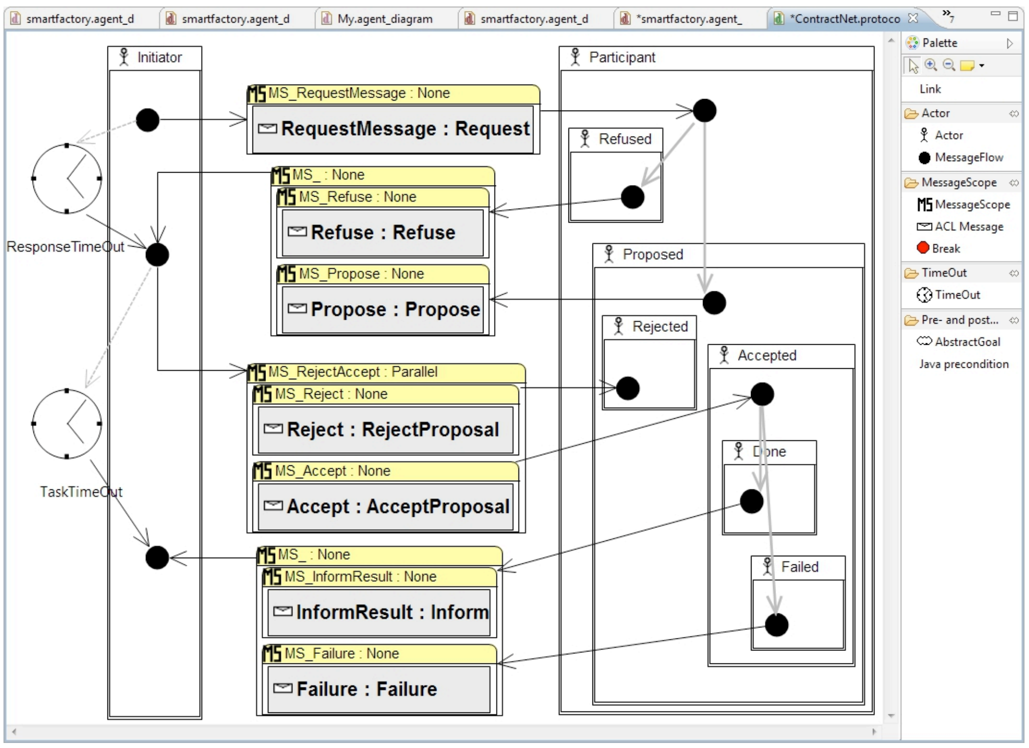


Figure 8.4: This figure depicts the CNP modeled with Bochica.

Collaborations. The concept of `Collaboration` specifies the bindings between domain roles of an organization and actors of an interaction protocol. Figure

8.5 depicts how the RequestHelpCollaboration defines the communication between the Production and Supervision domain roles using the RequestResponse protocol. The RequesterBinding binds the Production role to the Requester actor of the RequestResponse protocol. Likewise, the Supervision role is bound to the Responder actor. The role bindings define the minimum and maximum number of role-fillers of an actor. Interaction protocols define the valid message sequences in terms of abstract messages (concept `ACLMessage`). The concrete messages and content types are specific to a certain collaboration where the protocol is utilized in. For this purpose, the concept `ProtocolConfiguration` defines concrete message types (concept `Message`) for the abstract messages. For example, the abstract RequestMessage is realized by the RequestHelp message, which has the content type RequestHelpDocument (not depicted in the figure). The abstract goals linked to message flows of the RequestResponse protocol represent the responsibilities of an actor in a certain state. The abstract goals are realized by concrete goals of the microscopic layer.

Goals. Abstract goals of the macroscopic layer are realized by concrete goals of the microscopic layer. Concrete goals define the goal type, parameters, and conflicts with other goals. Figure 8.6 depicts how the abstract goals of the RequestResponse protocol from Figure 8.5 are realized by concrete goals. For example, the EvaluateEnquiry goal realizes the abstract ProcessRequest goal. The goal's requestMessage parameter holds the incoming RequestHelp message (direction `IN`). Likewise, the proposeMessage and refuseMessage slots hold the response messages (direction `OUT`). The out-slots have to be set by a behavior of an agent that performs the Supervision role (after it processed the incoming message). The behavior templates, concrete goals, and messages are derived from the interaction protocol. Figure 8.7 depicts an overview of the concrete goals related to the SmartFactoryOrganization. For example, the MoveNearGoal and the WanderAround goal are two concrete goals which realize the abstract Movement goal. The WanderAround goal is a perform goal that makes the agent wander around the 3D scene. The TurnAround goal is a sub-goal for exploring a room (e.g. scanning the environment with the agent's sensor). The MoveNearGoal is an achieve goal that makes the agent walk to some target object (parameter object). The goal's SPARQL target condition checks whether the agent is near the target object or located inside the object (e.g. a room). The MoveNearGoal has the sub-goal MoveThroughDoorGoal for the movement across rooms. The abstract MaintainProduction goal is realized by the KeepMachineRunning maintain goal (maintain condition: $hasComponent(?machine, ?light) \wedge StatusLight(?light) \wedge hasColor(?light, GREEN) \wedge On(?light)$). The variable $?machine$ is bound by the goal's `machine`

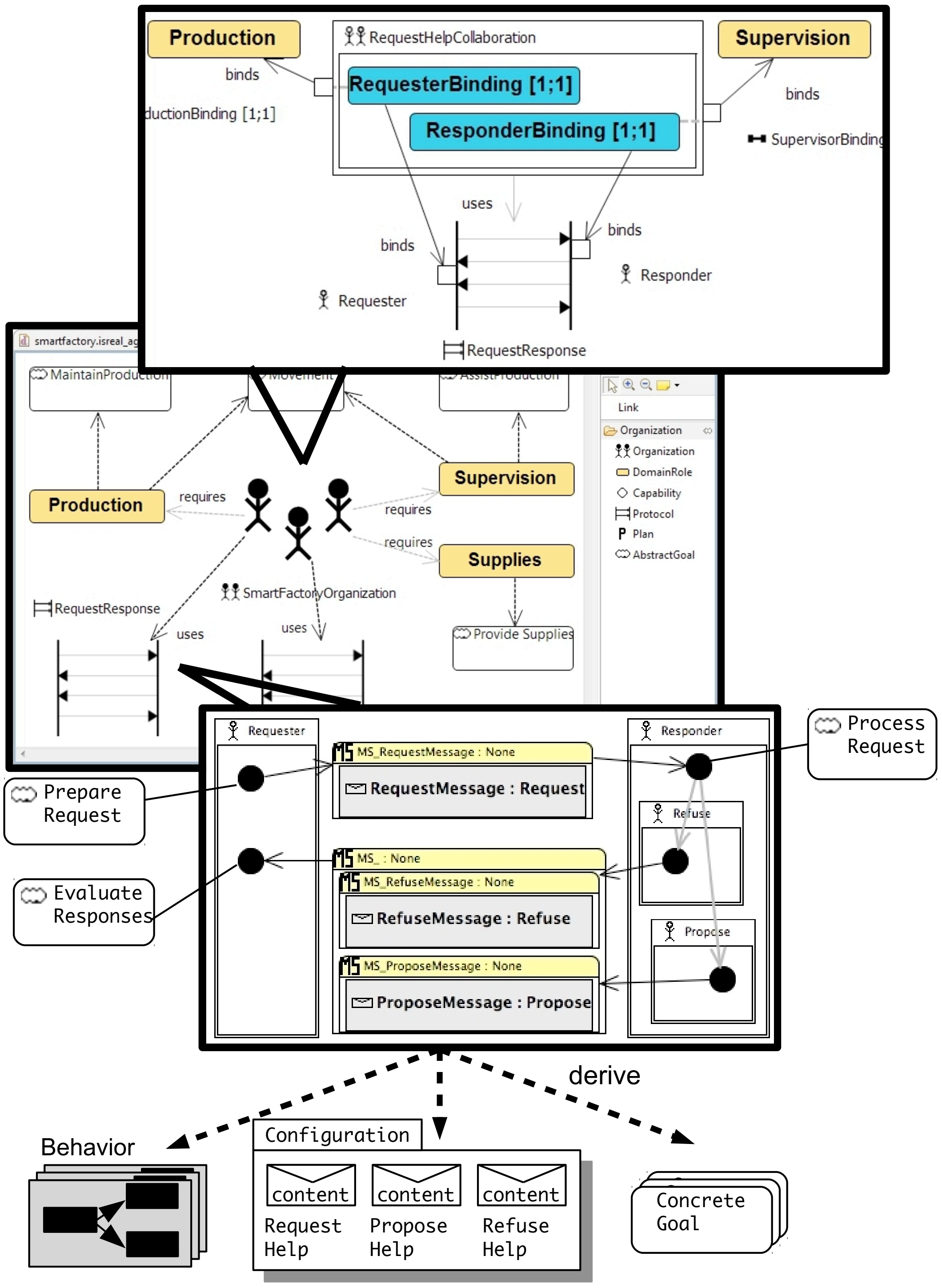


Figure 8.5: This figure depicts the interdependencies between organizations, domain roles, interaction protocols, abstract goals, and collaborations. The abstract goals PrepareRequest, ProcessRequest, and EvaluateResponses represent the responsibilities of the actors in certain states of the RequestResponse protocol.

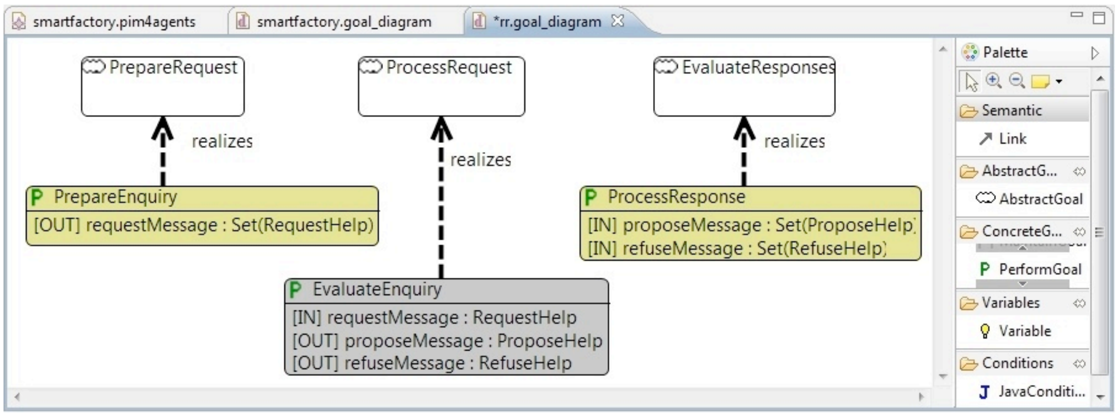


Figure 8.6: This figure depicts the concrete goals for the RequestResponse protocol as utilized by the RequestHelpCollaboration. The PrepareRequest and Evaluate Reponses goals belong to the Requester actor, whereas the EvaluateEnquiry goal is part of the Responder.

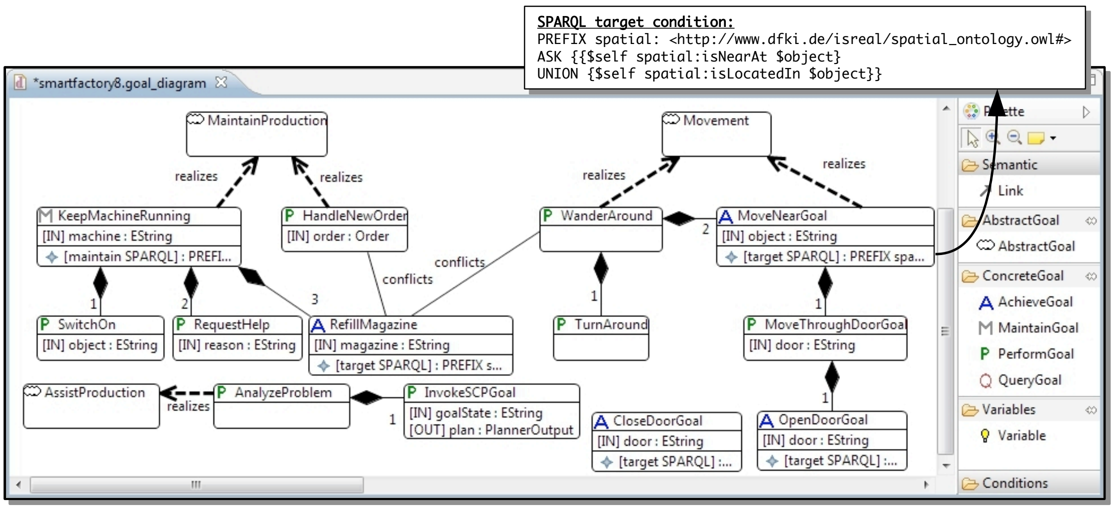

```
SPARQL target condition:
PREFIX spatial: <http://www.dfki.de/isreal/spatial_ontology.owl#>
ASK {{$self spatial:isNearAt $object}
UNION {$self spatial:isLocatedIn $object}}
```

Figure 8.7: This figure depicts an overview of the concrete goals of the SmartFactory case study. The black diamonds represent sub-goal relationships.

parameter (see MR^E-5). The RefillMagazine goal conflicts with the HandleNewOrder and WanderAround goals. The target and maintain conditions of the concrete goals are directly specified in SPARQL.

Agent Types. Figure 8.8 depicts the three agent types that have been designed for performing the domain roles of the SmartFactoryOrganization. Each agent has an own LSE, a geometrical body, and a sensor. The Worker agent performs the Production domain role. Moreover, Figure 8.8 depicts the EvaluateResponsesPlan and PrepareEnquiriesPlan which handle the Production role's goals of the RequestResponse protocol. The two capabilities Movement and SmartFactory equip the agents

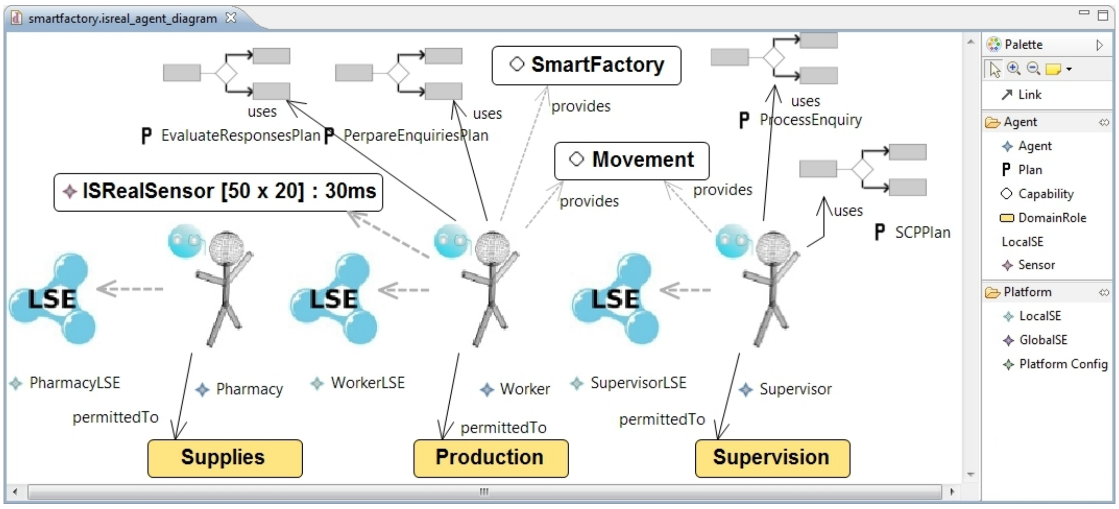


Figure 8.8: This figure depicts an overview of the agent types of the SmartFactory scenario. The ISReal extension model enables the agent engineer to configure an agent's body geometry, knowledge base, and sensor.

with additional behaviors for moving through the scene and operating the Smart-Factory. The Supervisor agent's ProcessEnquiry plan handles the EvaluateEnquiry goal of the RequestResponse protocol. Moreover, the supervisor is equipped with a behavior for invoking the SCP. It enables the agent to react more flexible to problems (e.g. to support the Worker agent). Finally, the Pharmacy agent is responsible for providing the ingredients for manufacturing new pills on the workbench.

Capabilities. Capabilities are reusable and self-contained components that group together a set of behaviors, goals, and knowledges. Two different capabilities have been created for the SmartFactory case study. The SmartFactory capability provides basic functionality for operating the SmartFactory (e.g. switch on/off, use RFID writer). In the following, we focus on the Movement capability (see Figure 8.9). The Movement capability equips an ISReal agent with the behavior related to the abstract Movement goal. This includes (i) movement to objects in the same room, (ii) navigation to known objects in remote rooms, (iii) search for unknown objects and rooms, and (iv) interaction with doors. Some selected behaviors are detailed in the following.

Behaviors. As already discussed in Chapter 7, intelligent ISReal agents interact with their environment by orchestrating object services. Moreover, they use SPARQL expressions for reasoning. Figure 8.10 depicts the MoveNearPlan. It is used for in-room navigation (movement inside a room – no pathfinding across rooms). All high-level movement behaviors (e.g. WanderAround) make use of the

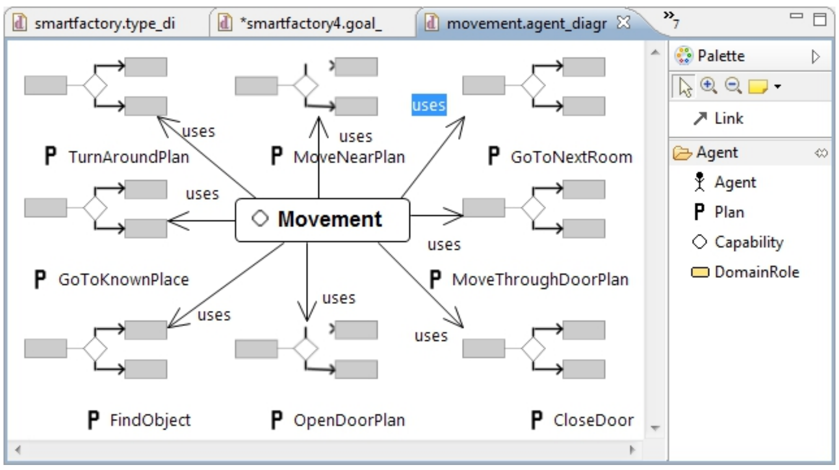
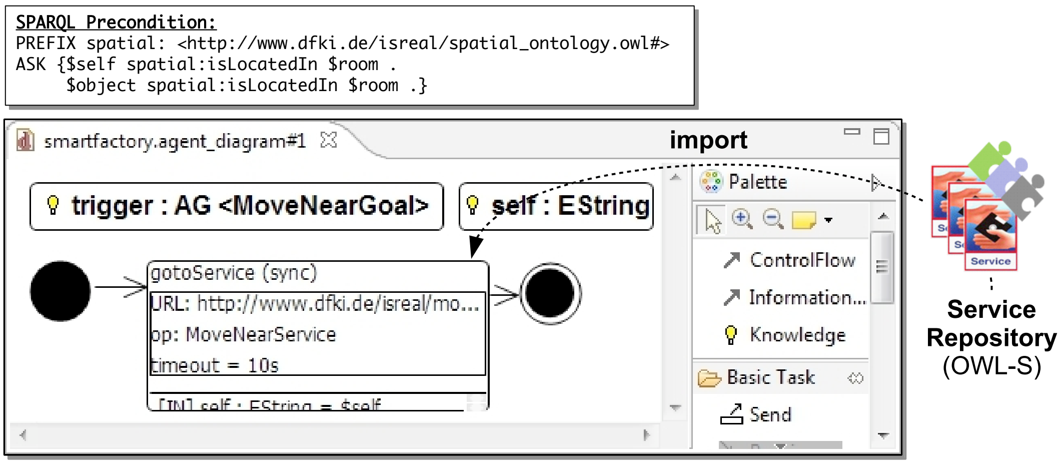


Figure 8.9: This figure depicts the plans of the Movement capability. It covers low-level plans such as the TurnAroundPlan or the OpenDoorPlan. High-level plans, such as the FindObject plan, make use of low-level plans.



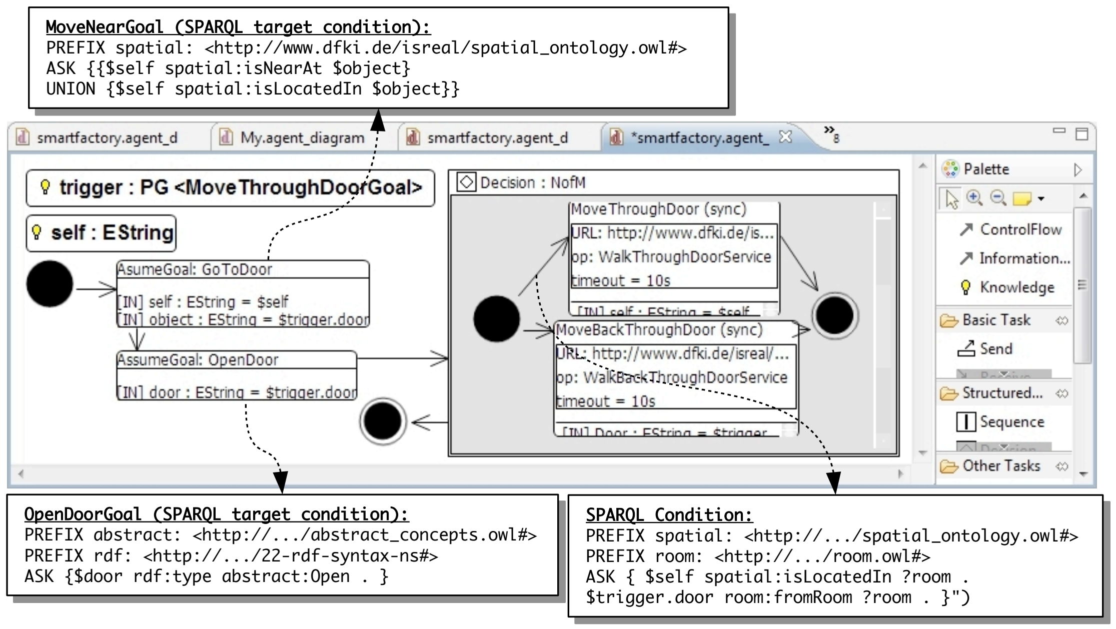
```
SPARQL Precondition:
PREFIX spatial: <http://www.dfki.de/isreal/spatial_ontology.owl#>
ASK {$self spatial:isLocatedIn $room .
     $object spatial:isLocatedIn $room .}
```

smartfactory.agent_diagram#1
import
trigger : AG <MoveNearGoal>
self : EString
gotoService (sync)
URL: http://www.dfki.de/isreal/mo...
op: MoveNearService
timeout = 10s
[IN] self : EString = $self
Palette
ControlFlow
Information...
Knowledge
Basic Task
Send
Service
Service Repository (OWL-S)

Figure 8.10: This figure depicts the MoveNearPlan. It is responsible for in-room navigation of ISReal agents. It invokes the MoveNearService by passing the parameters of the triggering goal.

MoveNearPlan. The plan is triggered by the MoveNearGoal and its SPARQL precondition ensures that it is only activated for target objects located in the same room as the agent. The plan's body invokes a single object service, called MoveNearService. The MoveNearService is imported from the service registry (defined by an OWL-S-based semantic service description).

The MoveThroughDoorPlan enables an agent to walk from the current room to an aligned room (see Figure 8.11). The behavior is triggered by the perform goal MoveThroughDoorGoal. As depicted by the ontology in Figure 7.1, the concept `Door` defines the `fromRoom` and `toRoom` relationships for each side of the door.



```
MoveNearGoal (SPARQL target condition):
PREFIX spatial: <http://www.dfki.de/isreal/spatial_ontology.owl#>
ASK {{$self spatial:isNearAt $object}
UNION {$self spatial:isLocatedIn $object}}
```

```
trigger : PG <MoveThroughDoorGoal>
self : EString
AsumeGoal: GoToDoor
[IN] self : EString = $self
[IN] object : EString = $trigger.door
AssumeGoal: OpenDoor
[IN] door : EString = $trigger.door
Decision : NofM
MoveThroughDoor (sync)
URL: http://www.dfki.de/is...
op: WalkThroughDoorService
timeout = 10s
[IN] self : EString = $self
MoveBackThroughDoor (sync)
URL: http://www.dfki.de/isreal/...
op: WalkBackThroughDoorService
timeout = 10s
[IN] Door : EString = $trigger
```

```
OpenDoorGoal (SPARQL target condition):
PREFIX abstract: <http://.../abstract_concepts.owl#>
PREFIX rdf: <http://.../22-rdf-syntax-ns#>
ASK {$door rdf:type abstract:Open . }
```

```
SPARQL Condition:
PREFIX spatial: <http://.../spatial_ontology.owl#>
PREFIX room: <http://.../room.owl#>
ASK { $self spatial:isLocatedIn ?room .
$trigger.door room:fromRoom ?room . }")
```

Figure 8.11: Moving through a door involves (i) walking to the door, (ii) opening the door, and (iii) invoking the WalkThroughDoorService or WalkBackThroughDoorService.

Likewise, there exist two different animation services for moving an agent from one side of the door to the opposite one. Depending on the side of the door the agent is located at, the according animation service has to be invoked. This decision is made by the MoveThroughDoorPlan (see Figure 8.11). The body of the MoveThroughDoorPlan consists of (i) walking to the door by posting a MoveNearGoal, (ii) opening the door by posting the OpenDoorGoal, and (iii) either invoking the WalkThroughDoorService or WalkBackThroughDoorService. Figure 8.11 also depicts the target conditions of the sub-goals. If a target condition is already fulfilled at the time the goal is posted (e.g. the door is already open), the goal is immediately achieved. The control flow forking from the decision's begin node uses a SPARQL-Ask query to decide on which side of the door the agent is located at and invokes the according object service.

Data Model. Figure 8.12 depicts a representative part of the Ecore-based SmartFacotory data model. The Order, Client, and Item classes define simple data types for assigning new orders to the Worker agent (see HandleNewOrder goal in Figure 8.7). Sometimes, it is necessary to access data types of the target platform. For example, the Topology and PlannerOutput classes are placeholders at the modeling level for already existing artifacts of the ISReal platform. For this purpose, the Ecore metamodels provides the `instanceClassName` attribute. It defines the fully

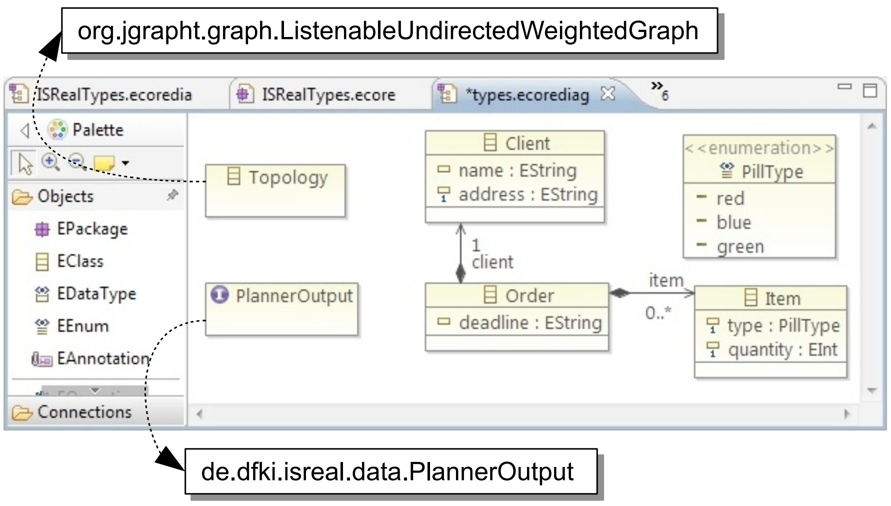


Figure 8.12: A part of the Ecore-based SmartFactory data model. The classes Topology and PlannerOutput are placeholders for data types of the target platform.

qualified class name. During code generation, the `instanceClassName` is resolved to the according platform artifact. For example, the Topology class is part of the *jgrapht*[70] library which is used to compute the shortest path (rooms are represented as nodes and doors as vertices). Moreover, the PlannerOutput class defines the result type of an SCP invocation. Thus, data types of the target platform can be made available at the modeling level. One example of how those types are used is the InvokeSCPGoal depicted in Figure 8.7. It uses the PlannerOutput data type as output parameter.

Deployment

Figure 8.13 depicts the deployment configuration of the SmartFactory scenario. The organization instance org1 is an instance of the SmartFactoryOrganization and has five members. Bob is an instance of the Supervisor agent and performs the Supervision role, nancy is a Worker agent that performs the Production role, and the three Pharmacy agents perform the Supplies role. The agent instances use knowledge and goal initializers to configure the initial state. For example, nancy has an initial goal of type KeepMachineRunning with the SmartFactory fair module as parameter.

Implementation

The Bochica model artifacts presented in the previous section are mapped by the model transformations introduced by Chapters 5 and 7 to the ISReal-enabled

[70]`http://jgrapht.org`

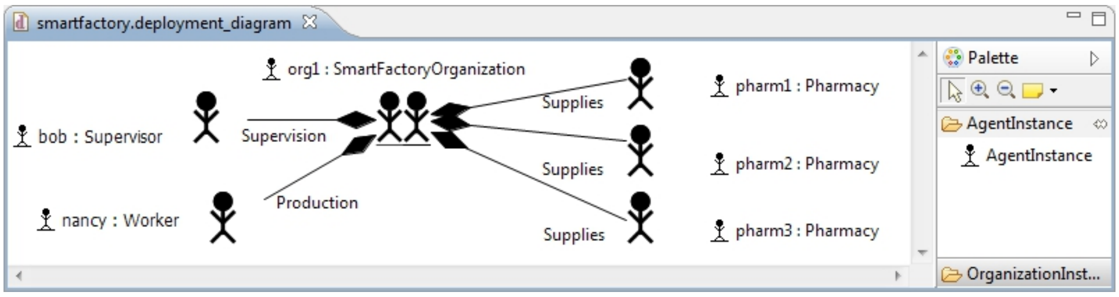


Figure 8.13: This figure depicts an example deployment configuration for the SmartFactory case study.

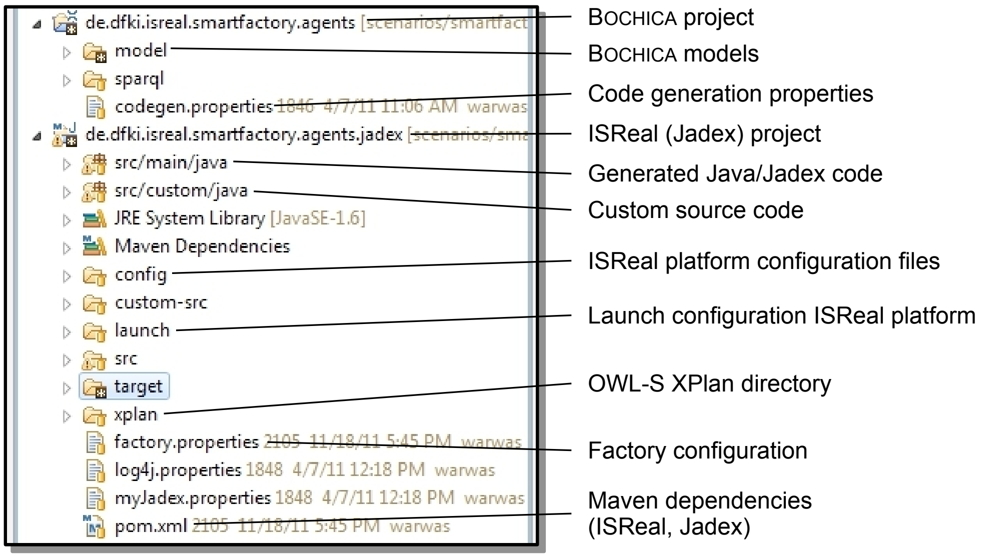
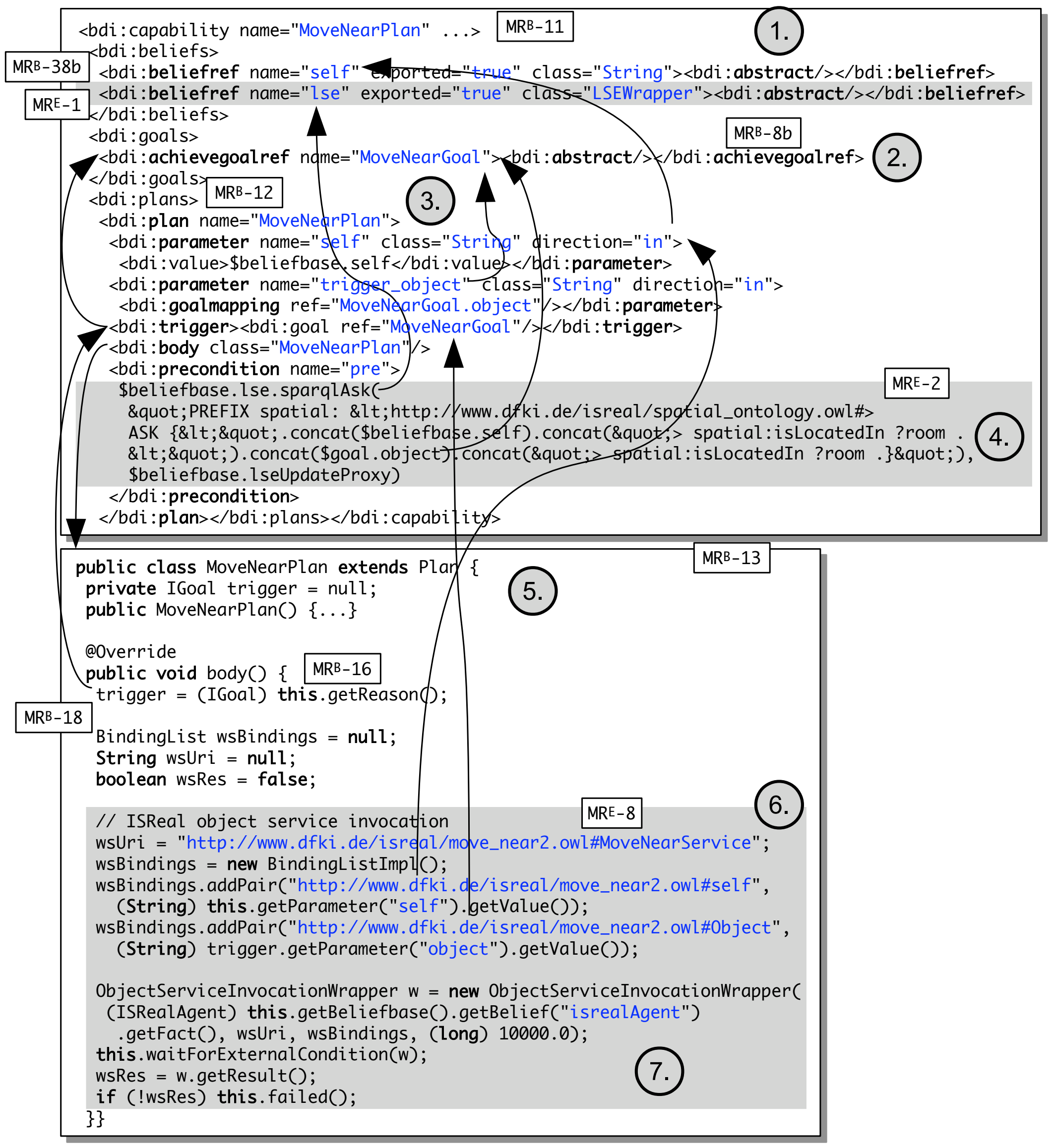

de.dfki.isreal.smartfactory.agents [scenarios/smartfact
model
sparql
codegen.properties 1846 4/7/11 11:06 AM warwas
de.dfki.isreal.smartfactory.agents.jadex [scenarios/sma
src/main/java
src/custom/java
JRE System Library [JavaSE-1.6]
Maven Dependencies
config
custom-src
launch
src
target
xplan
factory.properties 2105 11/18/11 5:45 PM warwas
log4j.properties 1848 4/7/11 12:18 PM warwas
myJadex.properties 1848 4/7/11 12:18 PM warwas
pom.xml 2105 11/18/11 5:45 PM warwas
BOCHICA project
BOCHICA models
Code generation properties
ISReal (Jadex) project
Generated Java/Jadex code
Custom source code
ISReal platform configuration files
Launch configuration ISReal platform
OWL-S XPlan directory
Factory configuration
Maven dependencies
(ISReal, Jadex)

Figure 8.14: Overview of the Eclipse project layout.

Jadex platform. Large parts of the source code can be generated and only punctual manual extensions are required (mainly for adding business logic). Figure 8.14 depicts an overview of the overall project structure. The top-most project is the BOCHICA modeling project. The generated Jadex project uses Maven[71] to configure the required Java libraries for ISReal and Jadex. The generated code (src/main/java folder) is separated from the manually written code (src/custom/java folder). Moreover, the config folder holds the configuration files for the ISReal platform (e.g. LSE and GSE configurations). The factory.properties file configures the location of the custom code.

Figure 8.15 depicts the fully automatic generated code of the MoveNearPlan shown in Figure 8.10. The code consists of (i) the plan capability and (ii) the actual plan body (see MR^{B}-11, MR^{B}-12, MR^{B}-13). The plan capability configures

[71]http://maven.apache.org



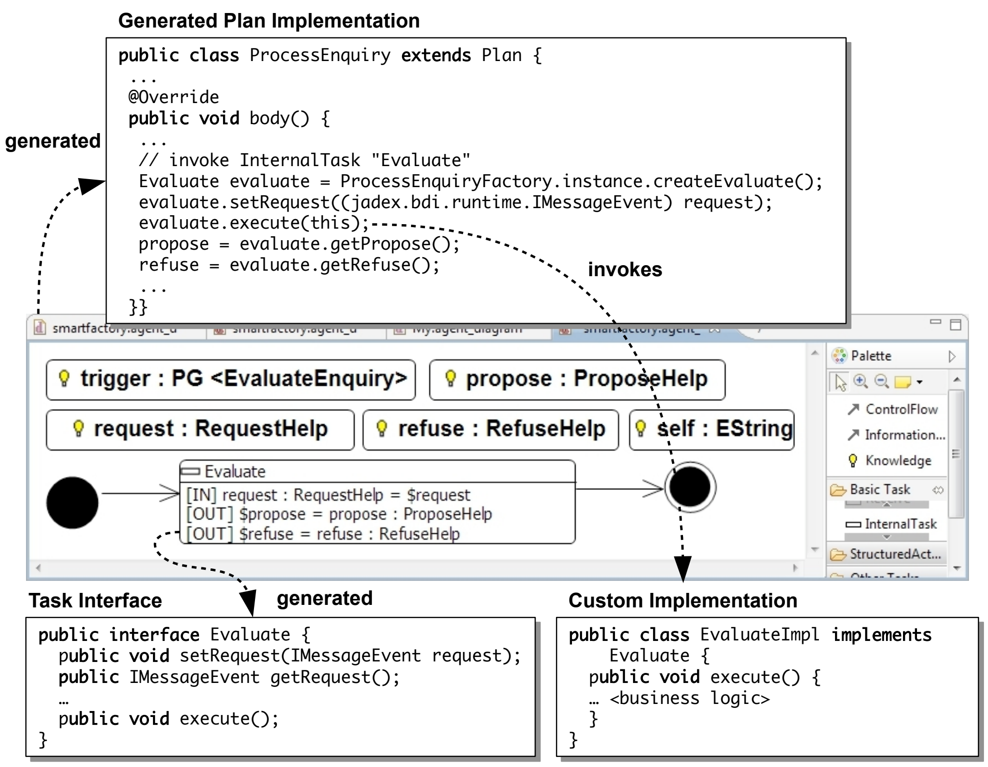
```
<bdi:capability name="MoveNearPlan" ...>
 <bdi:beliefs>
  <bdi:beliefref name="self" exported="true" class="String"><bdi:abstract/></bdi:beliefref>
  <bdi:beliefref name="lse" exported="true" class="LSEWrapper"><bdi:abstract/></bdi:beliefref>
 </bdi:beliefs>
 <bdi:goals>
  <bdi:achievegoalref name="MoveNearGoal"><bdi:abstract/></bdi:achievegoalref>
 </bdi:goals>
 <bdi:plans>
  <bdi:plan name="MoveNearPlan">
   <bdi:parameter name="self" class="String" direction="in">
    <bdi:value>$beliefbase.self</bdi:value></bdi:parameter>
   <bdi:parameter name="trigger_object" class="String" direction="in">
    <bdi:goalmapping ref="MoveNearGoal.object"/></bdi:parameter>
   <bdi:trigger><bdi:goal ref="MoveNearGoal"/></bdi:trigger>
   <bdi:body class="MoveNearPlan"/>
   <bdi:precondition name="pre">
    $beliefbase.lse.sparqlAsk(
     &quot;PREFIX spatial: &lt;http://www.dfki.de/isreal/spatial_ontology.owl#>
     ASK {&lt;&quot;.concat($beliefbase.self).concat(&quot;> spatial:isLocatedIn ?room .
     &lt;&quot;).concat($goal.object).concat(&quot;> spatial:isLocatedIn ?room .}&quot;),
     $beliefbase.lseUpdateProxy)
   </bdi:precondition>
  </bdi:plan></bdi:plans></bdi:capability>
```

```
public class MoveNearPlan extends Plan {
 private IGoal trigger = null;
 public MoveNearPlan() {...}

 @Override
 public void body() {
  trigger = (IGoal) this.getReason();

  BindingList wsBindings = null;
  String wsUri = null;
  boolean wsRes = false;

  // ISReal object service invocation
  wsUri = "http://www.dfki.de/isreal/move_near2.owl#MoveNearService";
  wsBindings = new BindingListImpl();
  wsBindings.addPair("http://www.dfki.de/isreal/move_near2.owl#self",
    (String) this.getParameter("self").getValue());
  wsBindings.addPair("http://www.dfki.de/isreal/move_near2.owl#Object",
    (String) trigger.getParameter("object").getValue());

  ObjectServiceInvocationWrapper w = new ObjectServiceInvocationWrapper(
   (ISRealAgent) this.getBeliefbase().getBelief("isrealAgent")
    .getFact(), wsUri, wsBindings, (long) 10000.0);
  this.waitForExternalCondition(w);
  wsRes = w.getResult();
  if (!wsRes) this.failed();
 }}
```

Figure 8.15: The fully automatic generated Jadex code for the MoveNearPlan. This figure depicts (1) the placeholders for the required beliefs, (2) the placeholder for the MoveNearGoal, (3) the meta information of the plan, (4) the plan's precondition in SPARQL-Ask, (5) the actual plan implementation, (6) the parameter bindings of the object service invocation, and (7) the actual object service invocation. The arrows visualize important references. The gray parts highlight code generated by the ISReal extension transformation.

the beliefs, goals, etc. that are accessed by the plan body. The ISReal extension transformation added the lse belief to the bliefbase. It enables the plan to access the agent's LSE (see MR^E-1). The LSE is accessed to evaluate the SPARQL-Ask precondition. The extension transformation resolved the variable symbols of the SPARQL-Ask expression in the surrounding scope and assembled the according

SPARQL-Ask expression as required by the generated Jadex code (see MR^{E}-2). For example, the $object reference was resolved to the triggering goal's *object* parameter ($goal.object). Moreover, the plan header references the plan implementation (attribute body). The plan implementation makes the parameters accessible within the code, prepares the parameter bindings of the service invocation, and invokes the MoveNearService which is executed by the GSE as discussed in Section 7.2. At runtime, the GSE checks the service's precondition globally, invokes the service grounding, and updates the global world state as the agent arrives at the target object. The effects are perceived by the agent through its sensor. As the agent updates its LSE, it achieves the target condition of the triggering MoveNearGoal. What should be noted is a significant difference in the level of abstraction compared to the BOCHICA model. Implementing the generated Jadex code manually is very error prone as (i) many references are defined as string values (possible typing errors), (ii) the passing of the parameters is verbose and one can easily forget elements, (iii) the SPARQL expression has to be assembled manually as depicted in Figure 8.15, and (iv) the engineer has to know how to implement the object service invocation at code level. The code that has been generated by the ISReal extension transformation is highlighted in gray. One can clearly see how the extension transformation applies punctual extensions which integrate into the overall structure generated by the base transformation. Figure 8.16 depicts the ProcessEnquiry plan. It implements the Supervisor agent's behavior for evaluating a RequestHelp message. The RequestHelp message is part of the RequestResponse protocol (see Figure 8.5). The business logic for evaluating the message is encapsulated by the internal task Evaluate. Figure 8.16 depicts the generated Java code for the plan and the internal task as discussed in Section 5.3 (MR^{B}-21a/b). The Evaluate Java interface defines the getter and setter methods for the parameters. The EvaluateImpl class is a custom implementation of the internal task. The agent engineer's task is to (i) process the content of the incoming message (parameters with direction IN) and (ii) set the out-parameters of the internal task (parameters with direction OUT). In order to execute the encapsulated business logic, the generated code (i) uses the plan's generated internal task factory to instantiate the business logic, (ii) uses the setter methods for passing the input parameters, (iii) executes the business logic by invoking the execute() method, and (iv) reads the output parameters using the getter methods. Afterwards, the generated code automatically processes the messages according to the interaction protocol specification (see MR^{B}-29, MR^{B}-30, MR^{B}-31). Thus, the implementation of the protocol is transparent to the agent developer. He can focus on the implementation of the business logic. Figure 8.16 also shows how the Message data type of BOCHICA has been mapped according to Table 5.5 to a Jadex IMessageEvent.



```
public class ProcessEnquiry extends Plan {
 ...
 @Override
 public void body() {
  ...
  // invoke InternalTask "Evaluate"
  Evaluate evaluate = ProcessEnquiryFactory.instance.createEvaluate();
  evaluate.setRequest((jadex.bdi.runtime.IMessageEvent) request);
  evaluate.execute(this);
  propose = evaluate.getPropose();
  refuse = evaluate.getRefuse();
  ...
 }}
```

```
public interface Evaluate {
  public void setRequest(IMessageEvent request);
  public IMessageEvent getRequest();
  …
  public void execute();
}
```

```
public class EvaluateImpl implements
    Evaluate {
  public void execute() {
  … <business logic>
  }
}
```

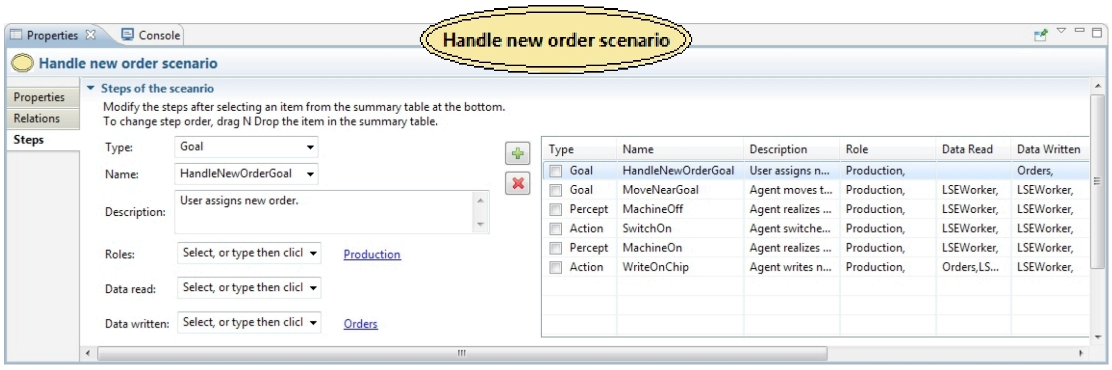
Figure 8.16: This figure depicts the ProcessEnquiry plan and the generated Java files for implementing the business logic of the internal task (see MR^B-21a/b).

8.1.3 Prometheus

The Prometheus methodology for AOSE guides agent engineers through several development phases from system specification to implementation (see Section 2.2.2). Prometheus is supported by the PDT modeling tool which provides graphical diagrams for capturing the design decisions. In contrast to Bochica, Prometheus provides support for gradually deriving the relevant model artifacts of a SUC (e.g. specification of goals, interactions, actions, perceptions). Later phases use those artifacts as basis for the system design (e.g. agent types, capabilities, and behaviors) similar to Bochica. In the following, we depict representative parts of the different development phases of the Prometheus methodology for the SmartFactory case study.

System Specification

Prometheus uses so called *scenarios* for identifying typical workflows of a SUC. Figure 8.17 depicts the "Handle new order scenario" of the SmartFactory case study. It describes the process for handling a new order by the Production role. First, the incoming order causes a HandleNewOrderGoal for the Production role. The order



Handle new order scenario

Steps of the sceanrio

Modify the steps after selecting an item from the summary table at the bottom.
To change step order, drag N Drop the item in the summary table.

Type: Goal
Name: HandleNewOrderGoal
Description: User assigns new order.
Roles: Select, or type then clicl Production
Data read: Select, or type then clicl
Data written: Select, or type then clicl Orders

Type	Name	Description	Role	Data Read	Data Written
Goal	HandleNewOrderGoal	User assigns n...	Production,		Orders,
Goal	MoveNearGoal	Agent moves t...	Production,	LSEWorker,	LSEWorker,
Percept	MachineOff	Agent realizes ...	Production,	LSEWorker,	LSEWorker,
Action	SwitchOn	Agent switche...	Production,	LSEWorker,	LSEWorker,
Percept	MachineOn	Agent realizes ...	Production,	LSEWorker,	LSEWorker,
Action	WriteOnChip	Agent writes n...	Production,	Orders,LS...	LSEWorker,

Figure 8.17: This figure depicts the "Handle new order scenario" in Prometheus.

is written to the Orders data set. Moreover, the order causes a MoveNearGoal for walking to the SmartFactory. In this scenario, the role filler performing the Production role realizes that the fair module is switched off. This causes the role filler to switch the machine on (represented by the SwitchOn action). After the agent perceived that the machine is running, it uses the WriteOnChip action to write the order onto the RFID tag of a cup. Prometheus scenarios are well suited for identifying model artifacts and responsibilities. The "Handle new order scenario" involves only one role. In the architecture design phase, the model artifacts of the different scenarios are grouped to functionalities[72]. Textual remarks are used to document the rational behind the model artifacts (e.g. descriptions of actions and perceptions). The system specification phase also identifies the system's goals and their dependencies. Figure 8.18 depicts an overview of the identified goals. At the first sight, the diagram looks very similar to the BOCHICA goal diagram depicted in Figure 8.7. However, goals in Prometheus are only design artifacts that are replaced by events in later development phases. Thus, Prometheus goals are similar to `AbstractGoals` in BOCHICA and are not directly used for code generation. Thus, Prometheus goals do not specify goal types, parameters, or formal target conditions.

Architectural Design

In Prometheus, interaction protocols specify the interactions between agents and actors (external entities of the system). Besides messages, the interactions also cover perceptions and actions. Interactions in Prometheus are derived from the modeled scenarios of the system specification phase (see Figure 8.19 a). After the identified functionalities and roles of the system are grouped to agents, the

[72]Since the publication in [Padgham and Winikoff, 2004], Prometheus has been further developed and some details changed. For example, functionalities are now replaced by roles and the concept of actor has been introduced to represent external entities of the system.

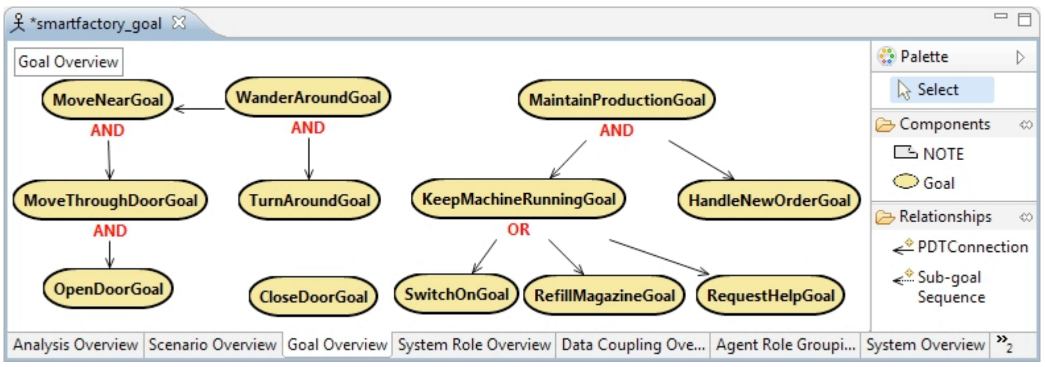
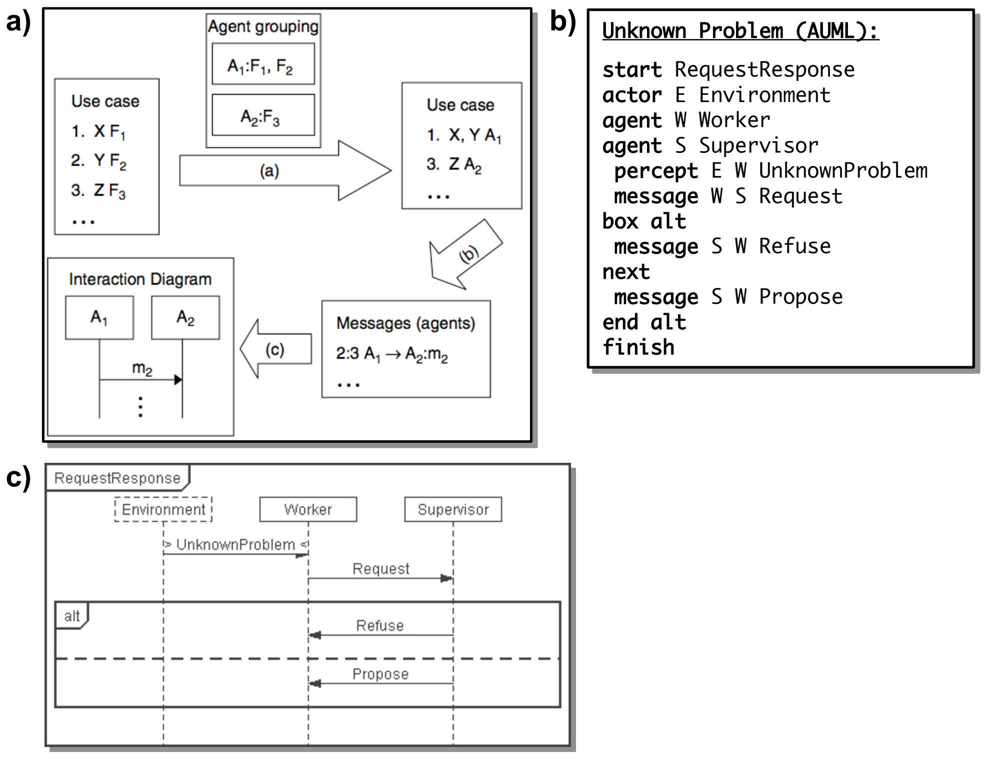


Figure 8.18: Prometheus goals of the SmartFactory scenario.



```
Unknown Problem (AUML):

start RequestResponse
actor E Environment
agent W Worker
agent S Supervisor
 percept E W UnknownProblem
 message W S Request
box alt
 message S W Refuse
next
 message S W Propose
end alt
finish
```

Figure 8.19: (a) depicts the role of interactions in Prometheus (taken from [Padgham and Winikoff, 2004, p.68]), (b) depicts the textual representation of an AUML interaction, and (c) depicts the graphical AUML representation of the interaction.

messages required for the communication between the agents are derived. Finally, those messages, perceptions, and actions are grouped to interaction protocols. Figure 8.19 b) depicts a textual AUML representation of the interaction which causes the worker agent to request help from the supervisor agent. The Environment is modeled as external actor. After the worker agent perceived a problem that cannot be solved by itself, it requests help from the supervisor using a request

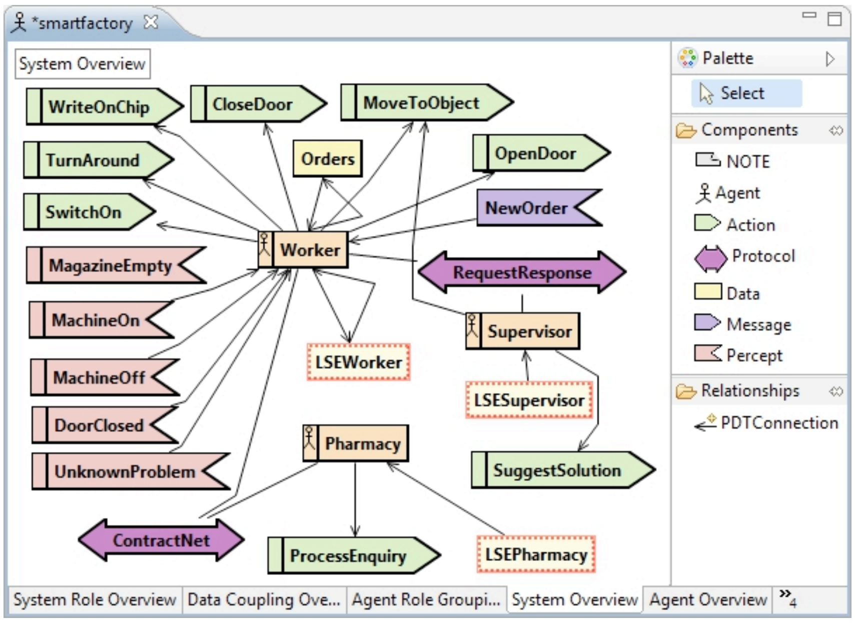


Figure 8.20: The Prometheus system overview diagram of the SmartFactory case study.

response protocol. Figure 8.19 c) depicts the graphical AUML representation. It is important to note that the interaction protocols model the message exchange between agent types, whereas interaction protocols in Bochica define the message exchange between named sets of role-fillers. Moreover, interaction protocols in Bochica are abstract artifacts that are independent of a concrete use case. In Prometheus, interaction protocols are used to capture the interactions of the system for supporting the manual implementation. Bochica does not guide the user in identifying the required message sequences and only covers messages (no actions and perceptions). Thus, protocols in Bochica and Prometheus have a slightly different use.

Figure 8.20 depicts the Prometheus system overview diagram for the SmartFactory case study. The three agent types Worker, Supervisor, and Pharmacy have been identified. Each of them has an own Data object that represents the ISReal LSE. Moreover, the used interaction protocols, messages, perceptions, and actions are visualized. For example, the OpenDoor action represents the ISReal object service for opening a door. However, the model artifacts are very abstract and require textual explanations to specify their purpose.

Detailed Design

Figure 8.21 depicts the Movement capability of the Worker agent. Six different plans have been modeled. All of them have access to the agent's LSE. Moreover,

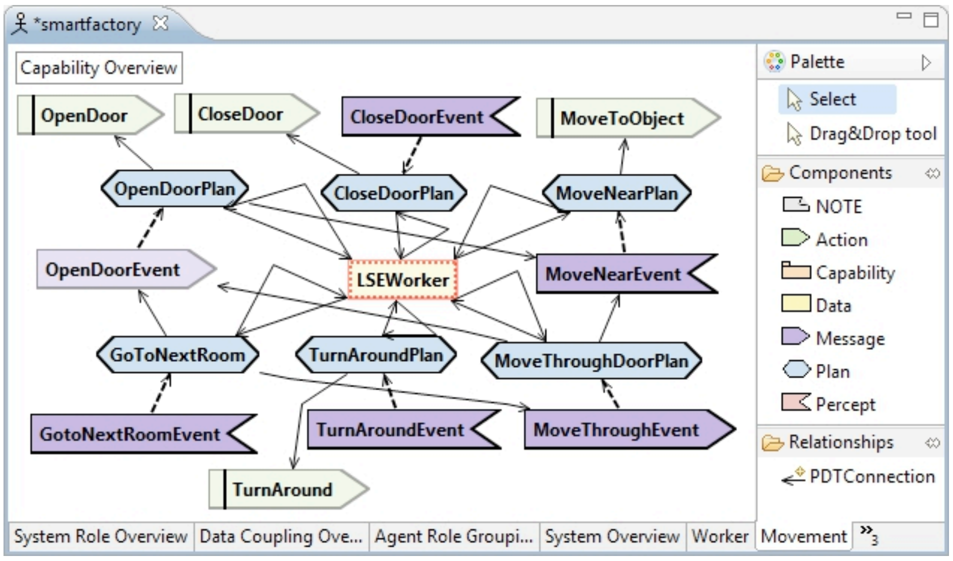


Figure 8.21: The movement capability modeled with Prometheus.

the involved messages and actions are visualized. For example, the OpenDoorPlan is triggered by the OpenDoorEvent and makes use of the OpenDoor action. Plan bodies are specified at code level. It is also important to note that PDT does not provide the possibility for modeling events with formal target conditions. Likewise, the LSEWorker is only a placeholder for something to be manually implemented at code level. Deployment configurations are not part of PDT.

Implementation

After the system has been specified using Prometheus, PDT offers the possibility to generate code for the Jack platform. Figure 8.22 depicts the generated code template for the OpenDoorPlan. The template consists of (i) the declaration of the handled and posted events, (ii) the used data, (iii) the declaration of the context condition, and (iv) the actual plan body. The context condition as well as the plan body are described using plain text. It is important to note that the integration of the LSE and SPARQL-based context conditions into Jack requires extensive customizations. The created Prometheus models and thus, the generated code do not cover those aspects. It is easy to see that the generated code only provides simple code templates with textual annotations. Moreover, interactions have to be implemented manually using the AUML diagrams. The succeeding section realizes the ISReal case study using a Jack-only approach. The results of the Jack-only approach are used to discuss the effort for customizing the generated Prometheus code templates presented in this section.

```
public plan OpenDoorPlan extends Plan {
/******** Start PDT Design Block *** DO NOT EDIT IT *********/

/* Plan Name: OpenDoorPlan
   Description: Makes an agent open a door in the same room. */

//Events handled by the plan are declared here.
#posts event MoveNearEvent movenearevent_p;
#handles event OpenDoorEvent opendoorevent_h;

// Declarations of any beliefset/data that the plan accesses.
#uses data LSEWorker LSEWorker_dat;

/******** End PDT Design Block *** DO NOT EDIT IT *********/
/* Action Reminder: Name: OpenDoor */

context() {
/*Context: Agent is located in same room as door.*/
// Trigger: Event "OpenDoorEvent"
true;}

body() {
// The plan body. This describes the actual steps
// an agent performs when it executes this plan.
// Procedure:
//  (1) move to door, (2) open door
}}
```

Figure 8.22: The generated Prometheus OpenDoorPlan template.

8.1.4 Jack

In the third part of the SmartFactory case study, we use a platform-specific approach to implement the SUC. As part of the ISReal project, the BDI agent platform Jack has been enabled for ISReal by implementing the architectural pattern depicted in Figure 7.3. The integration of the LSE and SPARQL-based reasoning into Jack required several extensive extensions. Since there exists no prescribed development process in Jack, the design of agents and the structure of the Java code depends on an agent engineer's experience. Figure 8.23 depicts an overview of the plans and events related to an ISReal agent's movement in Jack. The task of diagrams in Jack is to visualize the interdependencies between the different Java artifacts. Interaction protocols have to be implemented manually.

Figure 8.24 depicts the OpenDoor plan implemented in Jack. The plan is triggered by the OpenDoorGoal. The plan's context condition ensures that the plan only gets activated when the agent and the door are located in the same room. In order to evaluate the SPARQL-Select statement, the adapter class `JackISRealAgentAdapter` provides access to an agent's LSE using the `getSPARQL()` method. The invocation of the `select()` method (i) passes the SPARQL query to the LSE and (ii) converts the query result to a so called Jack *belief cursor*. Belief cursors are used by Jack to access the variable bindings of the context

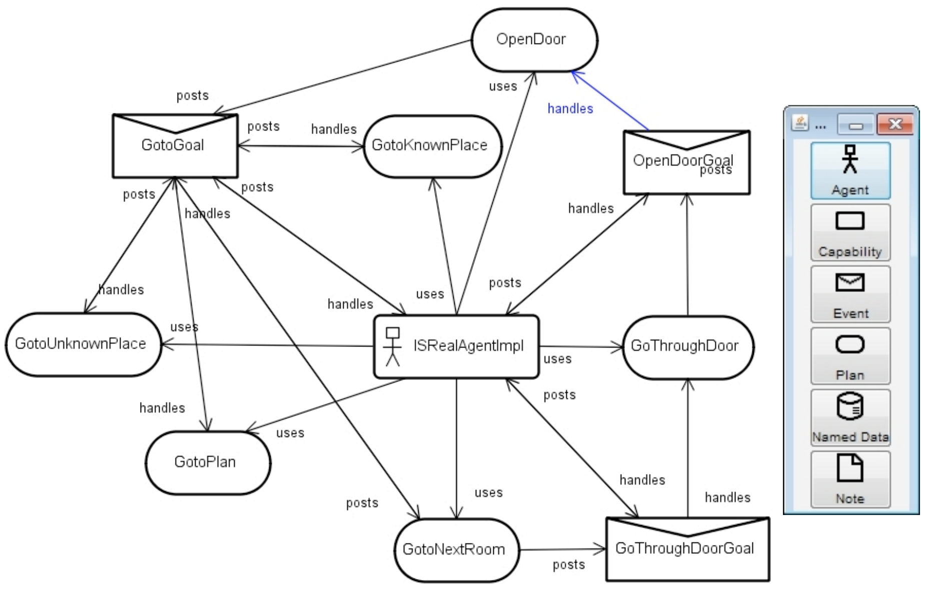


Figure 8.23: Jack goto goal and plan diagram.

condition. So called *logicals* make the variable bindings of the context condition available within the plan body (e.g. `v1` in Figure 8.24). The plan's body depicts the invocation of a Jack `@subtask` for executing the OpenDoorService object service. The sub-task requires the variable bindings for the object service invocation and posts a sub-goal to perform the invocation. The actual service invocation is implemented equivalently to the object service invocation in Jadex (see Section 8.1.2).

By comparing the generated Jadex code and the manually written Jack code, one can see that the effort for implementing ISReal behaviors is similar in Jack and Jadex. In both cases, the integration of the LSE and SPARQL required extensive custom extensions. The service invocation is done very similar in Jack and Jadex. Moreover, one can see that the Jack code templates generated by Prometheus only provide a rough structure of the code. One reason is that the Prometheus models leave many details open and use informal textual descriptions. The gap between the models and the implementation can cause design and code to diverge over time. The manual implementation of ISReal scenarios in Jack and Jadex requires the agent engineer to over and over again write down the same ISReal-specific extensions (e.g. for SPARQL preconditions and context conditions). A model-driven approach requires the customization of the model transformation only once. All succeeding code generations solve the issue automatically. Writing a custom transformation would also work for Prometheus but the modeling language leaves many details open. What is not specified in a formal way at the model level cannot be processed by model transformations.

```
public plan OpenDoor extends Plan {
  logical String v1;
  #handles event OpenDoorGoal ev;
   ...
   context() {
     ((JackISRealAgentAdapter)getAgent()).getSPARQL().select(this, 1,
        "SELECT ?v1 " + "WHERE  { " +
          "<" + ((ISRealAgent)getAgent()).getSelfURI() +
          "> <http://www.dfki.de/isreal/spatial_ontology.owl#isLocatedIn> ?v1 ." +
           "?v1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
           <http://www.dfki.de/isreal/room.owl#Room> . " +
          "<" + ev.door + ">
           <http://www.dfki.de/isreal/spatial_ontology.owl#isLocatedIn> ?v1 . " +
          "<" + ev.door + "> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
           <http://www.dfki.de/isreal/room.owl#Door> . " + "}");}

  #reasoning method
  body() {
    ISRealAgent a = (ISRealAgent)getAgent();
    SPARQLHelper helper = new SPARQLHelper(a.getLocalSE());

    // check whether the agent is already at the door
    if(!helper.isNearAt(a.getSelfURI(), ev.door)) {
      ... } ...

  // invoke open door service
  String baseURI = "http://www.dfki.de/isreal/open_door.owl";
  String openDoorService = baseURI + "#OpenDoorService";
  BindingList inp = new BindingListImpl();
  inp.addPair((String) baseURI + "#self", ev.door);
  @subtask(ev2.init(openDoorService, inp));
}}
```

Figure 8.24: This figure depicts the OpenDoor plan in Jack. The integration of SPARQL can be seen at the context condition. The body invokes the OpenDoorService using a Jack sub-task.

8.1.5 Summary and Discussion

There exist many different ways of how to implement the behavior of intelligent agents in the SmartFactory case study. One could hard code the behavior of the agents in more or less flexible scripting languages (e.g. Java Script) or use classical object-oriented languages (e.g. Java). An agent-oriented approach has the advantage that agent platforms already provide the infrastructure for goal-driven behavior, messaging, etc. MDSD was developed by industry to approach the increasing complexity of IT systems. The underlying idea of MDSD is to separate the design of a SUC from its implementation. The ideal modeling language for specifying intelligent agents in semantically-enriched virtual worlds would cover typical agent concepts, concepts related to Semantic Web technology and computer graphics, as well as concepts of the ISReal platform. The benefit of such a language is that the SUC can be efficiently modeled on an abstract level. Moreover, the better a modeling language covers the concepts of the target domain, the less manual modifications are required at code level. The creation of such a software language from scratch is a complex endeavour. Furthermore, many concepts would be similar to other agent-oriented languages of other application domains. Thus, it would mean to reinvent many things.

The SmartFactory case study presented in this section provides a challenging setting for the evaluation of agent-oriented modeling languages. The aim of the case study was to design agents that simulate human behavior in a virtual production line scenario based on the DFKI SmartFactory living lab and the ISReal platform. In order to specify a running system, a modeling language has to exploit the intelligent agent paradigm, address application domain-specific concepts and software languages, and consider features of a proprietary target platform. Especially the integration of knowledge bases and reasoning languages is an often neglected aspect in agent-oriented modeling – but is inevitable for intelligent ISReal agents. Finally, the agent platform is not considered as a self-contained system. Instead, the platform is embedded into a larger execution environment.

In Section 2.2.3, we have classified existing agent-oriented modeling approaches into platform-specific and platform-independent ones. This section compared the Bochica framework to one representative of each category. In the following, we further discuss and analyse the created system and made experiences. One important question is the customization effort required to tailor Bochica to a certain target environment. Table 8.1 compares the infrastructure provided by the Bochica framework and the ISReal-specific extensions. The Bochica core DSL encompasses around 124 concepts. The Jadex base transformation (introduced in Chapter 5) was implemented in 137 QVT-based mapping rules with 2795 *Lines of Code* (LOC) and 70 XPand-based rules with 1350 LOC. The ISReal extension model for Bochica consists of eight new concepts and one software language (see Chapter 7). The ISReal extension transformation was implemented in 16 QVT and six XPand-based mapping rules. One can see that around 10% of custom concepts and mapping rules were introduced. The relative increase between the number of mapping rules (10.6%) and LOC (17.5%) of the extension transformation is caused by helper functions for processing SPARQL expression models in the extension transformation. For example, the helper functions resolve variable symbols of SPARQL expressions in the surrounding scope and embed the LSE invocation into Jadex. During the implementation of the SmartFactory case study in Jadex, this functionality turned out to be one of the essential benefits over the manual implementation in Jadex. The manual implementation of SPARQL queries in Jadex XML files is very error-prone. The manual implementation of one SPARQL query (e.g. a context condition of a plan) sometimes took more than one hour (caused by typos and the debugging effort). One example for the code produced by

	BOCHICA	ISReal Extension	
Concepts	124	8 (1 language)	7.3%
QVT Rules	137	16	11.7%
XPand Rules	70	6	8.6%
Total Rules	207	22	10.6%
QVT LOC	2795	576	20.6%
XPand LOC	1350	150	11.1%
Total LOC	4145	726	17.5%

Table 8.1: This table compares the infrastructure provided by the BOCHICA framework (BOCHICA core DSL and Jadex base transformation) and the ISReal-specific extension model and extension transformation.

the helper functions is depicted in Figure 8.15. The base transformation provides the basic infrastructure and the extension transformation complements the existing code where necessary. The result is a modeling environment that is tailored to the design of intelligent ISReal agents.

Another important aspect to discuss is how good BOCHICA and the ISReal extension close the gap between design and code. Table 8.2 compares the generated and the manually written code. In order to compute the LOC values, the automatic Eclipse code formatter was applied to the Java and XML files. The numbers are without comments and blank lines. Moreover, ontology files and semantic service descriptions are not considered. They are referenced by the `OMSConfig` concept of the ISReal extension model. In total, the SmartFactory implementation in Jadex consists of 6796 LOC. 6312 LOC have been automatically generated by the Jadex base transformation and the ISReal extension transformation. Additionally, 484 LOC were implemented manually. The manual code mainly encompasses business logic, such as the computation of the shortest path between two objects or the evaluation of message content in negotiations. In BOCHICA, the business logic is represented by InternalTasks. The model transformation separates the manual changes from the generated code. The third row of Table 8.2 depicts the portion of code that has been generated by the ISReal extension transformation (e.g. SPARQL queries to the LSE or object service invocations). One can take the gray parts in Figure 8.15 as example for the code generated by the extension transformation. The ISReal-specific code (20.6%) is an estimate for the manual changes required for developing the system using BOCHICA without the ISReal extension. However, it is important to note that the ISReal-specific code is tightly integrated with the rest of the implementation. Thus, it is hardly possible to separate the manual code from the generated one. This would potentially lead to problems during code regeneration (similar to Prometheus). Moreover, several

	Java	XML	Total	
Total LOC	3327	3469	6796	100%
Base Transf. LOC	2357	2554	4911	72.3%
Ext. Transf. LOC	486	915	1401	20.6%
Custom LOC	484	0	484	7.1%

Table 8.2: This table compares the total LOC (Java and XML code) for implementing the SmartFactory case study (first row) with (i) the generated code of the base transformation (second row), (ii) the generated code of the extension transformation (third row), and (iii) the manually written code (last row). In total, 92.9% of the code could be generated.

ISReal-specific aspects cannot be modeled without the Bochica extension, so that the model would be incomplete.

Prometheus and the PDT modeling tool were chosen as representative for the methodology-oriented approaches to AOSE (see Section 8.1.3). By comparing the model artifacts presented in Sections 8.1.2 and 8.1.3, one can see that the code produced by the Prometheus approach requires extensive manual modifications, as (i) the Prometheus models are less expressive (e.g. they do not cover plan bodies and several aspects are specified in natural language) and (ii) the models do not have ISReal-specific concepts (e.g. no LSE, no SPARQL integration, and no ISReal object service orchestration). One benefit of the Prometheus approach is that it supports the user in collecting the requirements of the system (e.g. using scenarios). This functionality could be provided by a Prometheus plug-in to Bochica. Jack was chosen as representative of the platform-specific approaches (see Section 8.1.4). Our experiences show that the implementation of ISReal applications in Jack and Jadex have similar difficulties (e.g. the integration of knowledge base into the agent). The agent engineer has to over and over again write the same glue-code for integrating the ISReal library into the agent platform. Moreover, the integration of SPARQL queries is very error prone. MDSD has the benefit that a well designed modeling language abstracts from those technical details.

As discussed in Section 2.2.3, the majority of the existing platform-independent modeling languages were created in order to support a certain agent methodology. For example, methodology-oriented languages support design artifacts which are not directly used for code generation (e.g. scenarios in Prometheus). One problem of existing methodology-oriented modeling approaches that we see is that they do not clearly distinguish between the methodology and the modeling language. Two indicators which support our perception are (i) the development of the modeling languages is not decoupled from the methodologies and (ii) none of the languages has an own name (only the tools have names). However, in our opinion the development of modeling languages is orthogonal to the development of agent methodologies and tools. Of course, a methodology can (and most likely will) have certain requirements to a modeling language (e.g. own methodology artifacts and views). Moreover, the methodology languages fail in addressing features of a certain target platform which makes extensive manual code modification necessary. This potentially causes design and code to diverge over time. One further drawback is the insufficient expressiveness of many methodology languages. Unfortunately, the majority of the developed modeling tools are only partially based on standardized technology for model-driven development which hampers the benefits of MDSD. For example, the *Prometheus Design Tool* (Prometheus methodology) has no explicit underlying metamodel. Others, like AgentTool III (O-MaSE), INGENIAS Development Kit (INGENIAS), TAOM4E (Tropos) are only partially based on standardized technology for MDSD (e.g. proprietary or non-MDA-based model transformations). To the best of our knowledge, the mentioned approaches do not consider extensibility as presented by this dissertation. Besides the methodology-based modeling languages, there exist also approaches for extending the *Unified Modeling Language* (UML) with agent concepts (e.g. OMG's *Agent Metamodel and Profile* (AMP) or FIPA Agent UML). Those approaches promise to reuse the ecosystem built around UML – including the large user group. However, modeling agents is fundamentally different from modeling objects. Agents possess an internal architecture and require different methods and design patterns. Moreover, our experiences in AMP showed that it is hard to extend UML for AOSE since the underlying *Meta Object Facility* (MOF) metamodel is complex and extensions of existing elements have many not desired and non-obvious implications. Thus, we are sceptical that extending UML in its current form suffices the needs of AOSE. UML, which is a general purpose modeling language, offers two extension mechanisms: (i) heavy weight metamodel extensions and (ii) light weight profiles. Metamodel extensions of UML underlie the standardization process of OMG and are not for the normal end user. Profile-based extensions can be created by end users and allow a limited customization. An alternative to our approach would be the creation of a platform-specific modeling language (e.g. for the ISReal-enabled Jadex platform). This would mean to reinvent many things that are already part of Bochica. Two platform-specific examples are Jadex DE and SEAGENT DE (see Section 2.2.4). The possibility to customize the languages as the agent platform (e.g. Jadex) is integrated into a larger execution environment is not discussed. Our approach is especially suited for large scale applications or target environments with many end-users (e.g. the ISReal platform) where customizations pay off. Small applications can be realized with the functionality provided by the core modeling language and the base transformations (similar to existing approaches).

We see Bochica complementary to existing methodology-oriented approaches as it provides a clean conceptual framework and extension interfaces.

8.2 IRL Case Study

This section evaluates the model-driven method for reverse engineering BDI agents introduced in Chapter 6. The case study is based on a demonstrator developed in a collaboration between the *German Federal Ministry of Education and Research* (BMBF)-founded research projects ISReal and *Semantic Product Memory*[73] (SemProM). One of the main objectives of SemProM was to equip products with a *Digital Product Memory* (DPM) that collects product-related information throughout a product's life cycle (e.g. supply chain, cooling chain, best-before date). In order to identify single entities, products are equipped with *Radio-Frequency IDentification* (RFID) tags. Every station a product passes in its life cycle has to maintain the according data records in the DPM (see Figure 8.25). The project involved the DFKI IUI and IRL research groups as well as industry partners. As part of the collaboration between the ISReal project (DFKI ASR) and the SemProM project (DFKI IUI, IRL), a prototype for agent-based maintenance of the DPM in a supermarket environment was developed. The underlying idea was to assign each product in the supermarket environment an according agent that proactively collects sensor data, notifies the management in case of problems, and updates the DPM. My task in the cooperation was to design the MAS. Moreover, Pascal Liedtke (DFKI ASR) was responsible for the XML3D-based visualization, and Gerrit Kahl (IRL) and Jens Haupert (DFKI IUI) were involved as representatives of the SemProM project.

The developed demonstrator was directly implemented using Jadex. One of the reasons was that Bochica and the Jadex tool stack were not mature enough at that time. Thus, the SemProM demonstrator is representative for the majority of today's agent-based systems which are also not yet making use of model-driven AOSE. The method for model-driven reverse engineering for MAS presented in Chapter 6 can be used (i) to visualize and analyze the underlying structure of an implemented MAS and (ii) to extract model artifacts for later reuse. It can also be used as staring point for migrating a system to model-driven AOSE. Moreover, the reverse engineering method drastically increases the number of available models for Bochica. In the following, Section 8.2.1 provides an overview of the IRL and the developed demonstrator. Afterwards, Section 8.2.2 evaluates the reverse engi-

[73] `http://www.semprom.org/semprom_engl/`

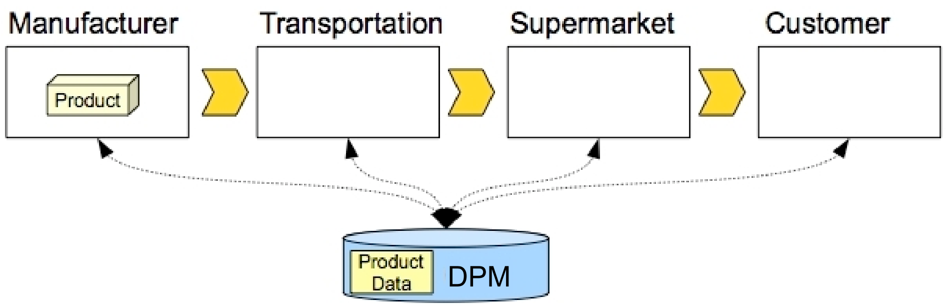


Figure 8.25: This figure depicts the stations a product passes throughout its life cycle. The digital product memory has to be maintained by the different stations.

neering approach in the SemProM case study. Finally, section 8.2.3 discusses the experiences and sets them into context to the related work.

8.2.1 The IRL Demonstrator

This section provides a brief overview of the developed demonstrator. The supermarket environment considered by the demonstrator was provided by the DFKI IRL living lab. It is co-located with Globus GmbH in St. Wendel, Germany. The living lab provides the typical infrastructure of a supermarket and is used by DFKI for developing and evaluating new technology in a realistic retail environment. Every product instance (e.g. a single pizza) is identified by an RFID tag throughout its life cycle. The different product types have individual requirements regarding the required sensor data for updating the DPM. For example, deep-frozen products need temperature data in order to observe the distribution cold chain. As of today, products are usually not equipped with own sensors and thus, the environment has to provide the data (e.g. a freezer's temperature sensor). The requirements for the demonstrator were:

- **Agent-based Guidance:** Every product of the supermarket should act proactively by collecting sensor data according to the product's type and maintain its DPM. Devices collaborate with products by providing the sensor data. The supermarket management has to be notified as a problem is recognized (e.g. product placement and violation of constraints).
- **Management Cockpit:** The management cockpit is responsible for monitoring the products and devices in the supermarket environment. It uses a 3D Web front-end for visualizing the current situation. The product constraints are configured using the management cockpit. Products have to update their status and location in the management cockpit as something changes.

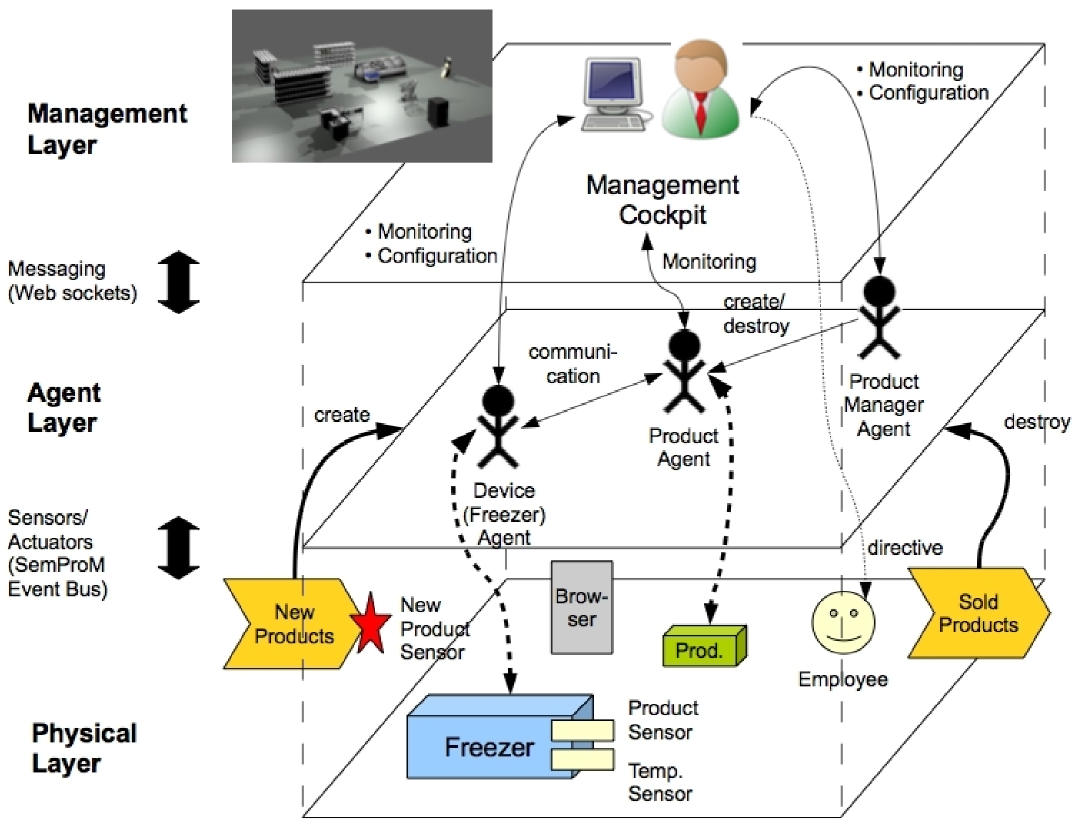


Figure 8.26: The *physical layer* is the real supermarket environment. The *agent layer* runs the agents that control the devices and products of the physical layer. Finally, the *management layer* uses an XML3D-based Web application to monitor and manage the supermarket environment.

- **Policies:** Policies are used to configure products and devices in the supermarket environment (e.g. minimum quantity of a product type in a shelf; warning x days before the best before day is reached; notification for misplaced products; violation of contractor agreements for the presentation of certain brands).

Figure 8.26 depicts an overview of the developed system architecture. The *physical layer* encompasses the (real) supermarket environment including devices, products, shelves, and employees. The entities are equipped with sensors. The sensor data is collected and processed by according agents running in the *agent layer*. Finally, the management cockpit is an XML3D-based Web front-end for monitoring the situation in the supermarket and for configuring the entities. Three different agent types have been realized: *Product agents* represents a single product instance and are responsible for (i) collecting the sensor data in the environment, (ii) updating the product memory accordingly, and (iii) reporting problems to the

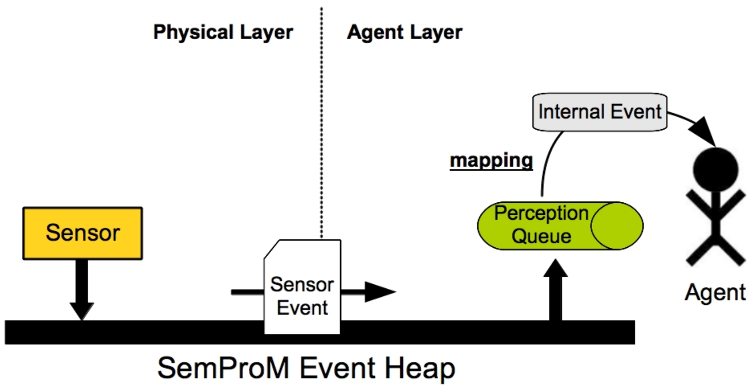
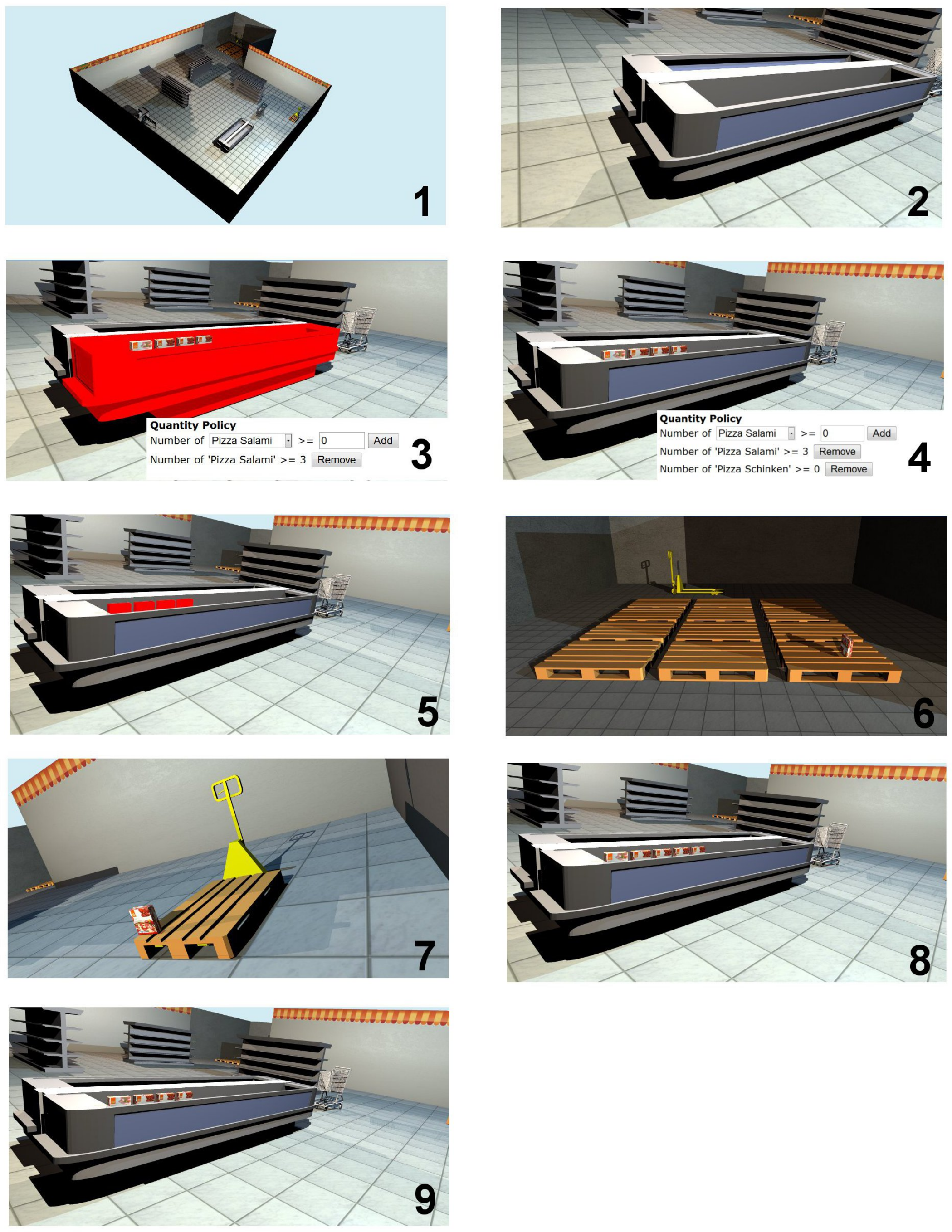


Figure 8.27: The sensors put sensor data on the event heap. Perception queues are registered for certain events and map them to according events of the agent system.

management cockpit. The *management agent* is responsible for creating a new product agent as a new product ID is sensed in the supermarket. Likewise, it destroys the agent as the product got payed. Finally, a *device agent* is responsible (i) to manage the products under its responsibility and (ii) to apply the device policies for configuring the device's behavior. In this case study, freezers are the only considered devices. The agents report their current states and events to the management cockpit. The system is currently running as a demonstrator in the IRL and is being further developed. As one can easily imagine, one agent per product easily leads to thousands of agents in a normal supermarket environment. However, the underlying idea behind the design decision was that in the near future RFID chips will have enough computing power to process the data on the product itself. Thus, the architecture of the demonstrator is a step into this direction.

Finally, we want to provide an overview of the different sensors. The infrastructure of the IRL provides an event heap where all sensors post their sensor data (see Figure 8.27). Agents register themselves for certain sensor events. The *perception queue* is responsible for mapping relevant sensor events to events of the agent system. Agents process this information and react to the changed situation. Supported sensors are (i) temperature sensors that provide temperature information in regular intervals, (ii) proximity sensors that sense RFID tags in their environment, (iii) room-change sensors that are located at doors, and (iv) product-payed sensors. Figure 8.28 depicts an example session of the developed system.



Quantity Policy
Number of Pizza Salami >= 0 Add
Number of 'Pizza Salami' >= 3 Remove

Quantity Policy
Number of Pizza Salami >= 0 Add
Number of 'Pizza Salami' >= 3 Remove
Number of 'Pizza Schinken' >= 0 Remove

Figure 8.28: Demonstrator overview (management cockpit)[74]: (1) overview of the supermarket, (2) empty freezer before initialization, (3) initialized freezer and violated device policy (misplaced product "Pizza Schinken"), (4) fixed device policy, (5) product temperature warning, (6) new pizza arrives at stock, (7) pizza transported to salesroom, (8) pizza added to freezer, (9) pizza removed from freezer and payed.

[74]This figure was created by Pascal Liedtke as part of the collaboration.

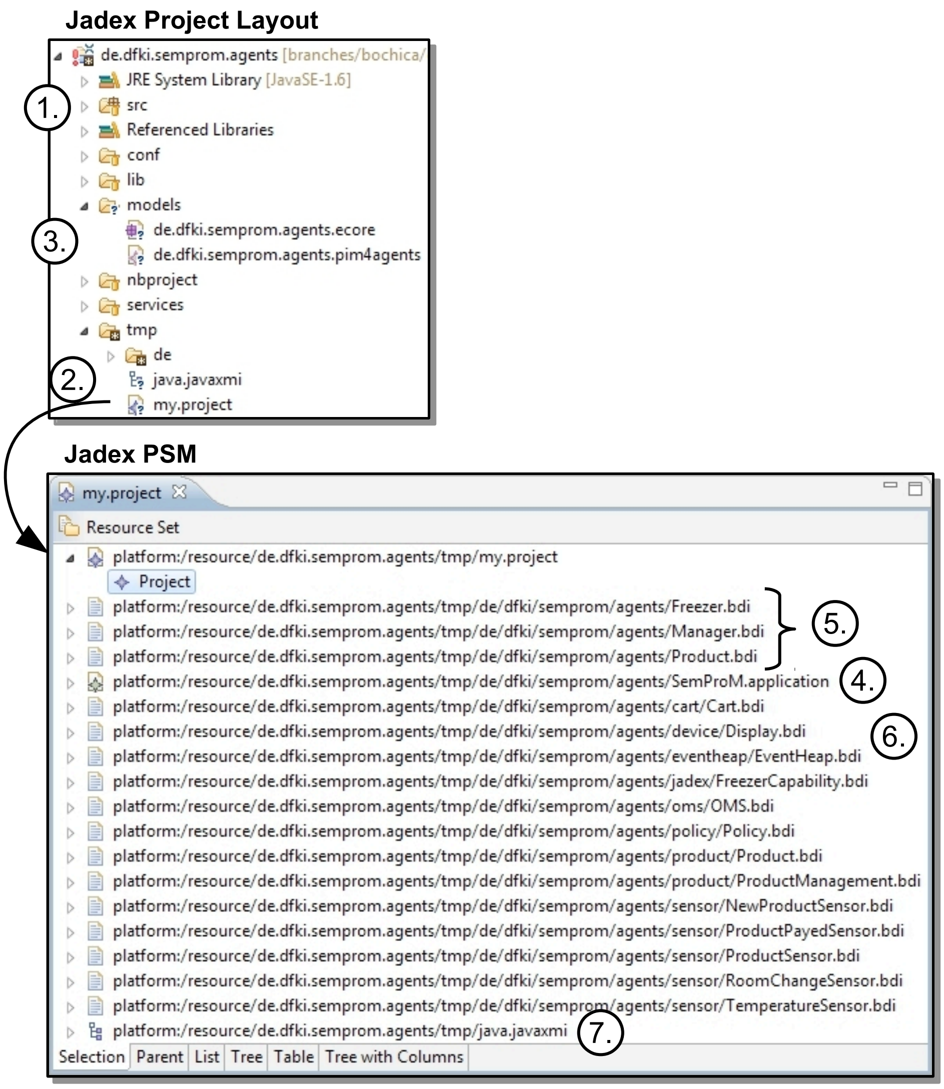
8.2.2 Reverse Engineering

As introduced in Chapter 6, the reverse engineering process consists of (i) lifting the platform artifacts to the Jadex PSM and (ii) the application of the design extraction rules. In total, the developed SemProM demonstrator encompassed one Jadex application, three agent types, 13 capabilities, and 58 Java-based files. The lifting of the source code to the Jadex PSM is done by the model extractor. Figure 8.29 depicts an overview of the overall project structure and the Jadex PSM. In the following, we discuss the Bochica artifacts extracted by the mapping rules presented in Chapter 6. The remainder of this section is structured into (i) agent types, (ii) goals and events, (iii) behaviors, (iv) the deployment configuration, and (v) the data model.

Agent and Capability Types

The underlying design of the agent types of the SemProM demonstrator is mapped by MR^R-6 to according agent definitions in Bochica. Figure 8.30 depicts an overview of the extracted artifacts of the Freezer agent. The agent diagram shows the agent's behaviors (mapped by MR^R-13) and capabilities (MR^R-7). The right-hand side of Figure 8.30 additionally depicts the agent's `Knowledges` (MR^R-8), `PlanUses` (MR^R-11), and `CapabilityUses` (MR^R-12). For example, the Freezer agent's internalTemp and externalTemp beliefs (see Figure 8.30 (4)) hold the current internal and external temperature values as measured by the freezer's sensors. The TemperatureSensor capability equips an agent with the behaviors and events for connecting it to a temperature sensor in the supermarket. For this purpose, the capability maintains a belief called currentTemperature. As depicted in Figure 8.30 (5), the Freezer agent imports the TemperatureSensor capability twice (once for the internal and once for the external temperature). The `CapabilityUses` bind the agent's internalTemp and externalTemp beliefs to the capabilities' currentTemperature beliefs. Thus, the capabilities automatically maintain the internal and external temperature beliefs of the Freezer agent.

Likewise, the ProximitySensor capability encapsulates the behaviors and events for connecting to and processing the data of a proximity sensor. The Freezer agent uses the ProximitySensor capability to sense the products contained by its freezer device. For this purpose, the capability (i) listens to sensor events delivered by the event heap and maps them to according events of the agent, and (ii) processes the events and updates the capability's sensedProducts belief accordingly. The `CapabilityUse` of the Freezer agent imports the sensedProducts belief and makes it available within the Freezer agent as containedProducts `Knowledge`. The Freezer agent has additional behaviors which are built on top of the functionality provided

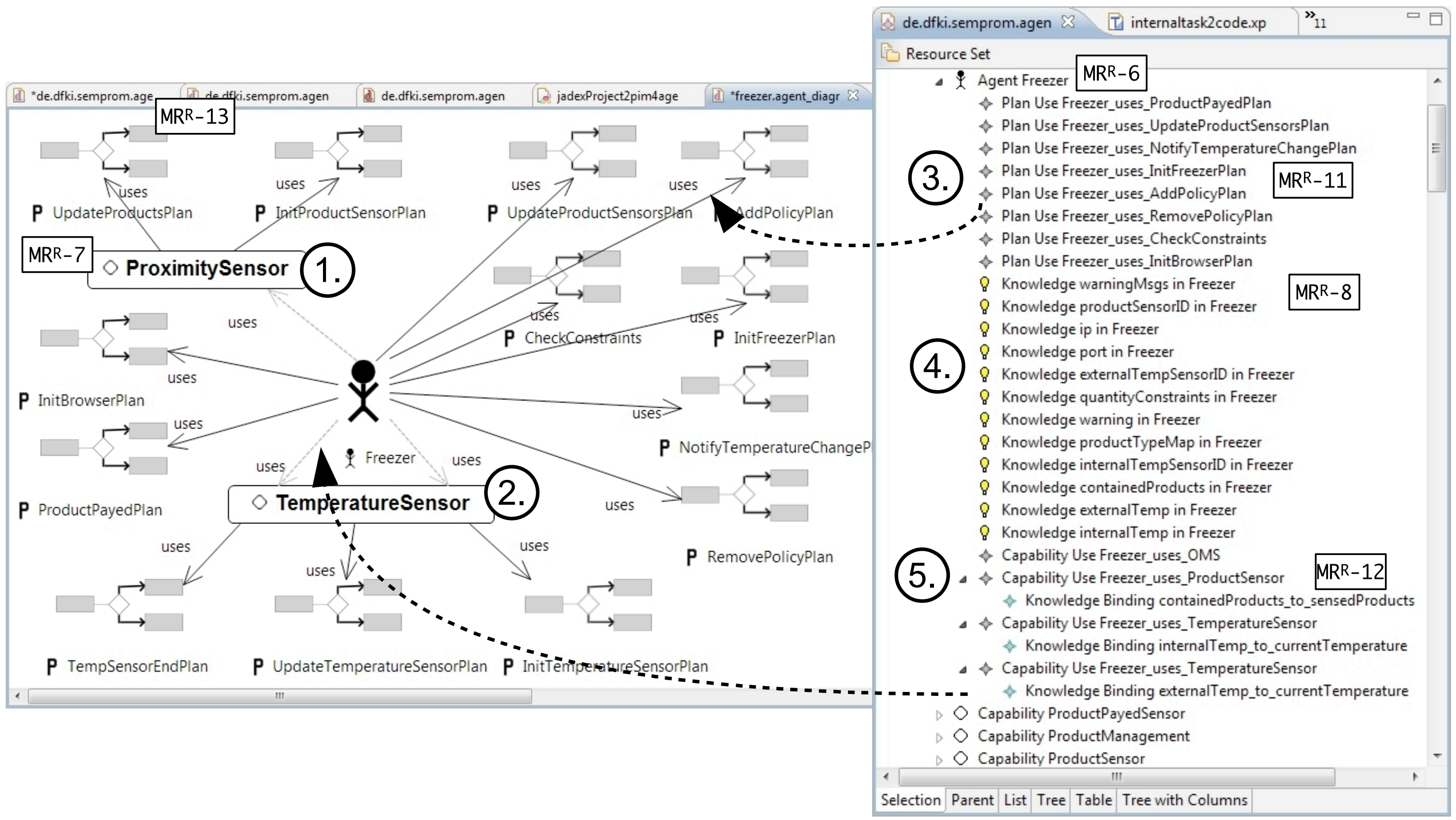


```
de.dfki.semprom.agents [branches/bochica/
  JRE System Library [JavaSE-1.6]
  src
  Referenced Libraries
  conf
  lib
  models
    de.dfki.semprom.agents.ecore
    de.dfki.semprom.agents.pim4agents
  nbproject
  services
  tmp
    de
    java.javaxmi
    my.project
```

```
my.project
Resource Set
platform:/resource/de.dfki.semprom.agents/tmp/my.project
  Project
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/Freezer.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/Manager.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/Product.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/SemProM.application
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/cart/Cart.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/device/Display.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/eventheap/EventHeap.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/jadex/FreezerCapability.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/oms/OMS.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/policy/Policy.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/product/Product.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/product/ProductManagement.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/sensor/NewProductSensor.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/sensor/ProductPayedSensor.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/sensor/ProductSensor.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/sensor/RoomChangeSensor.bdi
platform:/resource/de.dfki.semprom.agents/tmp/de/dfki/semprom/agents/sensor/TemperatureSensor.bdi
platform:/resource/de.dfki.semprom.agents/tmp/java.javaxmi
```

Figure 8.29: The figure at the top depicts the SemProM Jadex project. It shows (1) the source folder containing the Jadex source code, (2) the "tmp" folder containing the lifted Jadex PSM, and (3) the resulting BOCHICA model and data model. The figure at the bottom depicts the details of the Jadex PSM. It shows (4) the application model, (5) the agent models, (6) the capability models, and (7) the Java model.

by the imported capabilities. For example, the NotifyTemperatureChangePlan notifies the management cockpit about a temperature change. Moreover, the Freezer agent uses several plans for managing policies and products. For example, the InitFreezerPlan is responsible for initializing the sensors and connecting the agent to the management cockpit. The CheckConstraints plan is responsible for checking the current device policies (e.g. temperature constraints or misplaced products) and sending according warnings to the management cockpit. The extracted arti-



Agent Freezer
Plan Use Freezer_uses_ProductPayedPlan
Plan Use Freezer_uses_UpdateProductSensorsPlan
Plan Use Freezer_uses_NotifyTemperatureChangePlan
Plan Use Freezer_uses_InitFreezerPlan
Plan Use Freezer_uses_AddPolicyPlan
Plan Use Freezer_uses_RemovePolicyPlan
Plan Use Freezer_uses_CheckConstraints
Plan Use Freezer_uses_InitBrowserPlan
Knowledge warningMsgs in Freezer
Knowledge productSensorID in Freezer
Knowledge ip in Freezer
Knowledge port in Freezer
Knowledge externalTempSensorID in Freezer
Knowledge quantityConstraints in Freezer
Knowledge warning in Freezer
Knowledge productTypeMap in Freezer
Knowledge internalTempSensorID in Freezer
Knowledge containedProducts in Freezer
Knowledge externalTemp in Freezer
Knowledge internalTemp in Freezer
Capability Use Freezer_uses_OMS
Capability Use Freezer_uses_ProductSensor
Knowledge Binding containedProducts_to_sensedProducts
Capability Use Freezer_uses_TemperatureSensor
Knowledge Binding internalTemp_to_currentTemperature
Capability Use Freezer_uses_TemperatureSensor
Knowledge Binding externalTemp_to_currentTemperature
Capability ProductPayedSensor
Capability ProductManagement
Capability ProductSensor

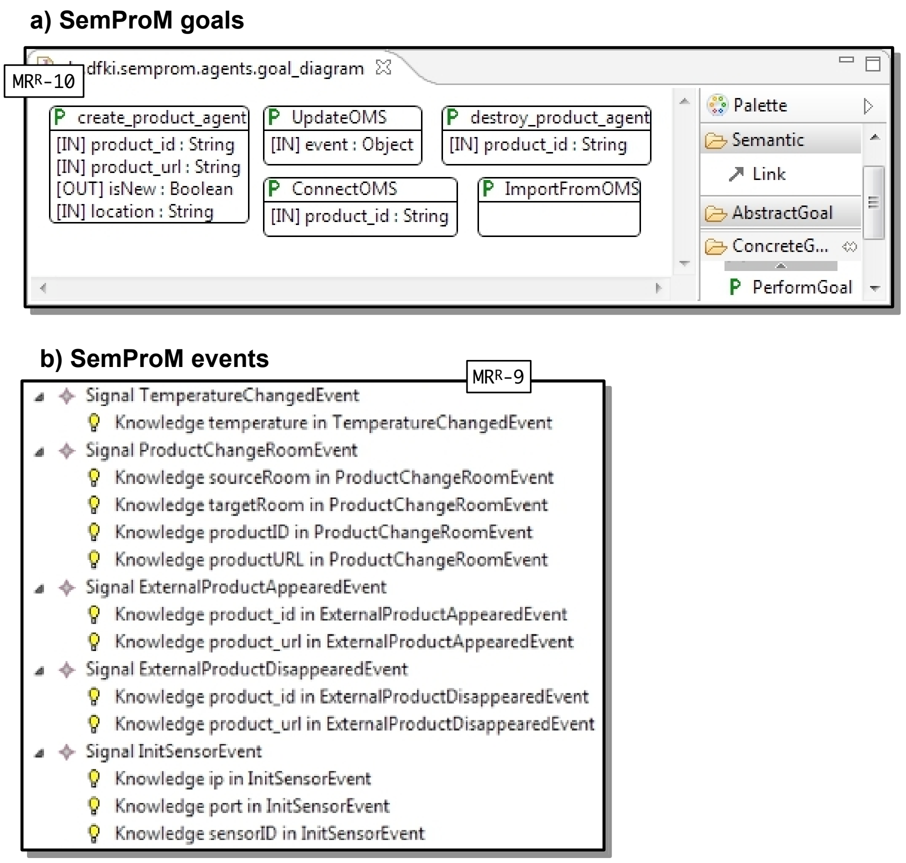
Figure 8.30: This figure depicts an overview of the extracted artifacts related to the freezer agent. It shows (1) the ProximitySensor capability, (2) the TemperatureSensor capability, (3) the agent's `PlanUses`, (4) the agent's `Knowledges`, and (5) the agent's `CapabilityUses`.

facts visualized by Figure 8.30 already provide a detailed overview of the overall structure of the system and the information passed between agents, capabilities, and plans.

Events and Goals

By extracting the relationship between behaviors, events, and goals, large parts of the structural dependencies of a MAS can be made reusable. Figure 8.31 depicts an overview of some of the extracted event and goal types of the SemProM demonstrator. For example, Figure 8.31 a) shows the `PerformGoal` create_product_agent of the Manager agent. It triggers the agent's behavior for initializing a new Product agent as a product is sensed for the first time. Moreover, it declares input and output parameters. For example, the product_id and product_url are read from the product's RFID chip and used to identify the product and access the product's DPM. The goal types are mapped by MR^{R}-10 and the parameters by MR^{R}-24. The agents of the IRL case study mainly make use of internal events for triggering behaviors. Thus, the goal hierarchy extraction is limited in this example. Section 8.2.3 provides an additional example to discuss the goal hierarchy extraction. Figure 8.31 b) depicts some of the extracted `Signals` mapped by MR^{R}-9. The events' parameters are mapped by MR^{R}-24 to `Knowledges` in Bochica. For example, the TemperatureChangedEvent is used by the TemperatureSensor capability to update the `currentTemperature` belief as a temperature change was recognized. The UpdateTemperatureSensorPlan is triggered by the TemperatureChangedEvent and performs the task.

Message events are used within the SemProM demonstrator for the communication between devices, products, and the manager agent. Figure 8.32 a) depicts an example communication for the initialization of a new product agent. The communication has been logged by the management cockpit at runtime. As discussed in Chapter 6, the extraction of interaction protocols from Jadex source code is only partially possible since Jadex has no explicit representation of interactions. Messages in Jadex can be assembled within Java code. The agent developer has to manage conversations on his own. Every Jadex capability that declares `MMessageEvents` is mapped by MR^{R}-22 to a Bochica `ProtocolConfiguration`. The `ProtocolConfiguration` holds the message events mapped by MR^{R}-23. Figure 8.32 b) depicts two example `ProtocolConfigurations`. The ProductProtocol was mapped form the Product capability of the Product agent and the ProductManagementProtocol from the ProcutManagement capability of the Manager agent. It is important to note that there exist at least two corresponding message declarations for each message type (one for sending and one for receiving the message).



P create_product_agent
[IN] product_id : String
[IN] product_url : String
[OUT] isNew : Boolean
[IN] location : String

P UpdateOMS
[IN] event : Object

P destroy_product_agent
[IN] product_id : String

P ConnectOMS
[IN] product_id : String

P ImportFromOMS

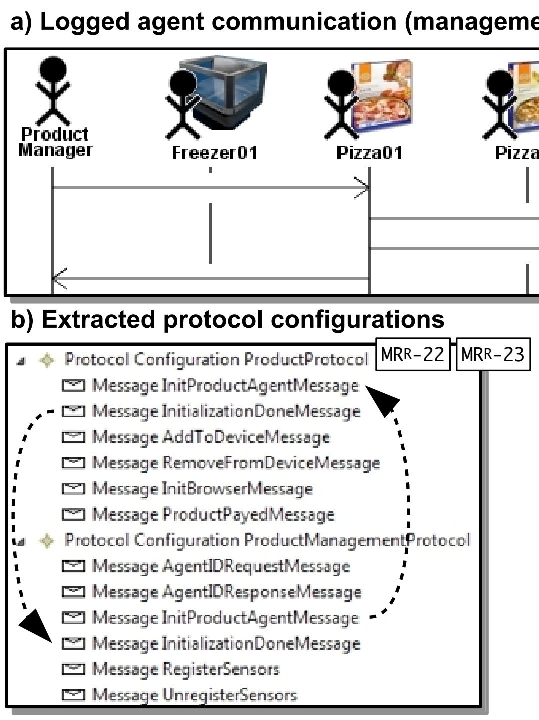
```
Signal TemperatureChangedEvent
    Knowledge temperature in TemperatureChangedEvent
Signal ProductChangeRoomEvent
    Knowledge sourceRoom in ProductChangeRoomEvent
    Knowledge targetRoom in ProductChangeRoomEvent
    Knowledge productID in ProductChangeRoomEvent
    Knowledge productURL in ProductChangeRoomEvent
Signal ExternalProductAppearedEvent
    Knowledge product_id in ExternalProductAppearedEvent
    Knowledge product_url in ExternalProductAppearedEvent
Signal ExternalProductDisappearedEvent
    Knowledge product_id in ExternalProductDisappearedEvent
    Knowledge product_url in ExternalProductDisappearedEvent
Signal InitSensorEvent
    Knowledge ip in InitSensorEvent
    Knowledge port in InitSensorEvent
    Knowledge sensorID in InitSensorEvent
```

Figure 8.31: The goals, events, and messages of the SemProM demonstrator.

Figure 8.32 b) visualizes this relationship for the InitProductAgentMessage and InitializationDoneMessage. The messages correspond to the communication depicted in Figure 8.32 a). However, the relationship between the messages is not explicitly represented at code level and was only monitored at runtime. Moreover, the matching names are not a generic indicator that two message declarations correspond to each other. Jadex uses pattern matching for matching message events to message type declarations of agents and capabilities. Thus, `ProtocolConfigurations` are candidates for manual refinement.

Behaviors

As introduced in Section 4.2, a Jadex behavior consists of an XML-based header and a Java-based plan body. Figure 8.33 depicts the Manager agent's ResolveProductIDToAgentPlan for resolving a product ID (stored on a RFID tag) to an existing product agent. The behavior is triggered by the AgentIDRequestMessage which has the product ID and product URL as parameters. Device agents can use the manager's service for resolving sensed product IDs to agents. If no existing

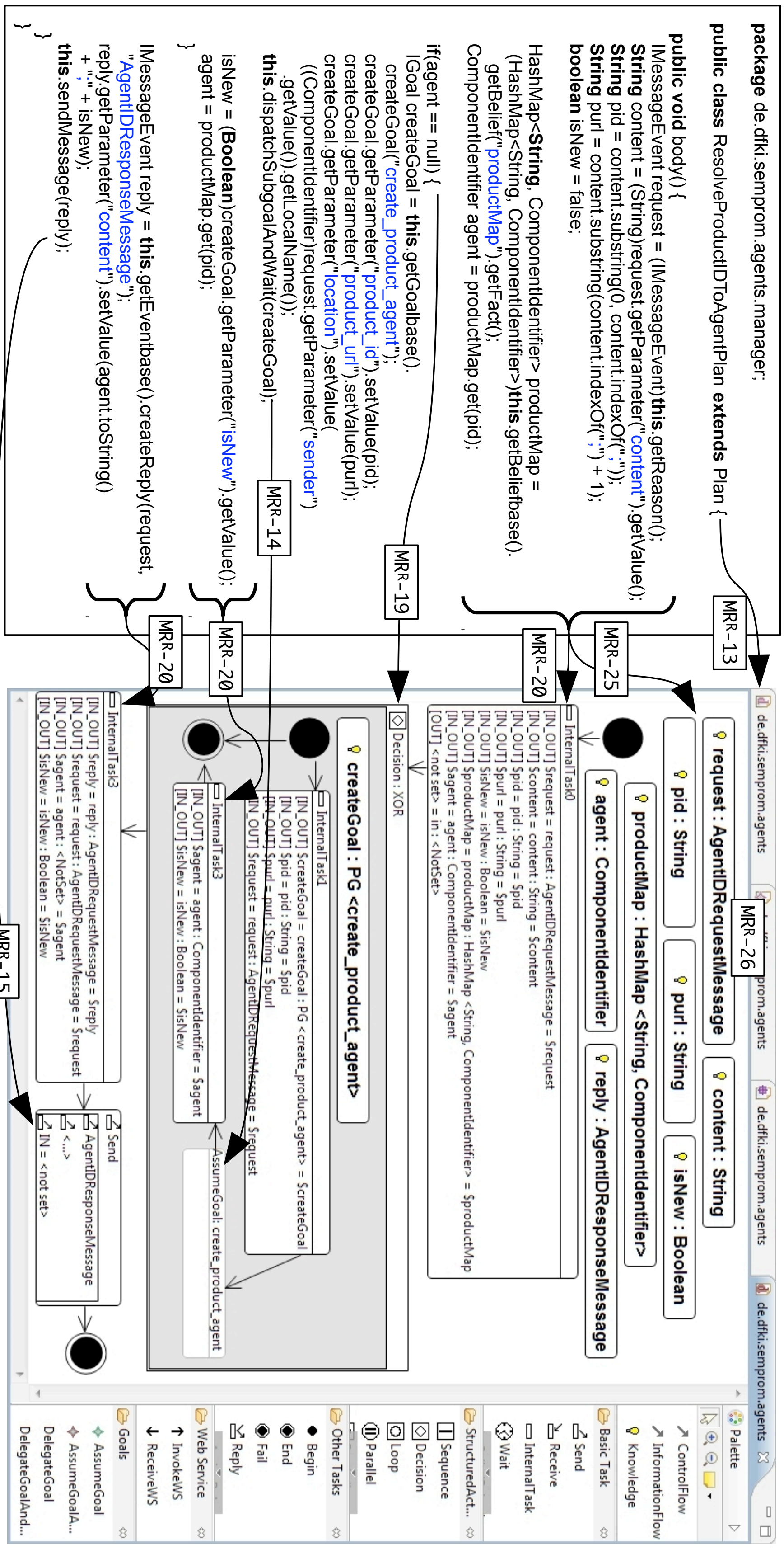

a) Logged agent communication (management cockpit)
Product Manager
Freezer01
Pizza01
Pizza02
DPM
ID
1
2
3
4
b) Extracted protocol configurations
MRR-22
MRR-23

Protocol Configuration ProductProtocol
- Message InitProductAgentMessage
- Message InitializationDoneMessage
- Message AddToDeviceMessage
- Message RemoveFromDeviceMessage
- Message InitBrowserMessage
- Message ProductPayedMessage

Protocol Configuration ProductManagementProtocol
- Message AgentIDRequestMessage
- Message AgentIDResponseMessage
- Message InitProductAgentMessage
- Message InitializationDoneMessage
- Message RegisterSensors
- Message UnregisterSensors

Figure 8.32: Figure a) depicts the logged communication for the initialization of a new product agent: (1) the manager creates and initializes the new product agent, (2) the product agent initializes itself from the product memory, (3) the product agent updates the product memory for the new location, and (4) the product agent confirms the successful creation. Figure b) shows the extracted message types in Bochica.

product agent can be found by the manager agent, it will initialize a new one and assign it to the product. The behavior is mapped by MR^{R}-13 to a Bochica behavior. The first part of the behavior consists of a number of variable declarations (concept `VariableDeclarationStatement`). They are mapped by MR^{R}-25 to `Knowledges` of the Bochica plan. For example, the request variable stores the incoming request message. Algorithm 6.1 iterates over the Java statements of the body() method and invokes the according mapping rules. For example, the `IfStatement` is mapped by MR^{R}-19 to a Bochica `Decision`. The decision uses the productMap knowledge (type HashMap) to check whether there already exists an agent for the product ID. If not, the plan creates a ProductAgent and returns the according ID. The actual creation of the product agent is done by posting a create_product_agent `PerformGoal`. The AssumeGoal task is mapped by MR^{R}-14. The initialization of the goal is done by InternalTask1 (mapped by MR^{R}-20). Finally, the AgentIDResponseMessage is initialized and sent as response to the requester. A sequence of statements that could not be mapped to Bochica



```
package de.dfki.semprom.agents.manager;

public class ResolveProductIDToAgentPlan extends Plan {

  public void body() {
    IMessageEvent request = (IMessageEvent)this.getReason();
    String content = (String)request.getParameter("content").getValue();
    String pid = content.substring(0, content.indexOf(";"));
    String purl = content.substring(content.indexOf(";") + 1);
    boolean isNew = false;

    HashMap<String, ComponentIdentifier> productMap =
      (HashMap<String, ComponentIdentifier>)this.getBeliefbase().
        getBelief("productMap").getFact();
    ComponentIdentifier agent = productMap.get(pid);

    if(agent == null) {
      IGoal createGoal = this.getGoalbase().
        createGoal("create_product_agent");
      createGoal.getParameter("product_id").setValue(pid);
      createGoal.getParameter("product_url").setValue(purl);
      createGoal.getParameter("location").setValue(
        ((ComponentIdentifier)request.getParameter("sender")
          .getValue()).getLocalName());
      this.dispatchSubgoalAndWait(createGoal);

      isNew = (Boolean)createGoal.getParameter("isNew").getValue();
      agent = productMap.get(pid);
    }

    IMessageEvent reply = this.getEventbase().createReply(request,
      "AgentIDResponseMessage");
    reply.getParameter("content").setValue(agent.toString()
      + ";" + isNew);
    this.sendMessage(reply);
  }
}
```

```
InternalTask0
[IN_OUT] $request = request : AgentIDRequestMessage = $request
[IN_OUT] $content = content : String = $content
[IN_OUT] $pid = pid : String = $pid
[IN_OUT] $purl = purl : String = $purl
[IN_OUT] $isNew = isNew : Boolean = $isNew
[IN_OUT] $productMap = productMap : HashMap <String, ComponentIdentifier> = $productMap
[IN_OUT] $agent = agent : ComponentIdentifier = $agent
[OUT] <not set> = in : <NotSet>

InternalTask1
[IN_OUT] $createGoal = createGoal : PG <create_product_agent> = $createGoal
[IN_OUT] $pid = pid : String = $pid
[IN_OUT] $purl = purl : String = $purl
[IN_OUT] $request = request : AgentIDRequestMessage = $request

InternalTask3
[IN_OUT] $agent = agent : ComponentIdentifier = $agent
[IN_OUT] $isNew = isNew : Boolean = $isNew

InternalTask3
[IN_OUT] $reply = reply : AgentIDRequestMessage = $reply
[IN_OUT] $request = request : AgentIDRequestMessage = $request
[IN_OUT] $agent = agent : <NotSet> = $agent
[IN_OUT] $isNew = isNew : Boolean = $isNew

Send
AgentIDResponseMessage
<...>
IN = <not set>
```

Figure 8.33: This figure depicts the ResolveProductIDToAgentPlan. The left hand side shows the Java source code and the right hand side the resulting BOCHICA plan.

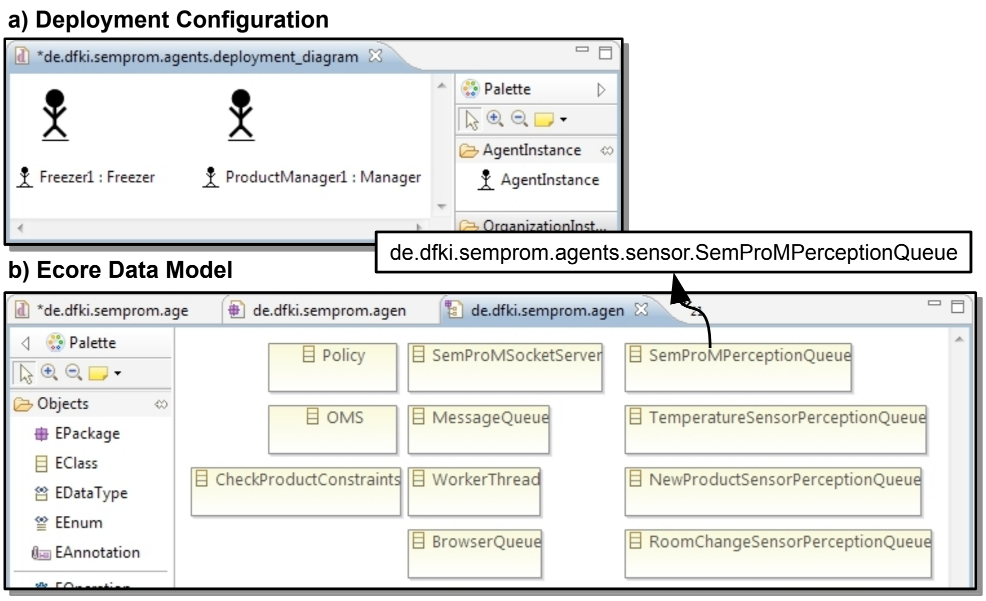


Figure 8.34: a) depicts the deployment configuration of the IRL demonstrator. It encompasses the two agent instances ProductManager1 of type Manager and Freezer1 of type Freezer. b) depicts the extracted IRL data model.

concepts is aggregated and mapped by MR^{R}-20 to an `InternalTasks` with input/output parameters for the accessed variables. This example mapping shows how the basic structure of a Jadex behavior is mapped to Bochica.

Deployment

Figure 8.34 a) depicts the deployment configuration of the developed demonstrator. The agent instances are mapped by MR^{R}-3 to Bochica. The system initially only consists of the manager agent and one freezer agent. The product agents are initialized as they are sensed by the manager or the freezer during runtime. The initial configuration of an agent instance's beliefs and goals (e.g. sensor IDs, and IP and port of the management cockpit) is mapped by MR^{R}-4 and MR^{R}-5 to Bochica `Knowledge-` and `GoalInitializers`.

Data Model

Figure 8.34 b) depicts a part of the data types that have been extracted by MR^{R}-26 and MR^{R}-27 from source code to Ecore. Every class uses the `instanceClassName` attribute to store a fully qualified reference to the Java type. For example, the SemProMPerceptionQueue implements the base class for mapping events of the IRL event heap to an agent's internal events. The class points to the `de.dfki.semprom.agents.sensor.SemProMPerceptionQueue` class. The extracted data types

	Jadex (source)	Jadex PSM	Bochica	Jadex (target)
Application	1	1	1	1
Agent Type	3	3	3	3
Capability	13	13	8	41 (8 + 33)
Behavior	38	38	33	33
Instance	2	2	2	2

Table 8.3: This table compares the model artifacts of the SemProM application at the different stages of the reverse transformation. The last column depicts the model artifacts after a forward transformation has been applied to Bochica (without previous refinements).

are used within the Bochica model (e.g. parameter types and knowledge types). The fully qualified type is used by a forward transformation to resolve the platform classes. A further example is the SemProMSocketServer. It realizes a socket server connection to the XML3D-enabled Web browser.

8.2.3 Summary and Discussion

This section applied the reverse transformation introduced in Chapter 6 to an already implemented Jadex application. Section 6.6 already discussed the main conceptual differences between the Jadex platform and the Bochica core DSL. In the following, we summarize and further analyze the made experiences.

Table 8.3 depicts an overview of the model artifacts of the SemProM demonstrator at the different stages of the reverse transformation process. The first column summarizes the original platform artifacts of the Jadex project. The second column shows the model artifacts of the Jadex PSM after the lifting. The third column depicts the extracted Bochica model artifacts after the application of the design extraction rules presented in Chapter 6. Finally, the fourth column summarizes the artifacts that were generated by a forward transformation to Jadex (based on the extracted Bochica model). The forward transformation was performed on the Bochica model without any previous refinements. One can see that the different model artifacts are lifted one-to-one from Jadex source code to the Jadex PSM presented in Section 4.2. The design extraction rules presented in Chapter 6 extracted eight capabilities and 33 behaviors. The mismatch between the source/target behaviors and capabilities is caused by five capabilities and behaviors which are part of the project but not used by the existing agent types. For example, one capability was created for a shopping cart agent that guides a customer through the supermarket environment. Due to the limited duration of the project, the focus was on the freezer device and the product agent. Thus,

	Jadex (source)	Jadex PSM	Bochica	Jadex (target)
Capabilities	4	4	4	12 (4 + 8)
Beliefs	12	12	12	12
Behaviors	8	8	8	–
Msg./Int.Evt.	7/6	7/6	–/6	–/6
Goals	–	–	–	–

Table 8.4: This table compares the model artifacts belonging to the freezer agent (`Freezer.agent.xml` file) at the different stages of the reverse transformation. The last column depicts the model artifacts in Jadex after a forward transformation has been applied to Bochica (without previous refinements).

the shopping cart capability was not used in the final demonstrator. Therefore, the reverse transformation did not consider the capability since it was not used by any agent type. Likewise, the five behaviors were not mapped. Finally, the forward transformation to Jadex creates one capability for every extracted plan. The underlying pattern was introduced as part of the forward transformation presented in Chapter 5. The reasoning behind it was that Bochica behaviors can be complex and the created Jadex capability groups multiple generated artifacts into one self-contained component with a clean interface. The plan capabilities are imported by the agent.

Table 8.4 depicts the model artifacts belonging to the freezer agent of the IRL case study. The freezer agent consists of four capabilities, twelve beliefs, eight behaviors, seven message events, and six internal events. As already discussed, the freezer agent does not make use of goals in the current implementation. One can see that all artifacts are lifted one-to-one to the Jadex PSM. The design extraction step maps the Jadex concepts to the Bochica core DSL. One difference discussed in Chapter 6 is that Jadex `MMessageEvents` are mapped to `ProtocolConfigurations` in Bochica. Thus, the message events are separated from the agent. The forward transformation to Jadex creates a separate messaging capability that declares the message events and implements the protocol behavior. However, in order to generate the messaging code, additional refinements are required at PIM level. The agent's behaviors are externalized to capabilities during the forward transformation. Finally, the agent imports those capabilities (see Chapter 5 for details).

Since the SemProM demonstrator makes only limited use of goals, Figure 8.35 depicts an extracted goal hierarchy of the *Mars World Classic (MWC)* application. The MWC application is part of the official Jadex distribution. Section 4.2 already used the MWC example to provide an overview of a typical Jadex application. The walk_around and produce_ore goals have the move_dest goal as sub-goal (visualized

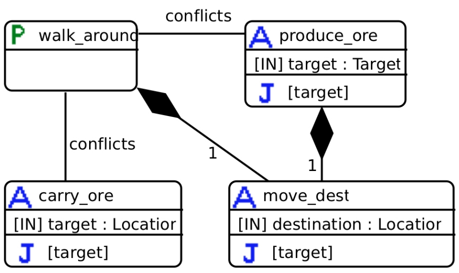


Figure 8.35: This figure depicts an extracted goal hierarchy. The black diamonds represent the sub-goal relationship.

by the black diamond). The sub-goal relationship has been computed by the design extraction rules presented in Chapter 6. Moreover, one can see that there are conflicts between the walk_around and the carry_ore and produce_ore goals (meaning, they cannot be performed at the same time). Those conflicts have been mapped from the Jadex `inhibits` relationship. Finally, one can see the parameters of the goals and their data types. The goal diagram is well suited to get an overview of a system.

Refinement. The reverse engineering approach presented in this dissertation focuses on lifting the platform artifacts to the Jadex PSM and the conceptual mappings from Jadex to Bochica. However, in the following we want to provide an outlook on how the extracted models can be refined with high-level artifacts. This step is important to prepare the artifacts for later reuse (e.g. to transfer the design to other use cases and agent platforms). Figure 8.36 depicts the basic work-flow for refining an extracted Bochica model. (1) depicts the extracted domain roles and agent types of the reverse transformation. In step (2), model artifacts (e.g. an interaction protocol - here the request response protocol) are imported from a model repository. Step (3) adds the IRLOrganization to the model for specifying the collaboration between the involved parties using interaction protocols. In step (4), the refined model is validated (e.g. using OCL-based constraints). Finally, the model is committed to a model repository in step (5). Of course, the refinement also requires the specification of the collaborations of agents (e.g. the role bindings inside an organization) and the behaviors of agents. In previous work, we already showed that models based on Dsml4Mas can be mapped to other agent platforms, such as Jack and Jade (see [Hahn et al., 2009a]).

In the following, we set our reverse engineering approach into context to the related work. In Favre [2010], a general overview of MDRE is provided. Chikofsky and Cross II [1990] provide a taxonomy of reverse engineering. Reverse engi-

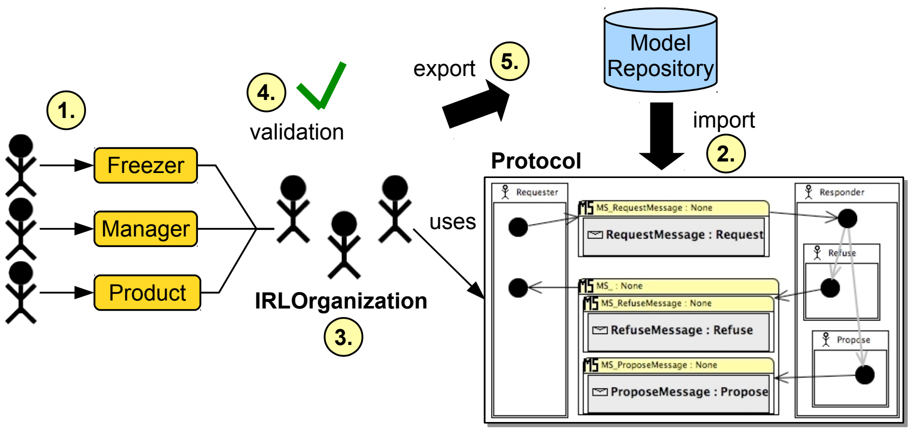


Figure 8.36: This figure depicts a general overview of the refinement process. The process consists of (1) the extraction of model artifacts by the reverse transformation, (2) the import of model artifacts form a model repository (e.g. interaction protocols), (3) the refinement of organizational structures, behaviors, etc., (4) a model validation step, and (5) the export of design artifacts to the model repository.

neering of agents differs from reverse engineering of object-oriented systems since additional agent-oriented artifacts have to be considered (e.g. organizational structures, goals, interaction protocols). According to Gómez-Sanz et al. [2008], the *INGENIAS Development Kit* (IDK) offers a component called *code uploader* that is used for synchronizing INGENIAS models with code that has been generated from those models (forward engineering). No further information is provided how this component works or which aspects are covered. Similar to IDK, the *Prometheus Development Tool* (PDT) offers code synchronization functionality for generated code [Padgham et al., 2007]. The mechanism is not further detailed. Both, IDK and PDT, support only forward engineering. To the best of our knowledge, there exists no MDRE approach for MAS. In general, there exists not much work regarding agents and reverse engineering. A. Hirst [2000] proposed a reverse engineering approach for Soar[75] agents. He focused on reverse engineering of production rules. In Lam and Barber [2005], an approach for software comprehension was presented. The idea was to observe agents at runtime (their execution traces) and reconstruct their structures. A similar approach could be used to collect additional information about the organizational structures of agents at runtime (e.g. to reconstruct the interaction protocol depicted in Figure 8.32). Agent-oriented methodologies also do not cover reverse engineering.

[75]`http://sitemaker.umich.edu/soar/home`

8.3 Summary

This chapter presented two case studies for the evaluation of the Bochica approach to AOSE. The SmartFactory case study evaluated the ISReal-enabled Bochica framework for the design of intelligent agents that operate a virtual representation of the DFKI SmartFactory. The evaluation results show that the ISReal extension model and extension transformation significantly narrow the gap between design and code. The ISReal extension enabled the generation of additional 20% of code compared to the Jadex base transformation. Only 7% had to be implemented manually. The Bochica approach was compared to one methodology-oriented modeling approach (Prometheus) and one platform-specific approach (Jack). It was shown that the Bochica core DSL is more expressive than Prometheus. Compared to the platform-specific approach, Bochica provides an abstraction layer that enables the engineer to focus on the overall design of the system, instead of dealing with technical details. The second case study evaluated the model-driven method for reverse engineering of Jadex agents. The reverse engineering approach was applied to the IRL demonstrator which has been implemented as a collaboration between the DFKI ASR and IRL. The evaluation showed that large parts of the model artifacts could be extracted. It was discussed how to refine the model artifacts for later reuse.

Chapter 9

Conclusion

With this dissertation, I propose a novel model-driven framework for engineering multiagent systems. The Bochica framework goes beyond the state-of-the-art in AOSE as it combines the benefits of a platform-independent modeling language with the possibility of addressing selected features of target platforms and application domains. Chapter 1 motivates this dissertation and provides an overview of the research questions and contributions. Afterwards, Chapter 2 discusses the state-of-the-art in MDSD and model-driven AOSE. The Bochica framework and the underlying design decisions are introduced in Chapter 3. Moreover, the extension interfaces and the alignment to existing methodologies are discussed. Chapter 4 presents the metamodel of the Bochica core DSL and a PSM for the Jadex BDI platform. Conceptual mappings from Bochica to Jadex are specified in Chapter 5. The mappings build the foundation for the forward transformation to Jadex. A reverse transformation for extracting the underlying design of BDI agents is introduced in Chapter 6. For this purpose, conceptual mappings between the Jadex PSM and the Bochica core DSL are specified (in upward direction). In order to enable Bochica for modeling agents in semantically-enhanced virtual worlds, Chapter 7 introduces an according extension model and extension transformation. Chapter 8 presents two case studies which evaluate the framework for modeling agents that operate a virtual representation of the SmartFactory and for extracting the underlying design of an already implemented MAS. The Bochica framework has been implemented as an Eclipse-based development environment. The development environment will be published at Sourceforge[76]. In the following, I will pick up the research questions of the introduction and summarize the contributions that this dissertation offers to answering them.

[76]`http://sourceforge.net/projects/bochica`

What are the core concepts of an expressive agent-oriented modeling language?

Formal and expressive agent-oriented modeling languages are required by model-driven AOSE to capture abstract design decisions and to project the created models to a target platform. Chapter 2 provides an overview of the diverse field of agent-oriented modeling languages. The strength of the Bochica core DSL is the tight integration of the different parts of a MAS specification. In order to increase the expressiveness, Chapter 3 discusses how often neglected aspects, such as variable scoping, knowledge bases, data models, and software languages, have been integrated into the modeling language. Furthermore, it is shown how to utilize OCL invariants for reducing ambiguities and increasing the expressiveness. Chapter 4 introduces the metamodel underlying the Bochica core DSL. The SmartFactory case study presented in Chapter 8 evaluates the Bochica framework for modeling agents that operate a virtual production line. The case study shows that the ISReal-enabled Bochica core DSL is expressive enough to generate 93% of the code automatically (including orchestration of services, interaction protocols, etc.). The manual implementation is limited to business logic (e.g. processing message content or computing the shortest path). Moreover, Bochica is compared to one methodology-driven and one platform-specific approach.

How can agent-oriented modeling languages better support concrete application domains and execution environments?

Besides the expressiveness of a modeling language, the possibility to address the concepts of a custom application domain or execution environments is important to capture the design decisions for a SUC. The Bochica framework approaches this problem with a platform-independent core DSL and several extension interfaces. The extension interfaces are specified in Chapter 3. Furthermore, I propose an iterative adaptation process for gradually incorporating conceptual extensions into the framework. In order to enable Bochica for the application domain of agents in semantically-enriched virtual worlds, Chapter 7 proposes an extension model which complements Bochica with the missing concepts. The ISReal extension model encompasses eight new concepts and one reasoning language – compared to 124 concepts of the Bochica core DSL. The evaluation in Chapter 8 shows that the ISReal extension model raised the amount of automatically generated code by around 20%. By using the extension mechanism, large parts of the infrastructure, which is common to most MAS, can be reused to create a customized modeling environment.

How to effectively close the gap between platform-independent agent models and concrete code?

As a part of Chapter 3, I have introduced an extensible transformation architecture for mapping Bochica models to a target platform. A Bochica model transformation is separated into a base transformation which covers the concepts of the Bochica core DSL and a complementary extension transformation which handles the concepts of an extension model. The idea is that a base transformation can be reused across application domains. An extension transformation is created as a part of the iterative adaptation process presented in Chapter 3. In order to enable the Jadex BDI platform for the Bochica approach, Chapter 4 introduces a PSM for Jadex. In Chapter 5, I specify conceptual mappings from Bochica to the Jadex PSM that build the foundation of the Jadex base transformation. As a part of this dissertation, an extension model and extension transformation for agents in semantically-enhanced virtual worlds have been introduced in Chapter 7. The extension transformation complements the Jadex base transformation with ISReal-specific conceptual mappings. The evaluation in Chapter 8 shows that around 11% custom mapping rules (compared to the base transformation) are sufficient to cover the ISReal-specific concepts. The conceptual extensions raise the amount of generated code by around 20%. In total, it was possible to automatically generate 93% of the overall code. Due to the used design patterns, the manually written code (around 7%) could be separated from the generated code (see Chapter 5).

How to make the underlying design of concrete implemented multiagent systems reusable?

The majority of existing agent-based systems has been implemented without MDSD in mind. In Chapter 6, I have introduced a model-driven reverse engineering approach for extracting the underlying design of already implemented MAS. The approach consists in (i) lifting the code to the platform-specific level and (ii) a design extraction step. In order to enable Jadex for MDSM according to MDA, Chapter 4 introduces a PSM for the BDI agent platform Jadex. Based on the Jadex PSM, I have specified conceptual mappings (design extraction rules) from the Jadex PSM to the Bochica core DSL in Chapter 6. In this chapter, I also discuss the conceptual mismatches between Jadex and Bochica. The case study presented in Chapter 8 applies the design extraction rules to an already implemented Jadex-based application. By comparing the design artifacts of the different reverse engineering stages it is shown that large parts of the underlying design can be extracted. Moreover, it is discussed how to refine the extracted model artifacts and prepare them for later reuse. The presented agent-oriented MDRE approach drastically increases the number of potential models available to model-driven AOSE and BOCHICA.

The BOCHICA framework is especially suited for projects, execution environments, and application domains with many end users (agent engineers who make use of an extension). The ISReal platform, for instance, can be used in a large variety of virtual reality applications so that the customization effort will soon pay off. One obstacle that might stand in the way of applying BOCHICA is the effort for getting comfortable with MDSD and the framework itself. The creation of an extension model requires a developer to be comfortable with metamodeling, model transformation, and the concepts and interfaces defined by BOCHICA. Until today, most agent researchers and developers are not familiar with those technologies. My personal experience with carrying out the SmartFactory case study shows that BOCHICA shifts the focus from technical details to the design of the overall system. Thus, the experience of designing a system differs from a classical code-centered approach. Small software projects can (i) make use of existing extensions or (ii) use the functionality of the BOCHICA core DSL and the existing base transformations (similar to existing agent-oriented modeling approaches). My expectation is that bundle providers providing ready-to-use combinations of conceptual extensions, software languages, views, and model transformations for certain target environments become available.

Future Work. Up to today, AOSE is still driven by research and has not yet reached main stream software engineering. One of the main problems is that there is no common terminology which is accepted throughout the agent community – this already starts with the definition of the concept *agent*. I see the BOCHICA core DSL as a nucleus which has the potential to build a basis for the unification of the wide area of AOSE. Metamodels are perfectly suitable for discussing (i) which concepts are relevant, (ii) how those concepts are defined, and (iii) how they relate to each other. At the same time, metamodels build the foundation of MDSD and can be used for transferring concepts from research to concrete software development. The framework proposed by this dissertation provides the infrastructure and conceptual core for extensions by external researchers. Ideally, the BOCHICA approach will lead to a metamodel becoming widely accepted within the agent community. Currently, BOCHICA covers the core concepts of MAS (from our point of view). External researchers working on certain sub-areas of MAS are likely to lack concepts specific to their research field (e.g. agent architectures and commitments). It would be highly interesting to integrate those demands into one consistent framework. Another important aspect which requires further research effort is the integration of the Bochica framework with existing agent-oriented methodologies. Currently, the agent-oriented methodologies are undergoing a consolidation phase in which methodologies are split-up into method fragments. The alignment of those method fragments to Bochica is a highly interesting task as it could lead to a holistic approach. Moreover, an integrated framework with a common conceptual core also reduces the maintenance costs of the tool chain. The proposed model-driven AORE approach enables agent engineers to extract the underlying design of concrete implemented (Jadex) applications to the platform-independent level. One of the next steps would be to develop refactoring methods which can be applied to the extracted model artifacts. Finally, the approach could be extended to a complete roundtrip engineering approach.

Bibliography

International Journal on Agent-Oriented Software Engineering (JAOSE), 1(1), 2007. ISSN 1746-1375.

AOS. *JACK Intelligent Agents Design Tool Manual. Release 5.3.* Agent Oriented Software Pty. Ltd., November 2011a. URL `http://www.aosgrp.com/documentation/jack/DesignTool_Manual_WEB/index.html`.

AOS. *JACK Intelligent Agents Graphical Plan Editor Manual. Release 5.3.* Agent Oriented Software Pty. Ltd., November 2011b. URL `http://www.aosgrp.com/documentation/jack/Plan_Editor_Guide_WEB/index.html`.

Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. A model driven development of platform-neutral agents. In *Multiagent System Technologies. 8th German Conference, MATES 2010, Leipzig, Germany, September 27-29, 2010, Proceedings*, volume 6251 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 3–14. Springer, 2010.

Roxana A. Belecheanu, Steve Munroe, Michael Luck, Terry Payne, Tim Miller, Peter McBurney, and Michal Pěchouček. Commercial applications of agents: Lessons, experiences and challenges. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1549–1555. ACM, 2006.

Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of the 4th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'99)*, pages 97–108. The Practical Application Company Ltd., 1999.

Ghassan Beydoun, Graham Low, Brian Henderson-Sellers, Haralambos Mouratidis, Jorge J. Gómez-Sanz, Juan Pavón, and Cesar Gonzalez-Perez. FAML: a generic metamodel for MAS development. *IEEE Transactions on Software Engineering*, 35(6):841–863, November 2009.

Barry W. Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, August 1986.

Michael E. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, 1987.

Lars Braubach, Alexander Pokahr, Kai Jander, Winfried Lamersdorf, and Birgit Burmeister. Go4Flex: goal-oriented process modelling. In *Intelligent Distributed Computing IV - Proceedings of the 4th International Symposium on Intelligent Distributed Computing (IDC'10), Tangier, Morocco, September 2010*, volume 315 of *Studies in Computational Intelligence (SCI)*, pages 77–87. Springer, 2010.

Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

Paolo Busetta, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. JACK intelligent agents - components for intelligent agents in Java, January 1999. AgentLink Newsletter Issue 19. ISSN 1465-3842.

Giovanni Caire, Wim Coulier, Francisco Garijo, Jorge Gómez-Sanz, Juan Pavón, Paul Kearney, and Philippe Massonet. The Message methodology. In *Methodologies and Software Engineering for Agent Systems. The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 177–194. Kluwer Academic Publishers.

Giovanni Caire, M. Porta, E. Quarantotto, and G. Sacchi. Wolf – an Eclipse plug-in for WADE. In *Proceedings of the 17th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '08)*, pages 26–32. IEEE Computer Society, 2008.

Xiaoqi Cao. Model-driven agent-based orchestration of services. Master Thesis. Saarland University, 2011.

Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

Paolo Ciancarini and Michael J. Wooldridge, editors. *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Limerick, Ireland, June 10, 2000, Revised Papers*, volume 1957 of *Lecture Notes in Computer Science (LNCS).* Springer, 2001.

Massimo Cossentino and Colin Potts. A CASE tool supported methodology for the design of multi-agent systems. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02)*, 2002.

Scott A. DeLoach. Engineering organization-based multiagent systems. In *Software Engineering for Multi-Agent Systems IV. Research Issues and Practical Applications*, volume 3914 of *Lecture Notes in Computer Science (LNCS)*, pages 109—125, 2006.

Scott A. DeLoach and Juan Carlos García-Ojeda. O-MaSE a customisable approach to designing and building complex, adaptive multi-agent systems. *International Journal of Agent-Oriented Software Engineering (IJAOSE)*, 4(3): 244–280, November 2010.

Scott A. DeLoach and Mark F. Wood. Developing multiagent systems with agentTool. In *Intelligent Agents VII. Agent Theories Architectures and Languages: 7th International Workshop (ATAL'00)*, volume 1986 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 46–60. Springer, 2001.

Oğuz Dikenelli. SEAGENT MAS platform development environment. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 1671–1672. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2008.

Oğuz Dikenelli, Riza Cenk Erdur, and Özgür Gümüs. SEAGENT: a platform for developing Semantic Web based multi agent systems. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 1271–1272. ACM, 2005.

Liliana Favre. *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution*. Engineering Science Reference, 2010.

Antonio Fernández-Caballero and José M. Gascueña. Developing multi-agent systems through integrating Prometheus, INGENIAS and ICARO-T. In *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 219–232. Springer, 2010.

FIPA. ACL Message Structure Specification. Foundation for Intelligent Physical Agents (FIPA), 2002a. URL `http://www.fipa.org/specs/fipa00061/SC00061G.html`. SC00061G.

FIPA. Communicative Act Library Specification. Foundation for Intelligent Physical Agents (FIPA), 2002b. URL `http://www.fipa.org/specs/fipa00037/SC00037J.html`. SC00037J.

FIPA. Contract Net Interaction Protocol Specification. Foundation for Intelligent Physical Agents (FIPA), 2002c. URL `http://www.fipa.org/specs/fipa00029/SC00029H.html`. SC00029H.

FIPA. Interaction Protocol Library Specification. Foundation for Intelligent Physical Agents (FIPA), 2003. URL `http://fipa.org/specs/fipa00025/DC00025F.html`. DC00025F.

FIPA. Agent Management Specification. Foundation for Intelligent Physical Agents (FIPA), 2004. URL `http://www.fipa.org/specs/fipa00023/SC00023K.html`. SC00023K.

Donald Firesmith and Brian Henderson-Sellers. *The OPEN Process Framework: An Introduction*. Addison-Wesley, 2001.

Klaus Fischer and Stefan Warwas. A methodological approach to model driven design of multiagent systems. In *Proceedings of the 13th International Workshop on Agent-Oriented Software Engineering (AOSE'12)*, pages 93–103. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2012.

Rubén Fuentes-Fernández, Jorge J. Gómez-Sanz, and Juan Pavón. Integrating agent-oriented methodologies with UML-AT. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1303–1310. ACM, 2006.

Ariel Fuxman, Marco Pistore, John Mylopoulos, and Paolo Traverso. Model checking early requirements specifications in Tropos. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pages 174–181. IEEE Computer Society, 2001.

Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1995.

Iván García-Magariño, Celia Gutiérrez, and Rubén Fuentes-Fernández. The INGENIAS development kit: A practical application for crisis-management. In

Bio-Inspired Systems: Computational and Ambient Intelligence, 10th International Work-Conference on Artificial Neural Networks, IWANN 2009, Salamanca, Spain, June 10-12, 2009. Proceedings, Part I, volume 5517 of *Lecture Notes in Computer Science (LNCS)*, pages 537–544. Springer, 2009.

Iván García-Magariño, Jorge J. Gómez-Sanz, and Juan Pavón. Representación de las relaciones en los metamodelos con el lenguaje ecore. In *Proceedings of Desarrollo del Software Dirigido por Modelos, MDA y Aplicaciones (DSDM'07)*, volume 1 of *Actas de Talleres de Ingeniería del Software y Bases de Datos*, pages 11–20, 2007.

Juan C. García-Ojeda, Scott A. DeLoach, Walamitien H. Oyenan, and Jorge Valenzuela. O-MaSE: a customizable approach to developing multiagent development processes. In *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 (LNCS)*, pages 1–15. Springer, 2008.

Juan C. García-Ojeda, Scott A. DeLoach, and Robby. agentTool process editor: supporting the design of tailored agent-based processes. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09)*, pages 707–714. ACM, 2009.

Francisco J. Garijo, Jorge J. Gómez-Sanz, and Philippe Massonet. The message methodology for agent-oriented analysis and design. In *Agent-Oriented Methodologies*. IGI Global.

José Manuel Gascueña and Antonio Fernández-Caballero. The INGENIAS methodology for advanced surveillance systems modelling. In *Nature Inspired Problem-Solving Methods in Knowledge Engineering, Second International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2007, La Manga del Mar Menor, Spain, June 18-21, 2007, Proceedings, Part II*, volume 4528 of *Lecture Notes in Computer Science (LNCS)*, pages 541–550. Springer, 2007.

José Manuel Gascueña and Antonio Fernández-Caballero. Prometheus and INGENIAS agent methodologies: A complementary approach. In *Agent-Oriented Software Engineering IX. 9th International Workshop, AOSE 2008 Estoril, Portugal, May 12-13, 2008 Revised Selected Papers*, volume 5386 of *Lecture Notes in Computer Science (LNCS)*, pages 131–144. Springer, 2009.

Michael P. Georgeff and François Félix Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 972—978. Morgan Kaufmann, 1989.

Paolo Giorgini, John Mylopoulos, and Roberto Sebastiani. Goal-oriented requirements analysis and reasoning in the Tropos methodology. *Engineering Applications of Artificial Intelligence*, 18(2):159–171, 2005.

Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The Tropos software development methodology: Processes, models and diagrams. In *Agent-Oriented Software Engineering III. Third International Workshop, AOSE 2002 Bologna, Italy, July 15, 2002 Revised Papers and Invited Contributions*, volume 2585 of *Lecture Notes in Computer Science (LNCS)*, pages 162–173. Springer, 2002.

Jorge J. Gómez-Sanz. *Modelado de Sistemas Multi-agente.* PhD thesis, Universidad Complutense de Madrid. Facultad de Informática, 2002.

Jorge J. Gómez-Sanz, Rubén Fuentes, Juan Pavón, and Ivan García-Magariño. INGENIAS development kit: a visual multi-agent system development environment. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 1675–1676. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2008.

Jorge J. Gómez-Sanz, Juan A. Botía Blaya, Emilio Serrano, and Juan Pavón. Testing and debugging of MAS interactions with INGENIAS. In *Agent-Oriented Software Engineering IX. 9th International Workshop, AOSE 2008, Estoril, Portugal, May 12-13, 2008, Revised Selected Papers*, volume 5386 of *Lecture Notes in Computer Science (LNCS)*, pages 199–212. Springer, 2009.

Christian Hahn, Cristián Madrigal-Mora, and Klaus Fischer. A platform-independent metamodel for multiagent systems. *International Journal on Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, 4 2009a.

Christian Hahn, Ingo Zinnikus, Stefan Warwas, and Klaus Fischer. From agent interaction protocols to executable code: a model-driven approach. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, pages 1199–1200. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2009b.

Christian Hahn, Ingo Zinnikus, Stefan Warwas, and Klaus Fischer. Automatic generation of executable behavior: a protocol-driven approach. In *Agent-Oriented Software Engineering X: 10th International Workshop, AOSE 2009, Budapest,* *Hungary, May 11-12, 2009, Revised Selected Papers*, volume 6038 of *Lecture Notes in Computer Science (LNCS)*, pages 110–124. Springer, 2011.

Brian Henderson-Sellers and Paolo Giorgini, editors. *Agent-Oriented Methodologies*. IGI Global, 2005.

Anthony J. Hirst. Reverse engineering of SOAR agents. In *Proceedings of the 4th International Conference on Autonomous Agents (Agents'00)*, pages 72–73. ACM, 2000.

Kai Jander and Winfried Lamersdorf. GPMN-Edit: high-level and goal-oriented workflow modeling. *Electronic Communications of the EASST*, 37, 2011.

Nicholas R. Jennings and Michael J. Wooldridge. Agent-oriented software engineering. *Artificial Intelligence*, 117:277—296, 2000.

Gerrit Kahl, Stefan Warwas, Pascal Liedtke, Lübomira Spassova, and Boris Brandherm. Management dashboard in a retail scenario. In *Proceedings of the IUI Workshop on Location Awareness for Mixed and Dual Reality (LAMDa'11)*, pages 22–25, 2011.

Patrick Kapahnke, Pascal Liedtke, Stefan Nesbigall, Stefan Warwas, and Matthias Klusch. ISReal: an open platform for semantic-based 3D simulations in the 3D internet. In *The Semantic Web - ISWC 2010: 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers*, pages 161–176. Springer, 2010.

Geylani Kardas, Erdem Eser Ekinci, Bekir Afsar, Oguz Dikenelli, and N. Yasemin Topaloglu. Modeling tools for platform specific design of Multi-Agent systems. In *Multiagent System Technologies. 7th German Conference, MATES 2009, Hamburg, Germany, September 9-11, 2009. Proceedings*, volume 5774 of *Lecture Notes in Computer Science (LNCS)*, pages 202–207. Springer, 2009.

Steven Kelly, Kalle Lyytinen, and Matti Rossi. MetaEdit+: a fully configurable multi-user and multi-tool CASE and CAME environment. In *Advanced Information Systems Engineering. 8th International Conference, CAiSE'96 Heraklion, Crete, Greece, May 2024, 1996 Proceedings*, volume 1080 of *Lecture Notes in Computer Science (LNCS)*, pages 1–21. Springer, 1996.

Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, 2008.

Per Kroll and Philippe Krutchten. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley, 2003.

Philippe Kruchten. *Rational Unified Process: An Introduction*. Addison-Wesley, 3rd edition, 2003.

John E. Laird. *The Soar Cognitive Architecture*. The MIT Press, 2012.

D. N. Lam and K. S. Barber. Comprehending agent software. In *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 586–593. ACM, 2005.

Esteban León-Soto. A model-driven approach for executing modular interaction protocols using BDI-agents. In *Agent-Based Technologies and Applications for Enterprise Interoperability. International Workshops ATOP 2009, Budapest, Hungary, May 12, 2009, and ATOP 2010, Toronto, ON, Canada, May 10, 2010, Revised Selected Papers*, volume 98 of *LNBIP*, pages 10–34. Springer, 2012.

Jürgen Lind. *Iterative Software Engineering for Multiagent Systems: The MASSIVE Method*, volume 1994 of *Lecture Notes in Artificial Intelligence (LNAI)*. Springer, 2001.

Cristián Madrigal-Mora, Esteban León-Soto, and Klaus Fischer. Implementing organisations in JADE. In *Multiagent System Technologies, 6th German Conference, MATES 2008, Kaiserslautern, Germany, September 23-26, 2008. Proceedings*, volume 5244 of *Lecture Notes in Computer Science (LNCS)*, pages 135–146. Springer, 2008.

Jez McKean, Hayden Shorter, Michael Luck, Peter McBurney, and Steven Willmott. Technology diffusion: analysing the diffusion of agent technologies. *Autonomous Agents and Multi-Agent Systems*, 17(3):372–396, 2008.

Brian G. Milnes, Garrett Pelton, Robert Doorenbos, Mike H. Laird, Paul Rosenbloom, and Allen Newell. A specification of the soar cognitive architecture in z. Technical report, Carnegie Mellon University, 1992.

Pavlos Moraïtis and Nikolaos I. Spanoudakis. Combining Gaia and JADE for multi-agent systems development. In *Proceedings of the 17th European Meeting on Cybernetics and Systems Research (EMCSR'04)*, 2004.

Pavlos Moraïtis, Eleftheria Petraki, and Nikolaos I. Spanoudakis. Engineering JADE agents with the Gaia methodology. In *Agent Technologies, Infrastructures, Tools, and Applications for E-Services, NODe 2002 Agent-Related Work-*

shops, Erfurt, Germany, October 7-10, 2002. Revised Papers, volume 2592 of *Lecture Notes in Computer Science (LNCS)*, pages 77—91. Springer, 2003.

Mirko Morandini. *Goal-Oriented Development of Self-Adaptive Systems.* PhD thesis, University of Trento. International Doctorate School in Information and Communication Technologies, 2011.

Mirko Morandini, Duy Cu Nguyen, Anna Perini, A. Siena, and Angelo Susi. Tool-supported development with Tropos: The conference management system case study. In *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 (LNCS)*, 2008.

Mirko Morandini, F. Migeon, M. Gleizes, C. Maurel, Loris Penserini, and Anna Perini. A goal-oriented approach for modelling self-organising MAS. In *Engineering Societies in the Agents World X. 10th International Workshop, ESAW 2009, Utrecht, The Netherlands, November 18-20, 2009. Proceedings*, volume 5881 of *Lecture Notes in Computer Science (LNCS).* Springer, 2009.

Mirko Morandini, Duy Cu Nguyen, Loris Penserini, Anna Perini, and Angelo Susi. Tropos modeling, code generation and testing with the TAOM4E tool. In *Proceedings of the 5th International i* Workshop 2011, Trento, Italy, August 28-29, 2011*, volume 766 of *CEUR Workshop Proceedings*, pages 172–174. CEUR-WS.org, 2011.

Jörg P. Müller. A cooperation model for autonomous agents. In *Intelligent Agents III Agent Theories, Architectures, and Languages. ECAI'96 Workshop (ATAL) Budapest, Hungary, August 1213, 1996 Proceedings*, volume 1193 of *Lecture Notes in Computer Science (LNCS)*, pages 245–260. Springer, 1997.

Jörg P. Müller and Markus Pischel. An architecture for dynamically interacting agents. *International Journal of Cooperative Information Systems (IJCIS)*, 3 (1):25–45, March 1994.

Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann, 2004.

Stefan Nesbigall, Stefan Warwas, Patrick Kapahnke, René Schubotz, Matthias Klusch, Klaus Fischer, and Philipp Slusallek. Intelligent agents for semantic simulated realities - the ISReal platform. In *ICAART 2010 - Proceedings of the International Conference on Agents and Artificial Intelligence, Volume 2 - Agents, Valencia, Spain, January 22-24, 2010*, pages 72–79. INSTICC Press, 2010.

OMG. MDA Guide. Version 1.0.1. Object Management Group (OMG), 2003. URL http://www.omg.org/cgi-bin/doc?omg/03-06-01. omg/2003-06-01.

OMG. MOF Model to Text Transformation Language. Version 1.0. Object Management Group (OMG), 2008a. URL http://www.omg.org/spec/MOFM2T/. formal/2008-01-16.

OMG. Software and Systems Process Engineering Meta-Model Specification. Version 2.0. Object Management Group (OMG), 2008b. URL http://www.omg.org/spec/SPEM/. formal/2008-04-01.

OMG. Business Process Model and Notation. Version 2.0. Object Management Group (OMG), 2011a. URL http://www.omg.org/spec/BPMN/. formal/2011-01-03.

OMG. Meta Object Facility (MOF) Core Specification. Version 2.4.1. Object Management Group (OMG), 2011b. URL http://www.omg.org/spec/MOF/. formal/2011-08-07.

OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1. Object Management Group (OMG), 2011c. URL http://www.omg.org/spec/QVT/. formal/2011-01-01.

OMG. Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1. Object Management Group (OMG), 2011d. URL http://www.omg.org/spec/UML/. formal/2011-08-06.

OMG. MOF 2 XMI Mapping Specification. Version 2.4.1. Object Management Group (OMG), 2011e. URL http://www.omg.org/spec/XMI/. formal/2011-08-09.

OMG. Object Constraint Language. Version 2.3.1. Object Management Group (OMG), 2012. URL http://www.omg.org/spec/OCL/. formal/2012-01-01.

Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide to Designing, Building, Implementing and Testing Agent Systems*. John Wiley & Sons, 2004.

Lin Padgham, John Thangarajah, and Michael Winikoff. Tool support for agent development using the Prometheus methodology. In *Proceedings of the 5th International Conference on Quality Software (QSIC'05)*, pages 383–388. IEEE Computer Society, 2005a.

Lin Padgham, Michael Winikoff, and David Poutakidis. Adding debugging support to the Prometheus methodology. *Engineering Applications of Artificial Intelligence*, 18(2):173–190, 2005b.

Lin Padgham, John Thangarajah, and Michael Winikoff. AUML protocols and code generation in the Prometheus design tool. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*, pages 1374–1375. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2007.

Lin Padgham, John Thangarajah, and Michael Winikoff. The Prometheus design tool - a conference management system case study. In *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 (LNCS)*, pages 197–211. Springer, 2008.

Lin Padgham, Michael Winikoff, Scott DeLoach, and Massimo Cossentino. A unified graphical notation for AOSE. In *Agent-Oriented Software Engineering IX. 9th International Workshop, AOSE 2008 Estoril, Portugal, May 12-13, 2008 Revised Selected Papers*, volume 5386 of *Lecture Notes in Computer Science (LNCS)*, pages 116–130. Springer, 2009.

Juan Pavón and Jorge J. Gómez-Sanz. Agent oriented software engineering with INGENIAS. In *Multi-Agent Systems and Applications III. 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003 Prague, Czech Republic, June 1618, 2003 Proceedings*, volume 2691 of *Lecture Notes in Computer Science (LNCS)*, pages 394–403, 2003.

Juan Pavón, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández. *Agent-Oriented Methodologies*, chapter The INGENIAS Methodology and Tools, pages 236–276. IGI Global, 2005.

Loris Penserini, Anna Perini, Angelo Susi, and John Mylopoulos. From stakeholder intentions to software agent implementations. In *Advanced Information Systems Engineering. 18th International Conference, CAiSE 2006, Luxembourg, June 5-9, 2006. Proceedings*, volume 4001 of *Lecture Notes in Computer Science (LNCS)*, pages 465–479, 2006.

Alexander Pokahr and Lars Braubach. From a research to an industrial-strength agent platform: Jadex V2. In *Business Services: Konzepte, Technologien, Anwendungen - 9. Internationale Tagung Wirtschaftsinformatik (WI 2009)*, pages 769–778. Österreichische Computer Gesellschaft, 2009.

Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer, 2005.

Alexander Pokahr, Lars Braubach, and Kai Jander. Unifying agent and component concepts - Jadex active components. In *Multiagent System Technologies, 8th German Conference, MATES 2010, Leipzig, Germany, September 27-29, 2010. Proceedings*, volume 6251 of *Lecture Notes in Computer Science*, pages 100–112. Springer, 2010.

Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann, 1991.

Walker W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering (ICSE'87)*, pages 328–338. IEEE Computer Society Press, 1987.

Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. XML3D: interactive 3D graphics for the web. In *Proceedings of the 15th International Conference on Web 3D Technology*, pages 175–184. ACM, 2010.

David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd revised edition, 2008.

Leon Sterling and Kuldar Taveter. *The Art of Agent-Oriented Modeling*. Intelligent Robotics and Autonomous Agents. The MIT Press, 2009.

Hongyuan Sun, John Thangarajah, and Lin Padgham. Eclipse-based Prometheus design tool. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, pages 1769–1770. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2010.

Angelo Susi, Anna Perini, John Mylopoulos, and Paolo Giorgini. The Tropos metamodel and its use. *Informatica*, 29:401–408, 2005.

Alexander Trenz. A generic approach for model-driven development of belief-desire-intention agents. Master Thesis. Saarland University, 2011.

W3C. RDF Primer. World Wide Web Consortium (W3C), 2004a. URL http://www.w3.org/TR/rdf-primer/. REC-rdf-primer-20040210.

W3C. XML Schema Part 1: Structures Second Edition. World Wide Web Consortium (W3C), 2004b. URL http://www.w3.org/TR/xmlschema-1/. REC-xmlschema-1-20041028.

W3C. RDFa Primer. World Wide Web Consortium (W3C), 2008a. URL http://www.w3.org/TR/xhtml-rdfa-primer/. NOTE-xhtml-rdfa-primer-20081014.

W3C. SPARQL Query Language for RDF. World Wide Web Consortium (W3C), 2008b. URL http://www.w3.org/TR/rdf-sparql-query/. REC-rdf-sparql-query-20080115.

W3C. OWL 2 Web Ontology Language Primer. World Wide Web Consortium (W3C), 2009. URL http://www.w3.org/TR/owl2-primer/. REC-owl2-primer-20091027.

Jos B. Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

Stefan Warwas. The BOCHICA framework for model-driven agent-oriented software engineering. In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence (ICAART'12), Revised Selected Papers, to be published*, Lecture Notes in Computer Science (LNCS). Springer, 2012.

Stefan Warwas and Christian Hahn. The concrete syntax of the platform independent modeling language for multiagent systems. In *Proceedings of the Agent-based Technologies and Applications for Enterprise Interoperability Workshop (ATOP'08)*, pages 94–105. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2008.

Stefan Warwas and Christian Hahn. The DSML4MAS development environment. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, pages 1379–1380. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2009a.

Stefan Warwas and Christian Hahn. The platform independent modeling language for multiagent systems. In *Agent-Based Technologies and Applications for Enterprise Interoperability. International Workshops, ATOP 2005 Utrecht, The Netherlands, July 25-26, 2005, and ATOP 2008, Estoril, Portugal, May 12-13, 2008, Revised Selected Papers*, number 25 in Lecture Notes in Business Information Processing (LNBIP), pages 129–153. Springer, 2009b.

Stefan Warwas and Matthias Klusch. Making multiagent system designs reusable: A model-driven approach. In *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'11)*, pages 101–108. IEEE Computer Society Press, 2011.

Stefan Warwas, Christian Hahn, and Klaus Fischer. A visual development environment for Jade (extended abstract). In *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, pages 1349–1350. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2009.

Stefan Warwas, Klaus Fischer, Matthias Klusch, and Philipp Slusallek. BOCHICA: A model-driven framework for engineering multiagent systems. In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence (ICAART'12)*, pages 109–118. SciTePress, 2012a.

Stefan Warwas, Matthias Klusch, Klaus Fischer, and Philipp Slusallek. A development environment for engineering intelligent avatars for semantically-enhanced simulated realities (demonstration). In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'12)*, pages 1479–1480. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2012b.

Gerhard Weiß, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

Gerhard Weiß and Ralf Jakob. *Agentenorientierte Softwareentwicklung: Methoden und Tools*. Springer, 2004.

Danny Weyns. *Architecture-Based Design of Multi-Agent Systems*. Springer, 2010.

WfMC. Process Definition Interface - XML Process Definition Language. Workflow Management Coalition (WfMC), 2008. URL `http://www.wfmc.org/xpdl.html`. WFMC-TC-1025.

Michael Winikoff. Defining syntax and providing tool support for Agent UML using a textual notation. *International Journal of Agent-Oriented Software Engineering*, 1(2):123–144, 2007.

Michael J. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2nd edition, 2009.

Michael J. Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Michael J. Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

Franco Zambonelli, Nicholas R. Jennings, and Michael J. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, 2003.

Zhiyong Zhang, John Thangarajah, and Lin Padgham. Automated unit testing for agent systems. In *Proceedings of the 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'07)*, pages 10–18. INSTICC Press, 2007.

Zhiyong Zhang, John Thangarajah, and Lin Padgham. Automated unit testing intelligent agents in PDT. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 1673–1674. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2008.