Universität des Saarlandes

A Platform-Independent Domain-Specific Modeling Language for Multiagent Systems

by

Dipl. Inform. Christian Steven Hahn

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der
Universität des Saarlandes
Saarbrücken, 2013

Tag des Kolloquiums: 20.12.2012

Zusammensetzung des Prüfungsausschusses:

Dekan:	Herr Prof. Dr. Mark Groves
Vorsitzender des Prüfungsausschusses:	Herr Prof. Dr. Joachim Weickert
Berichterstatter:	Herr Prof. Dr. Jörg Siekmann
	Herr Prof. Dr. Philipp Slusallek
	Herr Prof. Dr. Andreas Zeller
	Herr Prof. Dr. Bernhard Bauer
Akademischer Mitarbeiter:	Herr Dr. Klaus Fischer

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, 31.01.2013

Christian Steven Hahn

Short Abstract

Associated with the increasing acceptance of agent-based computing as a novel software engineering paradigm, recently a lot of research addresses the development of suitable techniques to support the agent-oriented software development. The state-of-the-art in agent-based software development is to (i) design the agent systems basing on an agent-based methodology and (ii) take the resulting design artifact as a base to manually implement the agent system using existing agent-oriented programming languages or general purpose languages like Java. Apart from failures made when manually transform an abstract specification into a concrete implementation, the gap between design and implementation may also result in the divergence of design and implementation.

The framework discussed in this dissertation presents a platform-independent domain-specific modeling language for MASs called DSML4MAS that allows modeling agent systems in a platform-independent and graphical manner. Apart from the abstract design, DSML4MAS also allows to automatically (i) check the generated design artifacts against a formal semantic specification to guarantee the well-formedness of the design and (ii) translate the abstract specification into a concrete implementation. Taking both together, DSML4MAS ensures that for any well-formed design, an associated implementation will be generated closing the gap between design and code.

Kurze Zusammenfassung

Aufgrund wachsender Akzeptanz von Agentensystemen zur Behandlung komplexer Problemstellungen wird der Schwerpunkt auf dem Gebiet der agentenorientierten Softwareentwicklung vor allem auf die Erforschung von geeignetem Entwicklungswerkzeugen gesetzt. Stand der Forschung ist es dabei das Agentendesign mittels einer Agentenmethodologie zu spezifizieren und die resultierenden Artefakte als Grundlage zur manuellen Programmierung zu verwenden. Fehler, die bei dieser manuellen Überführung entstehen, machen insbesondere das abstrakte Design weniger nützlich in Hinsicht auf die Nachhaltigkeit der entwickelten Softwareapplikation.

Das in dieser Dissertation diskutierte Rahmenwerk erörtert eine plattformunabhängige domänenspezifische Modellierungssprache für Multiagentensysteme namens DSML4MAS. DSML4MAS erlaubt es Agentensysteme auf eine plattformunabhängige und graphische Art und Weise darzustellen. Die Modellierungssprache umfasst (i) eine abstrakte Syntax, die das Vokabular der Sprache definiert, (ii) eine konkrete Syntax, die die graphische Darstellung spezifiziert sowie (iii) eine formale Semantik, die dem Vokabular eine präzise Bedeutung gibt. DSML4MAS ist Bestandteil einer (semi-automatischen) Methodologie, die es (i) erlaubt die abstrakte Spezifikation schrittweise bis hin zur konkreten Implementierung zu konkretisieren und (ii) die Interoperabilität zu alternativen Softwareparadigmen wie z.B. Dienstorientierte Architekturen zu gewährleisten.

Abstract

Agent-based computing can be considered as promising approach and powerful technology to develop applications in complex domains by designing and developing applications in terms of autonomous software entities. Associated with the increasing acceptance of agent-based computing as a novel software engineering paradigm, recently a lot of research addresses the development of suitable techniques to support agent-oriented software development.

The state-of-the-art in agent-based software engineering is to (i) design the agent systems basing on an agent-based methodology and (ii) take the resulting design artifact as a base to manually implement the agent system using existing agent-oriented programming languages or general purpose languages like Java. Apart from failures made when manually transform an abstract specification into a concrete implementation, the gap between design and implementation may also result in the divergence of design and implementation, which potentially makes again the abstract design less useful when it comes to maintenance of the agent system. Furthermore, in agent-oriented software engineering, there does not exist any unique methodology that can be applied to any kind of problem without customization, which is one of the main reasons why application developers omit the abstract design.

The framework discussed in this dissertation presents a platform independent domain specific modeling language for MAS called DSML4MAS that allows modeling agent systems in a platform independent and graphical manner. DSML4MAS is specified in accordance to the language-driven development initiative and hence consists of (i) an abstract syntax, providing the vocabulary of the language, (ii) a concrete syntax defining the graphical symbols of the language, and (iii) a semantics giving the vocabulary a clear and precise meaning. To define the abstract syntax of DSML4MAS a platform-independent metamodel for multiagent systems called PIM4AGENTS has been developed that consists of different viewpoints necessary to develop agent applications in an adequate manner. Apart from the language itself, DSML4MAS also allows to automatically (i) check the generated design artifacts against a formal semantic specification to guarantee the well-formedness of the design and (ii) translate the abstract specification into a concrete implementation closing the gap between design and code. Taking both together, DSML4MAS ensures that for any well-formed design, an associated implementation will be generated closing the gap between design and code. DSML4MAS is integrated into a (semi-automatic) model-driven methodology that (i) allows to refine the abstract design of DSML4MAS into more and more concrete and detailed artifacts that are necessary when it comes to the execution of the generated implementation and (ii) supports the interoperability between agents systems and other existing software development approaches, e.g. Service-oriented Architectures. The two main ingredients of the semi-automatic model-driven methodology are (i) the model transformation between the interaction view and the behavior view of PIM4AGENTS. This allows that the abstract communication between agents and groups of agents can be refined into concrete behaviors and (ii) the model transformation between the Service-oriented Modeling Language (SoaML) proposed by the Object Management Group and DSML4MAS.

For evaluation purposes, DSML4MAS has been applied to two industrial use case scenarios from different domains. In the first use case, DSML4MAS has been used to describe the core processes of the steel production at the Saarstahl AG in Völklingen. The second use case deals with the scheduling of ships at the Statoil terminal at Mongstad, where DSML4MAS was used by the application developers to describe the core requirements in an abstract manner and to use the code generators to produce the initial implementation that than needs to be manually refined.

Zusammenfassung

Die agentenbasierte Softwareentwicklung ist ein effektiver Ansatz zur Behandlung komplexer Problemstellungen mittels autonomen und intelligenten Softwareentitäten. Aufgrund wachsender Akzeptanz von Agentensystemen wird der Forschungsschwerpunkt auf dem Gebiet der agentenorientierten Softwareentwicklung vor allem auf die Erforschung von geeignetem Entwicklungswerkzeugen gesetzt.

Stand der Forschung ist es dabei das Agentendesign mittels einer Agentenmethodologie zu spezifizieren und die resultierenden Artefakte als Grundlage zur manuellen Programmierung zu verwenden. Zu diesem Zwecke werden dann spezifische Agentenprogrammiersprachen oder Standardprogrammiersprachen wie Java genutzt. Zwei Hauptprobleme lassen sich bei diesem Ansatz identifizieren: Zum Einem führen die Fehler, die bei der manuellen Überführung von Design in eine konkrete Implementierung entstehen, dazu, dass insbesondere das abstrakte Design weniger nützlich in Hinsicht auf Nachhaltigkeit der entwickelten Softwareapplikation wird. Zum Anderen existiert keine universelle Agentenmethodologie, die ausdrucksstark genug ist, um alle Problemstellungen bei denen Agententechnologie zum Einsatz kommt, adäquat zu beschreiben.

Das in dieser Dissertation diskutierte Rahmenwerk erörtert eine plattformunabhängige domänenspezifische Modellierungssprache für Multiagentensysteme namens DSML4MAS. DSML4MAS erlaubt es Agentensysteme auf eine plattformunabhängige und graphische Art und Weise darzustellen. Die Modellierungssprache basiert auf der sprachgetriebenen Entwicklungsinitiative (engl. Language-Driven Development) und umfasst somit (i) eine abstrakte Syntax, die das Vokabular der Sprache definiert, (ii) eine konkrete Syntax, die die graphische Darstellung spezifiziert sowie (iii) eine formale Semantik, die dem Vokabular eine präzise Bedeutung gibt. Zur Definition der abstrakten Syntax von DSML4MAS wurde dabei ein plattformunabhängiges Metamodell für Multiagentensysteme namens PIM4AGENTS entwickelt, welches dem Entwickler erlaubt das zu beschreibende Multiagentensystem aus verschiedenen Sichtweisen zu formulieren. Neben Syntax erlaubt DSML4MAS zum Einem das erstellte Design gegenüber der spezifizierten Semantik zu überprüfen und zum Anderen die abstrakte Spezifikation basierend auf DSML4MAS automatisch in eine konkrete Implementierung zu überführen und somit die Kluft zwischen Design und Implementierung automatisch zu schliessen. DSML4MAS ist Bestandteil einer (semi-automatischen) Methodolgy welche (i) Teile des abstrakten Designs in konkretere Artifakte automatisiert überführt und (ii) die Interoperabilität zwischen Agentensystemenen und alternativen Softwareparadigmen wie z.B. Dienstorientierte Architekturen fördert.

Neben dem direkten Vergleich zwischen DSML4MAS und den meist zitierten agentenbasierten Entwiklungsmethoden anhand eines vorgeschlagenen Analyserahmenwerks, fand DSML4MAS in zwei industriellen Anwendungsfällen Einsatz. Im ersten Fall wurden mittels DSML4MAS die Kernprozesse bei der Stahlproduktion der Saarstahl AG in Völklingen (Deutschland) dargestellt und in eine ausführbare Implementierung überführt. Der zweite Anwendungsfall befasst sich mit der Problemstellung der effizienten Disposition von Schiffen am StatoilHydro Terminal in Mongstad (Norwegen). Dabei kam DSML4MAS zur Beschreibung der kooperativen Algorithmen und automatisierten Codegenerierung zum Einsatz.

Statement on Publications

This thesis is a coherent presentation of my scientific work since June 2006. Parts of the presented material has been previously submitted, reviewed and published in various conference proceedings, book chapters and journals, and is the result of my collaboration with colleagues and the supervision of a number of master and diploma students.

Part II mainly deals with the language features of DSML4MAS. Early versions of PIM4AGENTS defining the abstract syntax of DSML4MAS has been presented in (Hahn; 2008; Hahn et al.; 2009a, 2007b; Warwas and Hahn; 2009b; Hahn et al.; 2007c). An early version of the formal semantics appeared in (Hahn and Fischer; 2008b,a). Chapter 5 reports on the concrete syntax and graphical editor built upon the results of the master thesis by Stefan Warwas (Warwas; 2007) and (Sadovykh et al.; 2009; Warwas and Hahn; 2009a; Warwas et al.; 2009; Warwas and Hahn; 2008). The DSML4MAS language (summarized in (Warwas and Hahn; 2009a)) won the best academic software award at the eight international conference on autonomous agents and multiagent systems.

Part III mainly deals with model transformations that were partly developed during the supervision of master students Gründel (2009) and Raber (2009). Early versions of the code generators appeared in (Hahn et al.; 2009a, 2007b,d). The internal model transformation between the interaction and behavior viewpoints of PIM4AGENTS has been presented in (Hahn and Zinnikus; 2008; Hahn et al.; 2009b, 2011).

Part IV deals with the evaluation of DSML4MAS. Chapter 9 reports on the industrial use case at the Saarstahl AG, which has been previously presented in (Jacobi et al.; 2009; Hahn et al.; 2010b,c) and (Jacobi et al.; 2010).

Other work published with collaborators not directly reported in this thesis—but conducted in related areas—is cited where appropriate (Hahn et al.; 2008b; Zinnikus et al.; 2008a; Fischer et al.; 2006; Kahl et al.; 2007; Zinnikus et al.; 2007; Leon-Soto et al.; 2009; Hahn and Fischer; 2007; Hahn and Slomic; 2008; Hahn et al.; 2008a, 2006c,b; Fischer et al.; 2009, 2007; Zinnikus et al.; 2008b, 2010; Hahn et al.; 2010a; Elvesæeter et al.; 2010; Nunes et al.; 2011; Hahn et al.; 2010d).

Acknowledgements

First of all, my research would not have been possible without the generous funding of the European Commission in projects like ATHENA (Advanced Technologies for Interoperability of Heterogeneous Enterprise Networks and their Applications, FP6-2002-IST-1), Interop-NoE (Interoperability Research for Networked Enterprises Applications and Software - Network of Excellence, IST-2003-508011) and SHAPE (Semantically-enabled Heterogeneous Service Architecture and Platforms Engineering, ICT-2007-216408), MODEST (Model-driven agents for semantic Web services) funded by the German Ministry for Education and Research (BMB+F), and the German Research Foundation (DFG) of funding the Socionics initiative.

I am indebted to Prof. Dr. Jörg Siekmann for his encouragement and motivation to conduct research in artificial intelligence. Dr. Klaus Fischer provided me with the support I needed during the ups and downs of my thesis project. He gave very constructive and illuminating feedback on drafts of this thesis. His spirit and energy have been an inspiration to me and I am grateful for the working environment he created. I am also grateful to Ingo Zinnikus, Stefan Warwas, Cristián Madrigal-Mora and my other colleagues at the MAS group of DFKI for all the discussions we had. I also thank Prof. Dr. Philipp Slusallek for letting me become a member of his research groups and I am grateful that he accepted to review my thesis. A special thank goes to Prof. Dr. Andreas Zeller accepting to review this thesis.

In 2006, I had the opportunity to visit the ICT research lab at SINTEF in Oslo, present my work, and benefit from many discussions. I am grateful to Dr. Arne J. Berre and Brian Elvesæter supporting me during my stay. On my first day, Dr. Berre introduced me into the field of Language-Driven Development, on which this dissertation is finally based on.

I particularly enjoyed the cooperation with the project partners in the SHAPE and Socionics projects: Dr. Arne Berre, Dr. Michael Stollberg, Brian Elvesæter, Dr. Andrey Sadovykh, Sven Jacobi, Einar Landre, Dina Panfilenko, Dr. Michael Florian, Dr. Frank Hillebrandt, Bettina Fley, and Daniela Spresny.

A special thank also goes to the members of the OMG's standardization groups of SoaML and AML, in particular Mr. James Odell and Dr. Arne Berre. We had a lot of best modeling practice discussions giving me the possibility to further improve DSML4MAS. I am deeply grateful that I had the opportunity to contribute parts of DSML4MAS to both OMG standards.

My supervision of the master students Rolf Schmidt, Stefan Warwas, Susanne Bölker, Torsten Gründel, David Raber was an interesting experience that I do not want to miss. I am very grateful for the many discussions we had and their support regarding implementing transformations part of DSML4MAS.

Last but not least, I thank Billi and Bene for all their support and I look forward to all our future adventures. I am also deeply grateful to my parents whose support made this dissertation possible.

Saarbrücken, January 2013

Christian Hahn

Contents

List of Figures

List of Tables

List of Object-Z Class Schemata

Part I

Introduction, Background, and Problem Statement

1. Introduction

1.1 Motivation

Software engineering is in accordance with (IEEE STD 610.12; 1990) the discipline of Computer Science, which is concerned with the creation process of software systems. Beyond others, two common paradigms for software engineering are the object-oriented and agent-oriented paradigms. Both share many similarities, primarily due to the fact that the agent-oriented paradigm evolved from the object-oriented paradigm. However, especially when designing complex and distributed software systems, agent-based computing can be considered as promising approach and powerful technology by designing and developing software applications in terms of autonomous software entities called agents that are situated in an environment in order to achieve their goals by cooperation in a flexible manner. The coordination between agents can either be achieved by interacting with one another in terms of high-level protocols and interaction languages or/and through plans defining the overall agent's behavior for achieving goals. Compared to the object-oriented paradigm, in accordance with (Jennings; 2001) and (Parunak; 1997) the advantages of agent-based computing are (i) the autonomy of the application components, (ii) the provision of better separation of concerns due to explicitly taking into account situatedness and modeling environmental resources, and (iii) addressing dynamic and high-level interactions (i.e., with societal rather than with architectural concepts).

Associated with the increasing acceptance of agent-based computing as a novel software engineering paradigm, recent research addresses the identification and definition of suitable models, methods and techniques to support the development of agent-based software systems. Agent-Oriented Software Engineering (AOSE) (Ciancarini and Wooldridge; 2001; Wooldridge et al.; 2002; Giunchiglia et al.; 2003; Odell et al.; 2004; Müller and Zambonelli; 2006; Padgham and Zambonelli; 2007; Luck and Padgham; 2008; Luck and Gómez-Sanz; 2009) is an established field—with its first workshop held in 2000—that is concerned with how to engineer agent-based software systems. AOSE as a new engineering paradigm has mainly evolved from Object-oriented Software Engineering (OOSE), where AOSE has placed greater emphasis on the autonomy, interaction, intelligence, and proactiveness of agents, which are the outstanding variations to OOSE.

Even if the AOSE community is very active, agent-based systems and multiagent systems (MASs) technology still face many challenges in being adopted by industry and possibly taking over from object-oriented technology as the dominant software development technology. Especially, the development of industrial-strength applications requires, in accordance to (Odell; 2002), the availability of software engineering methodologies and languages that typically consist of a set of methods, models, guidelines and techniques that facilitate a systematic agent-based software development process, resulting in increased quality of the software product.

- One of the main reasons for not being taken over by industry might be the overly strong emphasis on theory, as implementation and deployment of agent-based systems are still considered secondary compared to its theoretical foundations (Gomez-Sanz et al.; 2008b).

However, the even more essential reason might be the lack of adequate tool support providing methods that automatically guide the application developers from early analysis and design to implementation. Even though many AOSE methodologies have been proposed so far, few are mature or described in sufficient detail to be of real use. None of them is in fact complete (in the sense of covering all necessary activities involved in software engineering) and is able to fully support the industrial needs for agent-based system development (Dam and Winikoff; 2004).

- Another reason might be, in accordance to Kuan et al. (2005), the fact that existing AOSE methodologies are tied to a particular agent architecture from the early phases to the implementation. The resulting design is too constrained to a particular architectural paradigm. Specifying architecture independent analysis models could increase the understandability of domain experts and give the application developers and implementers the choice of selecting an appropriate architecture once the requirements are properly analyzed.
- Apart from the AOSE methodology issues there are also difficulties in implementing MASs (see (Luck et al.; 2006)). These difficulties are due to (i) a lack of specialized debugging tools, (ii) skills needed to move the requirements from analysis and design to code, (iii) problems associated with awareness of the specifics of different agent platforms, and (iv) in understanding the nature of what is a new and distinct approach to systems development. Luck et al. see the lack of mature developing and implementation methods and tools as one reason why developing MASs currently involves higher costs than using conventional (e.g. object-oriented) paradigms. The absence of tool support certainly reduces the change of AOSE in getting adopted by industry.

In the context of OOSE, the Unified Modeling Language (UML[1])—fathered by the *Three Amigos* Grady Booch, James Rumbaugh and Ivar Jacobson—is an international standard in the Software Engineering community declared by the Object Management Group (OMG). UML's main objectives is to provide an universal set of concepts independent of particular programming languages or development processes used within an expressive visual modeling language that is widely accepted by industry in all phases of software development, from design to code generation. Agents are—as previously mentioned—pretty similar to objects and often implemented using object-oriented programming languages, however, three distinguishing features make the difference in accordance to (Tveit; 2001): Firstly, agents offer structures for representing mental components like beliefs and commitments on which decisions are met. Secondly, agents support high-level interactions through either using agent-communication languages based on the speech act theory (Searle; 1969; Cohen and Levesque; 1979) or complex agent interaction protocols as opposed to ad-hoc messages sent between objects. Thirdly, objects are controlled from the outside, as opposed to agents that have some certain degree of autonomous behavior, which cannot be directly controlled from the outside. Even if the differences between objects and agents are not too serious, special agent-based modeling methods are needed as existing modeling languages for OOSE like UML do not provide the necessary vocabulary, notation elements nor methods to model all features of agent-based systems and MASs.

Hence, similar to UML for OOSE, the goal of this dissertation is to define, specify and establish a modeling language tailored to the domain of agents and MASs. We see the main benefit of such a domain-specific[2] modeling language for MASs (DSML4MAS) in its ability to provide abstractions that are tailored to the specific problem domain of agent-based computing. At this, we expect a potential increase in productivity and ease of use. Some other benefits of domain-specific

[1] see the current UML 2.2 specification at http://www.omg.org/technology/documents/formal/uml.htm

[2] Domain-specific software development deals with developing software systems for a specific domain. As one of the core principles on which this thesis base, the basic ideas of domain-specific software development are clarified in Chapter 3

modeling languages are the possibility to raise the level of abstraction and the ability to produce a more precise model, since it is focused in a narrower view of the problem of defining MASs. This leads to a more flexible and agile agent-based software product.

To realize DSML4MAS, we developed a framework for MASs in accordance to the language-driven development principles[3]. With the development of this language, we pursue the following abstract objectives:

- Provide strong concepts that allow to build practical and convincing MAS and agent applications
- Integration with non-MAS systems to ensure that non agent-based practitioner can easily use DSML4MAS in combination with their already existing traditional software engineering approaches
- Model-driven integration and harmonization with mainstream software engineering
- Fast application development advanced by the design notation and validation facilities of DSML4MAS

In order to achieve these promising objectives, we carefully need to explore the general problems we want to solve. This is done by stating the main issues with state-of-the-art AOSE, followed by listing the key research questions related to this dissertation, we consider important to answer and clarify to meet the previously mentioned objectives.

1.2 Problem Statement and Research Questions

The development of complex systems by applying the agent-oriented paradigm requires adequate modeling techniques and methodologies that provide key functionalities to decrease complexity in developing agent-based systems. This is confirmed by the recent AgentLink[4] roadmap (Luck et al.; 2005, p. 85):

> Most new software technologies require supporting tools and methodologies. A fundamental obstacle to the take-up of agent technology is the current lack of mature software development methodologies for agent-based systems.

To resolve the lack of mature agent-based software development methodologies, languages and tools, we consider the following research questions necessary to investigate:

What are the most important and core building blocks of MASs?

The absence of a clearly defined vocabulary for modeling agent-based systems is certainly one major reason not being adopted by industry. In order to build adequate agent-based mechanisms, methodologies and tools, we believe that as a first step, we need to agree on the basic concepts of MASs. The basic definitions of object-oriented notions of objects, like classes, generalization, specialization, and aggregation are widely accepted by practicing professionals. However, at the same time, the multiagent community could not reach any agreement on the core concepts of MASs and the relationships among them. The successful and widespread deployment of complex

[3] In opposite to model-driven development, language-driven development focuses on the definition of an abstract syntax, concrete syntax and precise semantics. A precise definition of language-driven development is given in Chapter 3

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

software systems based on MASs requires the identification of an appropriate set of agent-based concepts that provide the baseline for defining adequate agent-based engineering methods and tools.

While basic agent-oriented methods have commonalities, the MAS community is far from having community-wide consensus on the majority of agent and multiagent concepts (Padgham et al.; 2008). Even if several agent definitions are available and researchers and practitioners use the same concepts to represent similar things, the real problem lies in the relationships of concepts similar used in different definitions. For example, when comparing the vocabularies provided by the three most cited agent-based methodologies ADLEFE (Bernon et al.; 2005a, 2003), Gaia (Wooldridge et al.; 2000; Zambonelli et al.; 2003) and PASSI (Cossentino; 2005), surprisingly, the only concept all of them have in common is the concept of an agent. A first attempt toward the development of a unified vocabulary has been described in (Bernon et al.; 2005b). The corresponding metamodel was developed by merging the metamodels of ADELFE, Gaia and PASSI aiming at combining the strengths of each metamodel. Even if unifying relevant aspects from each metamodel seems to be a promising approach. The resulting merged metamodel might work if restricting MASs to the academic domain. However, it is still far too abstract to design complex industrial scenarios. Moreover, approaches like SODA (Molesini et al.; 2005; Omicini; 2001), MESSAGE (Caire et al.; 2002) or MaSE (O-MaSE) (DeLoach et al.; 2001; DeLoach; 2005) already nicely identify the core blocks of MASs. Nevertheless, like in the case of the unified metamodel, it is pretty unclear how to automatically derive executable code from them, as aspects like the internal behaviors of agents or agent interactions are not covered in detail.

What is an appropriate graphical visualization and notation of a MAS language?

A second issue we see is the lack of meaningful graphical notation providing the domain experts a clear intuition in how to use the underlying vocabulary correctly. The development of MASs or agent-based systems, in general, is more complex and error-prone than conventional object-oriented design. Thus, specific methods need to be developed reducing the overall complexity. Often, for this purpose, a methodology is introduced, guiding the system developer through the different phases and views necessary when designing MASs. In the context of OOSE, scientists firstly agreed on the basic concepts, followed by the tool vendors agreeing on a common notation as an appropriate graphically visualization naturally supports the scalability of the design. Even if many AOSE research tools exist, they are mainly built from scratch and only little effort has been undertaken to integrate them into integrated development environments (IDEs) such as Eclipse. Luck et al. (2006) see the need of integration as main step toward reducing implementation costs.

Agent-based programming language like JADE (Java Agent DEvelopment Framework, (Bellifemine and Rimassa; 2001)) or Jadex (Pokahr et al.; 2005b) typically only rely on textual representation. In contrast, Jack Intelligent Agents (JACK Intelligent Agents; 2005) already provides visual techniques for specifying code, however, only code skeletons are generated that need to be completed by the experienced domain expert who is familiar with the syntax and semantics of the underlying textual programming language. In (Padgham et al.; 2008), a unified notation for AOSE has been proposed that bases on the notation of the methodologies O-Mase (García-Ojeda et al.; 2007), Prometheus (Thangarajah et al.; 2005) and PASSI. This unified notation is a starting point for developing the notation of DSML4MAS.

What is an adequate formal semantics that support testing, validation, and code generation issues?

The third obstacle we see is the lack of semantics in agent-oriented development approaches. Event if a clear vocabulary and notation is provided, the generated artifacts are rarely complete with respect to all requirements needed for full code generation. A formal semantics can increase the domain experts' understanding on how to model correctly in terms of ensuring that all requirements are met to automatic generate code. Even when agent-based code is generated, full testing and validation is usually required, which consumes a significant chunk of development effort. This effort can be decreased if validation and testing facilities base on a formal semantics and techniques such as model checking.

Several contributions exist proposing a formal approach to develop MASs. The most prominent approach is proposed in (d'Inverno and Luck; 2001b), which is based on the specification language Z (Spivey; 1992). However, in that approach, the development of MASs is purely restricted to the formal specification, no graphical visualization nor automatic code generation is offered. The authors of (Brandão et al.; 2004) propose an approach in which Object-Z (Smith; 2000) is extended for specifying MASs. In accordance to them, AgentZ extends Object-Z with new constructs to enhance structuring and to accommodate new agent-oriented entities such as agents, organizations, roles, and environments. The Agent-Z approach seems to be promising, however, the authors only cover the static semantics, operational aspects are not considered. Another approach is specified in (Hilaire et al.; 2000) combining Object-Z and statecharts to define MASs as the authors consider Object-Z too weak for specifying the complex features associated with MASs. However, it is unclear whether existing Object-Z tools (e.g. type checker) can be used for checking, validating, and verifying the generated models. Other formal or at least semi-formal approaches exist, like for instance, the $i*$ framework proposed in (Yu; 1995). A mapping between $i*$ and Z is discussed in (Vilkomir et al.; 2004), however, again neither graphical visualization support nor full automatic code generation is included in the framework.

How to enhance the usability of agent-based tools to ease the design made by domain experts?

A further barrier toward making MASs a mainstream paradigm is the lack of tool support. Even if the research on MASs is a very active area, only little research has been done with respect to the development of adequate tools to support the design of agent-based systems. In particular, an adequate IDE support for developing MASs is rather weak, and existing agent tools do not offer the same level of usability as state-of-the-art object-oriented IDEs (Luck et al.; 2006). Apart from a graphical visualization, this kind of tool support should provide facilities to support the domain experts in testing, evaluating, and executing the designed artifacts. To be of real benefit, this should all happen in an homogeneous environment that (i) can be easily installed and used by the application developers, and (ii) fits into the existing tool environment utilized for traditional software engineering in order to improve the interoperability from a technical perspective between traditional mainstream paradigms and AOSE approaches. Furthermore, as pointed out by (Luck et al.; 2006), the inherent complexity of agent applications also demands a new generation of computer-aided software engineering (CASE) tools to assist application designers in harnessing the large amount of information involved. This requires to reason at appropriate levels of abstraction, automating the design and implementation process as much as possible, and to allow for the calibration of deployed MASs by simulation and run-time verification and control.

Few agent-oriented design methods exist (e.g. MaSE with agentTool (DeLoach; 2001), ROADMAP with REBEL (Juan et al.; 2002), and PASSI with PTK (Cossentino and Potts; 2002)). However, none of them—to the best of our knowledge—provide all the described features nor offer the same level of usability as object-oriented IDEs.

How to close the gap between agent-based design and implementation?

Despite the number of languages, frameworks, development environments, and platforms that have been developed in the AOSE field, implementing MASs can still be considered as a complex task (Luck et al.; 2006). In part, to manage MASs complexity, the research community has produced a number of methodologies that aim at structuring agent development. However, even if practitioners follow such methodologies during the design phase, there are difficulties in the implementation phase, partly due to the lack of maturity in both methodologies and programming tools. The current state of the art in developing MASs is to design the agent systems by applying an AOSE methodology and take the resulting design artifact as a base to manually code the agent system with an agent-oriented programming platform. Agent-based systems can be deployed across a number of different implementation platforms, each has its own requirements and languages. The fifth obstacle we have identified is that the separation between the core functionality and the requirements of the deployment platform is rarely kept clean during the development of the system. The fact that the deployment (i.e. implementation) is developed completely manually from the requirements (i.e. design) may tend to the divergence of design and implementation which makes again the design less useful for further work in maintenance and comprehension of the system (Bordini et al.; 2007a).

The state of the art in mainstream OOSE is to apply the principles of model-driven software engineering to close the gap between design and implementation. In the area of AOSE, this trend has been recognized as several methodologies apply these principles for the same reasons. However, the generated code is still skeleton-like that further needs to be manually completed.

How to improve the interoperability between existing standards of software applications and MASs?

The sixth barrier for not getting accepted by industry is that the agent community has not been sufficiently well integrated the existing agent-based methodologies, frameworks and languages into existing standards of software engineering. For instance, bringing the key ideas from business-oriented languages and MAS together can be considered as one of key research challenges of AOSE.

Service-oriented architectures (SOAs) are often considered as glue that brings agents closer to business. SOAs have emerged as a direct consequence of specific business and technology drivers that have appeared over the past decade. From the business side, major trends such as the outsourcing of non-core operations and the importance of business process re-engineering are driving SOAs as important architectural approach to business information technology today (Weerawarana et al.; 2005). From the SOA side, adequate mechanisms need to be explored to combine business requirements and the underlying execution engines.

The agent paradigm is not the paradigm of choice of business analysts when it comes to designing business requirements. Even if agent-based computing certainly offers advantages like flexibility and adaptability, particular tailored languages (e.g. Business Process Modeling Notation

(BPMN[5], (Object Management Group; 2006))) are normally used for this purpose. Consequently, only few works exist that aim at bridging the gap between business-oriented approaches and MASs. Taveter (2004) presented an agent-based approach for business modeling, where Agent-Object Relationship Modeling Language (AORML, (Wagner; 2003)) is used as underlying agent modeling language. Endert et al. 2007 presented a transformation between BPMN and JIAC IV (Java-based Intelligent Agent Componentware, (Albayrak and Wieczcorek; 1999)) to bridge the gap between business process languages and agent-based systems. However, apart from the fact that only a single platform is involved, the even more problematic issue of the proposed model transformation architecture is that in most cases the gap between business languages like BPMN and agent platforms cannot be automatically bridged. An intermediate level like represented by SOAs is often considered as more helpful as the business requirements can stepwise be refined.

Considering these issues, the objective is to develop a modeling language for MASs aiming at enhancing the state of the art concerning the obstacles identified. The present dissertation offers several contributions that are highlighted in the following.

1.3 Approach and Main Contributions

This thesis advances the state of the art in agent-based computing research in several areas. The overall objective of this dissertation is to provide a domain-specific modeling language for MASs called DSML4MAS that can intuitively be applied by domain experts (but also non-programmers) to design MASs in a platform independent manner. In this respect, platform independent means that DSML4MAS itself is independent of any particular domain application or agent-based programming platform. Beside an integrated development framework supporting the users in building their particular application models, code generation facilities are offered to close the gap between design and implementation (i.e., code generation facilities are offered for different architectural agent programming languages). In particular, the following contributions—closely related to the problem statement and basic research questions debated in Section 1.2—are achieved by this dissertation.

Platform Independent Metamodel for Multiagent Systems

This thesis proposes a basic vocabulary to design MASs in an abstract manner. This vocabulary establishes the abstract syntax of DSML4MAS by applying the principles of metamodelling (cf. Section 2.2.3) to the domain of MASs. Apart from clearly defining the abstract syntax, the metamodelling approach, firstly, supports an easy integration into a model-driven framework and, secondly, establishes the base for defining a formal semantics and graphical editor support for DSML4MAS. The resulting metamodel called platform independent metamodel for MASs (PIM4AGENTS) includes the core building blocks aligned with the underlying concepts of agent-based computing.

[5] The Business Process Management Initiative (BPMI) (http://www.bpmi.org/) developed an initial standard called Business Process Modelling Notation (BPMN) that was adopted by the OMG and renamed to Business Process Model and Notation (BPMN). The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardised bridge for the gap between the business process design and process implementation. Another goal, but no less important, is to ensure that XML languages designed for the execution of business processes, such as BPEL4WS (Business Process Execution Language for Web Services), can be visualized with a business-oriented notation.

The structure of PIM4AGENTS is divided into several viewpoints each focusing on a particular aspects of MASs. This separation allows an elegant way to extend the core of PIM4AGENTS to reflect additional application-specific requirements.

Formal Semantics

This thesis gives the concepts of PIM4AGENTS a formal semantics. In particular, denotational and operational semantics are introduced providing additional information and meaning to support the domain experts in terms of the generation of test cases, their validation, as well as means for the automatically parsing. Moreover, means are offered for (i) syntax checking to ensure that the design conforms to the vocabulary proposed by PIM4AGENTS, (ii) well-formedness checking to ensure that more complex statements deduced from the static semantic hold, (iii) consistency checking to ensure that different viewpoints of PIM4AGENTS are consistent, and finally (iv) model checking to perform dynamic analysis on finite state models to identify whether functionalities like liveness, deadlock-freeness, fairness, or reachability hold for feasible executions of the design. Apart the formal specification, one of the main contributions is the feature that the formal static semantics are integrated into the graphical IDE of DSML4MAS. This enables the domain experts to check the well-formedness of the created design and in case to correct improper parts during the design phase to ensure that the created design can generically be translated into executable code.

Graphical Visualization

This thesis proposes a graphical visualization of the vocabulary provided by PIM4AGENTS. Thus, it presents means to formulate the design artifacts produced by DSML4MAS in a graphical notation. The produced graphical IDE, which is one of the cornerstone of DSML4MAS allows structuring the generated design into separate diagrams to foster the development of highly scalable[6] MASs. Beside the graphical notation, the graphical editor is extended with (i) validation facilities that base on the formalized static semantics and (ii) code generation facilities. The former gives the application developers the opportunity to (i) get a better understanding of the vocabulary's meaning as well as (ii) validate the generated artifacts during design time to ensure that the concepts provided by DSML4MAS have been used correctly. The latter allows transforming the design into code artifacts. To foster the integration of DSML4MAS into existing model-driven frameworks, the DSML4MAS editor is provided as plugin for the Eclipse IDE.

Code Generation and Model-driven Methodology

This thesis proposes translation facilities to automatically close the gap between agent-based design done with DSML4MAS and code based on agent-oriented programming languages (AOPLs). This is achieved through (i) a model transformation on the DSML4MAS level to automatically transfer requirements into the detailed design and (ii) generic code generator mechanisms in accordance to model-driven development. As the one size fits all approach to AOSE is increasingly inappropriate, we developed code generators to different AOPLs to ensure that DSML4MAS—as a platform independent modeling language—can be applied to different requirements of software applications. Based on these requirements, the application developers can freely choose, which agent-based programming language is the most adequate for execution, and freely chooses the

[6] from a design perspective

particular model transformation to finally transform the design into desired code. The static semantics part of the graphical editor ensure that (i) the generated design is rich enough to apply the particular model transformation on it and (ii) the model transformation produces meaningful output on the particular agent-based platform.

Combining Service-Oriented Architectures and Multiagent Systems

This thesis proposes a mechanism to reduce the interoperability barriers between MASs and standard business languages. In particular, inspired by the research projects ATHENA[7] (Advanced Technologies for Interoperability of Heterogeneous Enterprise Networks and their Applications), Interop-NoE[8] (Interoperability Research for Networked Enterprises Applications and Software), and SHAPE[9] (Semantically-enabled Heterogeneous Service Architecture and Platforms Engineering) funded by the European Commission, we explored the opportunity to combine MASs and SOAs in a model-driven manner. This kind of integration is an important step toward making MASs more attractive for industrial usage as SOAs are nowadays the preferred approach to design distributed software systems in real-world scenarios. Following the model-driven development approach to resolve interoperability issues, the integration of SOAs and MASs is performed through a model transformation between the Service-oriented architecture Modeling Language (SoaML)—the new emerging standard for SOAs proposed by the OMG—and DSML4MAS. The resulting PIM4AGENTS model can then, in a second step, be further refined in terms of agent-based concepts and in a third and last step, translated into executable code.

Industrial Case Study

The approach presented in this dissertation has successfully been applied in various industrial application domains (see for instance (Zinnikus et al.; 2008a; Fischer et al.; 2009; Zinnikus et al.; 2007, 2008b)). To demonstrate the usefulness of DSML4MAS in industrial settings, we indicate how to utilize DSML4MAS in the steel and gas and oil industry. In the first industrial use case, we use DSML4MAS for designing the main processes of the supply chain at the steel producer Saarstahl AG. The main objectives are thereby that (i) abstract formulated business requirements can easily be translated into a running system and (ii) existing systems holding strategic information can be re-used within the system to keep the high product quality the Saarstahl AG currently holds. The second industrial use case deals with the scheduling of ships at the Statoil terminal at Mongstad.

1.4 Outline of this Thesis

The structure of this dissertation is depicted in Fig. 1.1 and briefly summarized as follows:

Part I, Introduction, Background, and Problem Statement, discusses the preliminaries of this dissertation by providing sufficient background.

Chapter 2, MD-AOSE: Model-Driven Agent-Oriented Software Engineering, reviews relevant prior literature from the fields of AOSE and model-driven development. In

[7] http://www.athena-ip.org
[8] http://interop-vlab.eu/
[9] http://www.shape-project.eu/

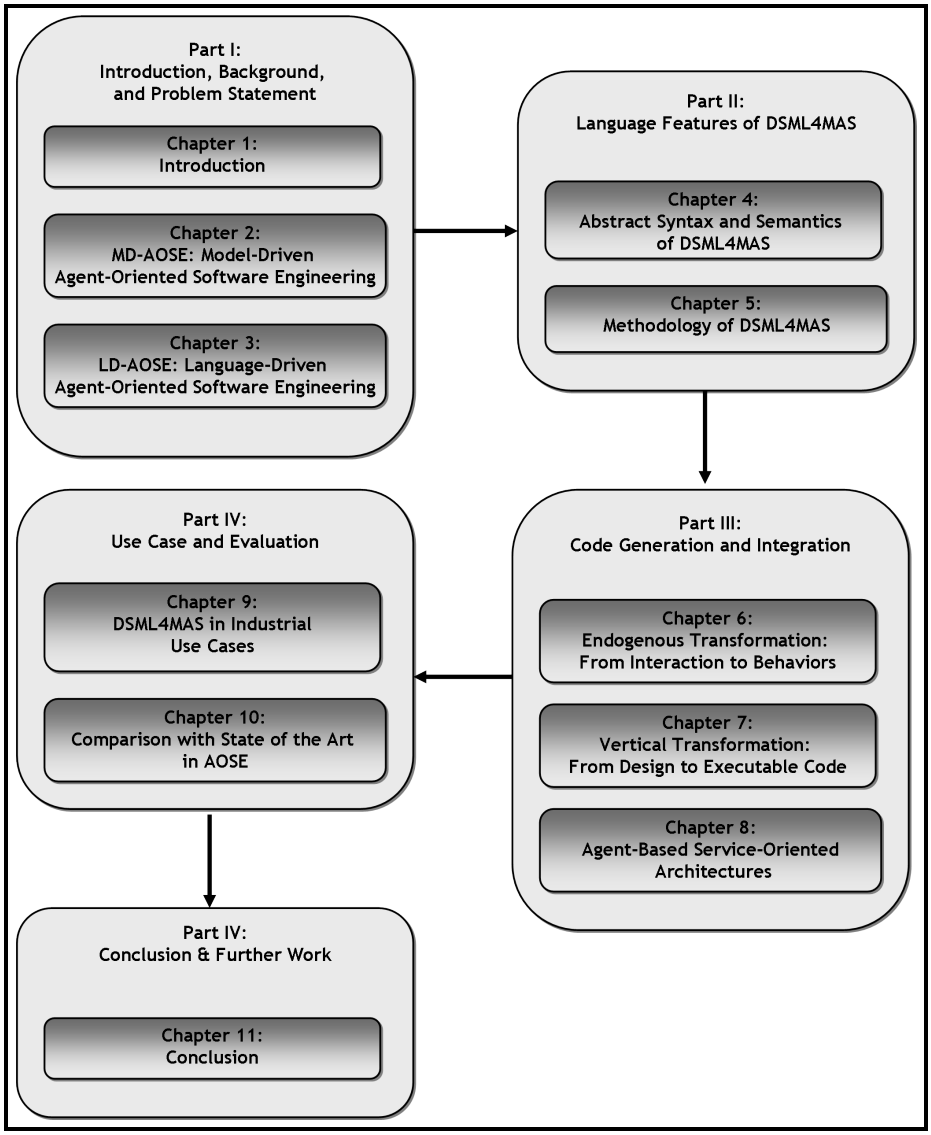


Fig. 1.1: The (graphical) outline of this dissertation.

particular, this chapter discusses the basic building blocks of MASs and gives an introduction into the area of model-driven development, i.e. model-driven architecture, and metamodeling. Furthermore, model transformations and related tool support are debated and evaluated.

Chapter 3, LD-AOSE: Language-Driven Agent-Oriented Software Engineering, similarly reviews relevant prior literature from the fields of language-driven development and domain-specific languages and introduces the architecture DSML4MAS.

Part II, Language Features of DSML4MAS, discusses the language features of DSML4MAS by focusing on its abstract syntax, concrete syntax, and semantics.

Chapter 5, Abstract Syntax and Semantics of DSML4MAS, addresses the vocabulary of DSML4MAS defined by PIM4AGENTS. Moreover, the formal semantics of PIM4AGENTS is stated by taking advantage of the formal specification language Object-Z.

Chapter 6, Methodology of DSML4MAS, addresses the methodology framework of DSML4MAS by focusing on the methodology's notation, tool support, and model-driven process guiding the design.

Part III, Code Generation and Integration, addresses the transformation from modeling to implementation and illustrates how the interoperability gap between agent systems and mainstream approaches like SOAs is bridged by the DSML4MAS framework.

Chapter 7, Horizontal Transformation: From Interaction to Behaviors, demonstrates how to model interaction protocols using DSML4MAS and discusses a model-driven development approach to generically transform these protocol descriptions into process-centric models of DSML4MAS.

Chapter 8, Vertical Transformation: From PIM4AGENTS to Jack Intelligent Agents, demonstrates how to close the gap between design and implementation in DSML4MAS by depicting a model transformation mapping the design made with DSML4MAS into platform-specific code of the Jack Intelligent Agents execution engine.

Chapter 9 Agent-Based Service-Oriented Architectures demonstrates how the interoperability gap between business language and agent-based systems are bridged. For this purpose, we depict a model transformation between SoaML developed by the OMG and DSML4MAS.

Part IV, Use Cases and Evaluation, deals with the evaluation of the presented framework.

Chapter 10, DSML4MAS in Industrial Use Cases, demonstrates how the results of this thesis can be utilized in industrial settings like the supply chain of the Saarstahl AG. For this purpose, we show how non-experts on AOSE can translate the business requirements into MASs conforming to PIM4AGENTS and execute the generated design by applying the code generators to agent-based execution platforms.

Chapter 11, Comparison with State of the Art, illustrates the current state of the art concerning modeling MASs and draws a comparison with the DSML4MAS language.

Part V, Conclusion and Future Work, summarizes the main contributions presented in this dissertation and addresses future research directions.

Chapter 12, Conclusion, gives a summary on the main contributions of this dissertation and describes feasible directions for future work.

2. MD-AOSE: Model-Driven Agent-Oriented Software Engineering

This chapter reviews literature from the areas of AOSE and Distributed Artificial Intelligence (DAI), on the one hand, and model-driven development (MDD) on the other hand. The link between both areas is established through listing relevant work in the domain of AOSE applying principles from MDD.

Structure of this Chapter Section 2.1 gives an overview on the core building blocks (e.g. agent, organization, interaction, etc.) of MASs. Section 2.2 follows by giving a brief overview on the basic principles of MDD and describes how these principles are applied in recent works in the context of AOSE. Section 2.3 concludes this chapter.

2.1 Agent-Oriented Software Engineering

DAI is, in accordance to (Weiss; 1999), the study, construction, and application of multiagent systems (MASs), that is, systems in which several interacting intelligent agents pursue some set of goals or perform some set of tasks. Two lines of research can be distinguished. Firstly, Distributed Problem Solving (DPS) (Durfee; 1999) refers to systems in which a particular complex problem is divided into several smaller sub-problems that are distributed among cooperative agents who interact, plan and work together to achieve the shared goal. Secondly, MASs allow for potentially non-cooperative forms of interaction other than those of DPS systems. In MASs, agents are typically self-interested and do not necessarily share a common goal, but mechanisms like negotiation allow to reach a consensus.

Although every problem can also be solved by a centralized approach, there exist several reasons for using distributed systems. Interactions, for instance, are a necessary ingredient for distributed systems as they pursue the purpose of making problem solving with cooperation easy, to share expertise and knowledge, to work parallel on common and/or distributed problems, to be developed and implemented modularly, and to be fault tolerant through redundancy. Interactions can be distinguished between direct by acting on the environment (i.e. black board systems) and indirect through communication with other agents. The overall aim of distributed problem solving is to reduce the complexity and size of some problems. Three techniques are especially used in this field, i.e. (i) programs are built as modules to reuse them as a black-box, (ii) a technique for handling large problems is the decomposition (i.e. large problems are divided into smaller ones), and (iii) abstraction, which is the process of defining a simpler problem by deleting details of the original problem.

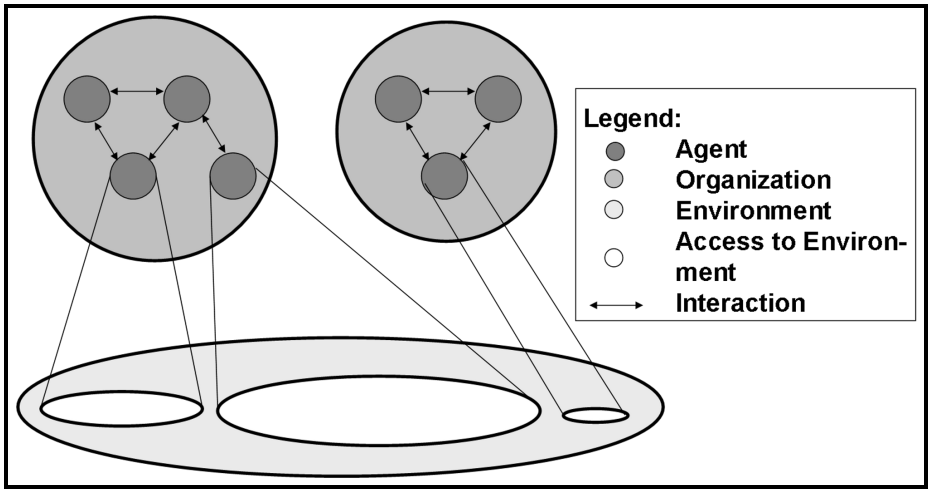


Fig. 2.1: The core MAS building blocks, in accordance to (Jennings; 2001).

2.1.1 Basic Building Blocks of Multiagent Systems

Building high quality software for complex, real world applications turns out to be a very difficult task. So far, a wide range of software engineering paradigms have been proposed in order to reduce the complexity of software. Object-oriented programming (OOP), for example, is such a paradigm. Although approaches like OOP are a step in the right direction, corresponding to Jennings (1999), they fall short in three main ways:

- The basic building blocks—i.e. the objects—are too fine grained
- The interactions between the objects are defined too rigidly
- The available mechanism for dealing with organizational structure is insufficient

Agent-oriented approaches can significantly enhance the ability to build complex (possibly distributed) software systems. Fields of application for software agents are, for instance but not limited to, electronic markets, computer integrated manufacturing, computer supported collaborative work, management, robots, electronic commerce or personal assistance.

One challenge in defining a platform independent language for MASs is to decide, which concepts to include and abstract from the target execution platforms that support the architectural style of agent-based systems. The main building blocks of MASs are depicted in Fig. 2.1. Three abstraction levels are distinguished that are in detail discussed in, for instance, (Fischer et al.; 2005). At the organization level (i.e. macro level), mechanisms to structure agent societies are described. The interaction level (i.e. meso level) mainly focuses on agent communication languages, interaction protocols, and resource allocation. The agent level (i.e. micro level) addresses, for instance, procedures for agent reasoning and learning. The building blocks of MASs are discussed in following in more detail to deepen our understanding and to lay the foundations for further discussions on the core concepts of the proposed DSML4MAS language.

2.1.2 Multiagent Systems

Definition 2.1.1 (MAS, according to Bond and Gasser (1988))

The research in MAS is concerned with coordinating intelligent behavior among a collection of (possibly pre-existing) autonomous intelligent agents and how they can coordinate their knowledge, goals, skills, and plans jointly to take action or to solve problems. The agents of a MAS may be working toward a single global goal, or toward separate individual goals that interact.

Bond and Gasser (1988) emphasize on the coordination of autonomous intelligent agents within a MAS toward solving problems that may either base on a single global goal that is desirable for the whole group or a single individual agent. Beside coordination issues, aspects like environment and interaction are important factors to solve problems efficiently. These are addressed by Weiss (1999) in the following definition.

Definition 2.1.2 (MAS, according to Weiss (1999))

In accordance to Weiss (1999), MASs have the following characteristics:

- *Multiagent environments provide an infrastructure specifying communication and interaction protocols.*
- *Multiagent system environments are typically open and have no centralized designer.*
- *Multiagent system environments contain agents that are autonomous and distributed and may be self-interested or cooperative.*

Weiss (1999) emphasizes on the MAS's infrastructure allowing agents to cooperate for solving problems in a distributed manner. For coordination purposes, hereby, the ability to communicate using pre-defined interaction protocols is one important feature. In OOSE, objects have a centralized organization and are more integrated to each other in a system, while agents are loosely integrated. Bond and Gasser assume in Definition 2.1.1 that agents always cooperate. However, especially in open MAS[1], where agents can freely enter and leave the system, agents do not necessarily cooperate nor should be considered as trustful. In OOSE, the absence of an object will cause an exception error, while MASs would still be stable, if an agent has left the system. Further characteristics of MASs are brought in by Jennings et al. (1998).

Definition 2.1.3 (MAS, according to Jennings et al. 1998)

A MAS is a system that has the following properties:

- *Each agent in a MAS has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint*
- *There is no centralized control*
- *Data is decentralized and computation is asynchronous.*

Jennings et al. focus in their aforementioned definition on single agents, where each of them has only limited capabilities to solve tasks and acts in an autonomous manner, i.e. it has its own control. Each agent has its own beliefs about the environment that are the base for making decisions in the environment.

Summarizing the definitions in this section, we conclude that a MAS consists of a collection of individual agents, each of them displays a certain amount of autonomy with respect to its actions

[1] see (Davidsson; 2001) for a detailed categorization of MASs

and perceptions. Overall computation is achieved by autonomous computation within each agent and by interaction (e.g. communication) among them. The capability of the resulting MAS is an emergent functionality that may surpass the capabilities of each individual cooperating agent.

It is a widely supported assumption in the multiagent community that the development of robust and scalable software systems requires autonomous agents that can complete their objectives, while situated in a dynamic and uncertain environment. These agents need to be able to engage in rich, high-level social interactions, and operate within flexible organizational structures (Jennings; 1999). Organizational structures institutionalize anticipated coordination, which is especially useful for medium- and large-scale applications that require limitation of the agents' communication behavior. Agents acting in such structures can encapsulate the complexity of subsystems and modularize its functionality providing the basis for rapid development and incremental deployment. Even if the definitions given in the previous section differ slightly, each of them naturally comprehends the notion of agent.

2.1.3 Agent

In accordance to Wooldridge (1997), an agent is an encapsulated computer system (e.g. software program, robot) that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objective. Agents are persistent computational entities capable of perceiving and acting upon their environment, in an autonomous manner. They interact and communicate with the environment and other agents and incorporate reasoning techniques (e.g., planning, decision making, and learning) to achieve flexible rational behavior (Wooldridge; 2000a). In the MAS literature, various definitions of the term agent exist (e.g. (Maes; 1995; Hayes-Roth; 1995; Smith et al.; 1994; Shoham; 1997)). We selected two of the most well-known agent definitions for further discussion.

Definition 2.1.4 (Agent, according to Russell and Norvig (1995))

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.

Russell and Norvig presumably gave the most general definition of the term agent. The definition mainly focuses on the interaction of an agent with its environment. Thus, the agent changes the environment through its effectors and reacts to changes through sensors. Even if the interaction with the environment is certainly one important feature of agents, there are further facets that need to be explored. This is done through the most cited definition used in the MAS community.

Definition 2.1.5 (Agent, according to Wooldridge and Jennings (1995b))

Autonomy *Agents are able to act independently, i.e, without the intervention of human beings or other agents and exhibit control over their internal states. Thus, the agent makes independent decisions and actions. Furthermore, Wooldridge and Jennings assume that an autonomous agent has its own beliefs, desires and intentions, which are not subservient of other agents. These beliefs, desires and intentions are necessarily represented within an agent.*

Reactivity *Agents are able to react on changes in the dynamic environment. This is done in order to satisfy the specific goal in time for the reaction to be useful. So, if the agents become aware that their plans have gone awry, they do not ignore this fact and keep on trying to satisfy their plans, but they respond by choosing an alternative course of action.*

Pro-activity *Agents are able to exhibit goal-directed behavior by taking the initiative to generate and attempt possible actions in order to reach the specific goal. Proactiveness rules out of entirely passive agents, who never try to do anything. By trying to achieve goals and intentions, the agents will exploit additional goals which can assist the achievement of the former goal or being contrary.*

Social Ability *Agents are able to interact with other agents or human beings via some agent communication language and to cooperate with other agents in order to achieve the specific goal. The exchange of information via communication is not hard to achieve, since every day millions of computers around the world exchange information with other computers or human beings. The exchange of information is not really social ability. Since other agents are also self-interested and autonomous, the agents have to negotiate and cooperate with these agents in order to achieve their goals.*

Wooldridge and Jennings consider an agent as autonomous (see (Verhagen; 2000) for a detailed discussion on autonomy) entity that is able to communicate within the environment and other agents in social forms like groups. They are, moreover, capable of acting in a reactive and proactive manner. These agent's characteristics are called the *weak notion of agency*. The *strong notion of agency* (Shoham; 1993), further brings up notions, such as knowledge, belief, intention, and obligation.

2.1.4 Agent Architectures

In order to act in a reactive, autonomous, social and pro-active manner, agents are typically based on complex architectures that support the deliberation on the agent's objectives. In this section, we focus on different agent's internal deliberation processes and agent's architectures, respectively.

In accordance to Wooldridge and Jennings (1995a), three different kinds of architectures exist in the agent community. It is interesting that none of these architectures is directly reflected by standards.

Deliberative Architectures Deliberative architectures are based on the physical-symbol system hypothesis (Newell and Simon; 1976), on the foundations of logic and on theorem provers. It is therefore assumed that an agent has a model of its environment expressed through symbols, which can be used to deduce general intelligent actions. Several different approaches exist (e.g. *Planning agents* (Chapman; 1987), *Mentalistic Agents*) concentrating on different AI techniques for specifying the central agent computational entity. Examples for deliberative architectures are GRATE (Jennings et al.; 1992) and Mecca (Bauer and Stiener; 1998).

Reactive Architectures Reactive architectures result from the limitations imposed by symbolic AI. The types of agents do not model complex reasoning, but instead they are constructed in a way that allows them to react to a changing environment by their *instincts*. Hence, reactive architectures are associated with observations of behaviors from the animal world. For instance, an ant colony consists of different, but very simple individuals, but the colony itself exhibit more intelligent behavior than one would expect. The *subsumption architecture* (Brooks; 1991, 1986, 1990) proposes a layered design of competing task accomplishment behaviors. Lower layers exhibit more primitive kinds of behavior and have precedence over layers further up the hierarchy. Examples for reactive architectures are, for instance, Dynamic Action Selection (Maes; 1989)

or SynthECA (White; 2000). Nevertheless, most researchers agree that reactive agents are not well-suited for many kinds of problems.

Hybrid Architectures Hybrid architectures combine the advantages of the above mentioned paradigms with the aim of an integrated effective and efficient agent behavior. Therefore, AI components and reactive elements are subsumed into one design model. A well known example is INTERRAP (Müller and Pischel; 1993; Müller; 1996), which consists of a layered world model and an execution entity. It distinguishes between purely reactive, planned, and social behaviors. Other well-known approaches are RAP (Firby; 1995, 1994, 1989) or AIS (Hayes-Roth; 1995) (for more information we refer to (Wooldridge and Jennings; 1995a)).

For the design of agents with rational and flexible problem solving behavior, the belief-desire-intention (BDI) agent architecture (Rao and Georgeff; 1991, 1995; Georgeff et al.; 1999) has been proven successful during the last decade (Bratman; 1987). Three mental attitudes (beliefs, desires, and intentions) allow an agent to act in and to reason about its environment in an effective manner.

Beliefs are reflections of the current state of the world that can change over time. Often, beliefs are based on sensory information and stand for the information the agent has about the environment it inhabits and its own current state, where these beliefs provide a domain dependent abstraction of entities, by highlighting important properties and omitting unnecessary information (cf. (Braubach et al.; 2005)). Important, beliefs about the world could be incomplete or incorrect, which may result in incorrect interpretation of the state of the world and may lead to incorrect actions.

Goals are an other central concept, following the general idea in accordance to (Pokahr et al.; 2005b) that goals are concrete, momentary desires of an agent. For any goal it has, the agent will perform suitable actions, until it considers the goals as being reached, unreachable, or not wanted anymore.

Intentions stand for desires the agent has committed to achieve. Wooldridge (2000b) annotated that the intuition is that an agent will not, in general, be able to achieve all its desires, even if theses desires are consistent. Ultimately, an agent must therefore fix upon some subset of its desires and commit resources to achieving them. The desires an agent has committed to are called intentions.

Plans represent the behavioral elements of an agent. Plans are in general composed of a head and a body part, where the head specifies the circumstances under which a plan may be selected, e.g. by stating events or goals handled by the plan and preconditions for the execution of the plan. Additionally, the plan head states a context condition that must be true to continue executing the plan. The body provides a predefined course of action, given in a procedural language. When the agent selects a plan for execution, it will execute actions like sending messages, manipulating beliefs, sending messages, executing algorithms, or creating subgoals that may be achieved by other agents through cooperation.

As stated by (Braubach et al.; 2004), viewed from the outside, an agent is a black box that receives and sends messages and acts in its environment. All kinds of events, such as incoming messages or goal events serve as input to the internal reaction and deliberation mechanism, where events are dispatched to plans selected from the plan library. The reaction and deliberation mechanism is the only global component of an agent. All other components are grouped into reusable modules called capabilities (Braubach et al.; 2005).

2.1.5 Organization

Up to now, we mainly focused on agent-centered MASs. However, a lot of research (e.g. (Schillo; 2004; Zambonelli and Parunak; 2002; Parunak and Odell; 2002; Jennings; 2000; Ferber and Gutknecht; 1998)) has be done with respect to organization-centered MAS. Both terms should certainly not be considered as totally separated as suggested by (Ferber et al.; 2004), but rather overlapping as MASs need both, the single autonomous agents, as well as, a society established through organizations or other forms of social groupings. The major advantage of social units like organizations is that those are formed to take advantage of the synergies of its members, resulting in a possibly intelligent entity that enables products and processes that are not possible from any single individual. To deepen our understanding of the term organization in the domain of MASs, we discuss two different definitions and, thereby, illustrate the core aspects of organizations, which are an important concept in PIM4AGENTS.

Definition 2.1.6 (Organization, according to Gasser (1992))

An organization provides a framework for activity and interaction through the definition of roles, behavioral expectations and authority relationships (e. g. control).

The definition given by Gasser is rather general, without especially focusing on the domain of MASs. However, this definition already nicely illustrates the organizational building blocks, like for instance, *role, interaction, behavior*, and *authority*. In the domain of organizations, roles are of major importance as they are, in accordance to (Ferber et al.; 2004), an abstract representation of a functional position of an agent in an organization. The functional position is normally characterized by activities and services required to achieve social objectives. Hence, a role is the abstract representation of a policy, service or function and they typically describe an organizationally-sanctioned structured bundle of activity types (Gasser; 2001).

Yan et al. (2003) distinguish between two perspectives. From the conception perspective, a role is a constraint under which an agent takes part in some interactions and evolves in a certain way. From the implementation perspective, a role is an encapsulation of certain attributes and behaviors of the agent it is bound to. From the society design perspective, roles provide the building blocks of the agent systems, from the agent design perspective, roles specify the expectations of the society with respect to the agent's activity in the society as the agents behave under their bound roles.

An agent may play several roles within an organization, and on the other hand, a role may be played by several agents. In OOSE, an object class usually has a specific capability or functionality, where an agent could play different roles in different domains or situations. Wooldridge et al. focuses on an important aspect in their definition.

Definition 2.1.7 (Organization, according to Wooldridge et al. (2000))

We view an organisation as a collection of roles, that stand in certain relationships to one another, and that take part in systematic institutionalised patterns of interactions with other roles.

Wooldridge et al. consider patterns of interactions as, for instance, defined by agent interaction protocols as core part of agent-based organizations. Thus, the focus inside organizations is mainly shifted from the internal agent architecture as described in Section 2.1.4 toward the communication between roles and the assignment of agents to roles. However, as stated by Ferber et al. in (Ferber et al.; 2004), the latter definition lacks a very important feature of organizations: their partitioning, i.e., the way boundaries are placed between sub-organizations. Organizations

are normally structured into partitions that may (i) overlap and (ii) self be decomposed into sub-partitions again. The partitioning could be done through roles.

According to Pfeffer and Salancik (2003), the key of organizational survival is the ability to acquire and maintain resources. This problem would be simplified if organizations were in complete control of all components necessary for their operation. But, as we can see in all fields where organizations act and perform, no organization is completely self-contained. Organizations are embedded in an environment, which will not be determined by a single organization and consists also of other organizations. The environment regulates the performance of each organization by prescribing norms, constraints and rules. Organizations are linked to environments by federations, associations, customer-supplier relationships, competitive relationships and a social-legal apparatus defining and controlling the nature and limits of these relationships (Pfeffer and Salancik; 2003). One view on organizations, probably the predominate and the most suitable for our purpose, conceives organizations as rational instruments for achieving some goal or set of goals. Goal-oriented organizations imply that the members of the organizations are also goal-oriented, which means that they collaborate in order to achieve something that might not be accomplished otherwise through individual action. This means that the individual members establish the organization, where each individual has an own suggestion about how the organization should look like, according to which rules the organization should behave or which structure the organization should take on. Consensus must be found in order to get rid of these differences, so that each possible member could agree on to build an organization. An organizational framework that covers all these aspects is illustrated in (Schillo; 2004; Hahn; 2004). It discusses several different organizational structures and evaluates, which form is the most adequate for particular scenarios and perturbations.

In accordance to Panzarasa and Jennings (2001), any organization is a MAS in which some form of joint behavior is carried out through differentiation and coordination of tasks among the constituent members. The problem of finding an organizational structure every agent could live with, could be seen as such a differentiation process. However, not every MAS is an organization. Panzarasa and Jennings also distinguish between the organization of a MAS and a MAS as organization. The former refers to the internal structure and coordination of the constituent parts of a MAS, whereas the latter refers to a MAS that relies upon its internal structure to undertake joint behavior.

2.1.6 Interaction

MASs define a powerful distributed computing model, enabling agents to cooperate with each other. Thus, beside aspects like agents and organizational relationships, the interaction between agents, as previously debated in Section 2.1.1, is considered as basic building block of MASs (see Fig. 2.1). Actually, Odell (2002) recognized that autonomy and interactions are the two most critical features of an agent.

In accordance to Malone and Crowston (1994), an interaction can be viewed as a formalization of a concept of dependence between agents, no matter on whom or how they are dependent. Coordination is a special case of interaction in which agents are aware of how they depend on other agents and attempt to adjust their actions appropriately.

However, one of the main questions in the multiagent community is how and why autonomous agents should cooperate with one another. As Castelfranchi (1995) pointed out, cooperation is a paradox, since it involves giving up some autonomy in order to work cooperatively on a shared

goal. Especially in the domains of problem- and task solving, agents need to partially give up autonomy to cooperate mainly carried on interaction.

Definition 2.1.8 (Interaction, according to Malone and Crowston (1994))

An interaction can be viewed as a formalisation of a concept of dependence between agents, no matter on whom or how they are dependent. Coordination is a special case of interaction in which agents are aware how they depend on other agents and attempt to adjust their actions appropriately.

Basically, the interaction between agents is the ability to specify and possibly enforce goals, constraints and desired properties that are not specific to a given agent, but to a group of agents. Interaction improves the possibility of achieving the goal(s) of each single agent, it can also enable the agents to coordinate their actions and behavior resulting in systems that are more coherent. When agents work together in an environment to reach a single complex goal, they do not need to solve any subproblem again and again, but can coordinate their area of responsibility in order to save time and resources.

In accordance to Agostini (2003), coordination is the process of managing dependencies between activities in a given context and interaction is required as a means to coordinate the actions of the individual agents of the MAS. This is necessary to achieve overall system goals and to improve the effectiveness of the system. Coordination can be achieved in different ways: If the agents involved share one common goal, they cooperate by distributed or centralized planning techniques[2]. Otherwise, if the involved agents are self-interested, they need to negotiate for reaching common agreements. Negotiation is defined by Walton and Krabbe (1995) as a form of interaction in which a group of agents, with conflicting interests and a desire to cooperate, try to reach a mutually acceptable agreement. At this, the challenge is to design mechanisms for effectively allocating resources (Rahwan et al.; 2007). The terms communication and interaction are often used in a similar manner. To clarify the differences, in the following, we discuss the definition given by Lind.

Definition 2.1.9 (Interaction, according to Lind (2001))

Interaction is the mutual adaption of the behavior of agents while preserving individual constraints.

Lind considers interaction as any kind of behavior that is related to other agents and is thus more than communication or the exchange of message. Like in the definition given by Malone and Crowston, interaction itself is considered as *mutual adaption* meaning that interacting agents need to coordinate their behavior for the purpose of conversation.

However, undoubtedly, communication as the intentional exchange of information (Russell and Norvig; 1995) is fundamental to achieve agent interaction. Research on agent communication languages (ACLs) (Labrou and Finin; 2000; Labrou et al.; 1999) mainly focus on the design, formalization, implementation, and verification of appropriate communication languages for agents. The most prominent examples are FIPA-ACL (Foundation for Intelligent Physical Agents; 2002) and Knowledge Query and Manipulation Language (KQML) (Finin et al.; 1994a,b) that both base on early works on speech act theory of Searle (e.g. (Searle; 1969)).

Despite the importance of interactions in general, the realization is one main source of difficulties during the development of a MAS. These difficulties arise from the fact that, unlike traditional systems, MASs are usually inherently distributed and asynchronous without any central control

[2] we refer to (Russell and Norvig; 1995) for an introduction on planning techniques

(Braubach and Pokahr; 2007). Research that aims at improving the agent interaction realization can be coarsely divided in protocol-based interactions and flexible interactions.

Flexibility of interactions is achieved by using planning mechanisms to combine predefined protocols, where the planning algorithm takes existing constraints into account and generates a course of communication actions. This leads to more fault-tolerant and hence robust communications, which are driven by the interests of the communication participants. The Hermes methodology (Cheong and Winikoff; 2005b) proposes, for instance, the idea of goal-oriented interaction. In this approach, an agent determines how interaction goals are achieved and plans in which sequence messages need to be exchanged between itself and entities involved. This makes the interaction itself more flexible and less error-prone compared to pre-defined agent interaction protocols. A similar approach to goal interactions is presented in (Rahwan et al.; 2006).

Agent interaction protocols (AIP) (see e.g. (McBurney et al.; 2002; Schillo et al.; 2002; Xueguang and Haigang; 2004)), on the other hand, define communication patterns between several parties as an allowed sequence of messages between agents to form a conversation of a particular type (Cranefiled et al.; 2002). The purpose of AIPs is to determine shared goals and common tasks, to avoid unnecessary conflicts and to make evidence and knowledge available. The importance of AIPs in MASs is underlined by the fact that existing methods for designing MAS like Tropos (Susi et al.; 2005), Prometheus (Padgham et al.; 2007a), Gaia (Cernuzzi and Zambonelli; 2004), or INGENIAS (Pavón and Jorge; 2003) already include mechanisms to express AIPs. In particular, all of them use some sort of Agent UML diagrams (AUML) (Bauer and Odell; 2002) for defining agent-based interactions. However, as discussed in Section 10.2.1, AUML has in its current version some shortcomings when it comes to a broadcast-like message exchange.

2.1.7 Environment

Definition 2.1.2 given by Weiss (1999) already states that the environment plays a major role within MAS. Miles stated: "Just take the universe, subtract from it the subset that represents the organization, and the remainder is the environment" (Miles; 1980, p. 195). In principle, we agree with Miles and his statement. As we do not think that it is that simple, in the following, the term environment is discussed in more detail by stating two more definitions.

Definition 2.1.10 (Environment, according to Panzarasa and Jennings (2001))

Before a MAS comes into existence, environment is the set of resources and phenomena that can determine whether or not the system is generated and what its structure and functioning will be. After a MAS has been generated, environment is the set of resources and phenomena that the system, as a cognitive entity, believes are outside its boundaries, and can affect its structure and functioning.

By referring environmental resources, the authors consider agents, physical object, cognitive properties, techniques, norms (Hahn et al.; 2006a) and institutional values (Hahn et al.; 2007a). In contrast, environmental phenomena are represented by the actions performed by the agents or event posted within the environment. A slightly different point of view has been broached by Weyns et al..

Definition 2.1.11 (Environment, according to Weyns et al. (2007))

The environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources.

Weyns et al. see the environment as first-class abstraction, i.e. the environment is an independent building block (comparable to an agent) of a MAS. Furthermore, they stress that without an environment, it is impossible to build MASs as the environment provides the surrounding conditions (e.g. interaction mechanism, resource access) for agents to exist.

Due to the fact that the environment is one of the basic building blocks of MASs, the type of environment drastically influences the performance and the architecture of the corresponding agents and organizations. The environment restricts the behavior of these by specifying constraints, norms and rules.

The next part of this chapter provides the interested reader with a detailed state-of-the-art summary on model-driven development to set the foundations for the forthcoming context in particular with respect to closing the gap between design and implementation in an automatically manner. In particular, we focus in this section on metamodelling (cf. Section 2.2.3) and model transformations (cf. Section 2.2.4).

2.2 Model-Driven Development

In accordance to (Boydens and Steegmans; 2004), in the beginning of software engineering, programming was mainly done using low-level machine code resulting in machine-centric programs, consisting of series of 0s and 1s. A first improvement was the introduction of assembly languages (1950-1965). A next level of abstraction was necessary in accordance to (Boydens and Steegmans; 2004) to move away from the machine-centric way of computing toward a more application-centric way. At this stage, procedural programming languages (3GLs) like Pascal, Fortran, and Cobol were introduced. Now it was possible to write a program almost independently of the processor. In the beginning of 1980s, the first object-oriented languages (C++, Smalltalk) were presented enabling fully platform-independent solutions. Programs written on specific architectures could now be ported to any other platform if the particular virtual machine is available (cf. (Boydens and Steegmans; 2004)). In the late 1990s, the requirements of a specific program are formulated in a graphical manner applying approaches like UML. Using UML for instance, the application developers were able to identify the core system architecture in a graphical manner and to take the resulting blueprint to manually define the source code that meets these architectural requirements.

However, as the execution environment is in principle built manually, this might result in a gap between the requirement specification often made by business analysts and the actual system implemented by the system engineers due to the system developers' lack of skills or necessary domain knowledge, which possibly leads to incorrect interpretation of the problems and requirements, or incorrect refinement to code. To reduce the chance of incorrect interpretations, recently, the initiative of model-driven development became very popular aiming at automatically transferring the requirements into a runnalbe implementation.

Model-driven development (MDD) (or as a synonym model-driven engineering (MDE)) is emerging as the state of practice for developing modern enterprise applications and software systems. MDD is in accordance to Favre (2004) a subset of system engineering, in which the process heavily relies on the use of models and model engineering. Therefore, software solutions are modeled independently of the underlying programming language, independently of the underlying middleware, and independently of the underlying enterprise architecture. Tools then provide facilities to separate concerns, i.e. making patterns available, generating code compliant to design, modeling in a business-centric way.

In accordance to (D'Souza; 2001), MDD frameworks define a model-driven approach to software development, in which visual modeling languages are used to integrate the huge diversity of technologies used nowadays in the development of software systems. As such, the MDD paradigm provides a better way of addressing and solving interoperability issues compared to earlier non-modeling approaches. MDD focuses on design-time aspects of system engineering, where specifically tailored methodologies describe how to develop and utilize mainly visual models as guideline in the analysis, specification, design, and implementation phases of an Information and Communications Technology (ICT) system. A system development process is model-driven if the following requirements are fulfilled:

- the development of the system is mainly carried out using models at different levels of abstraction and including various viewpoints on the system
- the system development process clearly distinguishes between platform independent[3] and platform specific models
- models play a fundamental role, not only in the initial development phase, but also in maintenance, reuse and further development
- models document the relations between various models, thereby providing a precise foundation for refinement as well as transformation

MDD is considered as software engineering approach that has the potential to greatly improve on current mainstream software development practices. In accordance to Swithinbank et al. (2005) the advantages of an MDD approach are as follows:

Increased productivity MDD reduces the costs of software development by generating code and artifacts from models, which increases developer productivity. Even if transformations need to be defined manually, careful planning will ensure that there is an overall cost reduction.

Maintainability MDD helps to avoid that past solutions components become stranded legacies of previous platform technologies by leading to a maintainable architecture where changes are made rapidly and consistently, enabling more efficient migration of components onto new technologies. A change in the technical architecture of the implementation is made by updating a transformation. The transformation is reapplied to the original models to produce implementation artifacts following the new approach.

Consistency Manually applying coding practices and architectural decisions is an error prone activity. MDD ensures that artifacts are consistently generated.

Adaptability Adaptability is a key requirement for ICT systems. When using an MDD approach, adding or modifying an ICT function is quite straight forward since the investment in automation was already made.

Models as long-term assets In MDD, models are important assets that capture what the ICT systems of an organization do. High-level models are resilient to changes at the state-of-the-art platform level. They change only when business requirements change.

Repeatability The return on investment of MDD from developing the transformations increases each time they are reused. The use of tested transformations increases the predictability of developing new functions and reduces the risk since the architectural and technical issues were already resolved.

[3] A definition of these terms is given in Section 2.2.1

Several approaches exist that apply the principles of MDD. In particular, these approaches are Microsoft's promoted Software Factories (Greenfield et al.; 2004), Model-Integrated Computing (MIC) (Sztipanovits and Karsai; 1997), or OMG's Model Driven Architecture initiative.

2.2.1 Model Driven Architecture

The current state of the art in MDD is much influenced by the ongoing standardization activities around the OMG's Model Driven Architecture (MDA) (Object Management Group; 2003a). MDA is a special MDD approach around a set of standards—e.g. Meta Object Facility (Object Management Group; 2004) MOF[4], UML[5], XML Metadata Interchange (XMI[6]), Common Warehouse Metamodel (CMW[7]), etc.—proposed by OMG. As MDA is based on standards like UML, it is interesting for tool vendors and research groups to implement and develop tool support (Doyle et al.; 2007).

In accordance to (Object Management Group; 2003a), MDA is an approach to system development, which increases the power of models. It is an instance of MDD, as it provides means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification. MDA offers a practical set of standards for delivering higher return on investment in software development. It encapsulates many important ideas—most notably the notion that real benefits can be obtained by using visual modeling languages to integrate the huge diversity of technologies used in the development of software systems. The three primary goals of MDA are portability, interoperability, and reusability. MDA establishes the idea of separating the specification of the system from the details of the way the system is implemented on a software execution platform. Hence, means are provided for (i) specifying a system independently of the software execution platform that supports it, (ii) specifying software execution platforms, (iii) choosing a particular software execution platform for the system, and (iv) transforming the system specification into particular software execution platform(s).

The metamodel[8] of MDA is depicted in Fig. 2.2. A *System* (possibly distributed over different computers) is described by a *Model* is either described as a textual description or visual presentation, or a combination of both. The architecture of the *System* is a specification of its parts and how these interact through clearly defined interfaces. MDA different models for different abstraction levels (cf. 2.2.1.1) and how they are related. A *System* is composed of different *Views* that represent the *System* from the perspective of a chosen *Viewpoint*. In accordance to (Object Management Group; 2003a), a *Viewpoint* on a *System* is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that *System*. Finally, a *Platform* illustrates a certain functionality necessary for the *System* to be provided to meet its requirements.

[4] MOF provides the standard modeling and interchange constructs that are used in MDA. These constructs are a subset of the UML modeling constructs. This common foundation provides the basis for model interchange and interoperability.

[5] UML is the de-facto standard industry language for specifying and designing software systems. UML addresses the modeling of architecture and design aspects of software systems by providing language constructs for describing, software components, objects, data, interfaces, interactions, activities etc.

[6] XMI is a format to represent models in a structured text form. In this way, UML models and MOF metamodels may be interchanged between different modeling tools.

[7] CWM is the OMG data warehouse standard, which covers the full life cycle of designing, building and managing data warehouse applications and supports management of the life cycle.

[8] The term *metamodel* is formalized in Section 2.2.3.

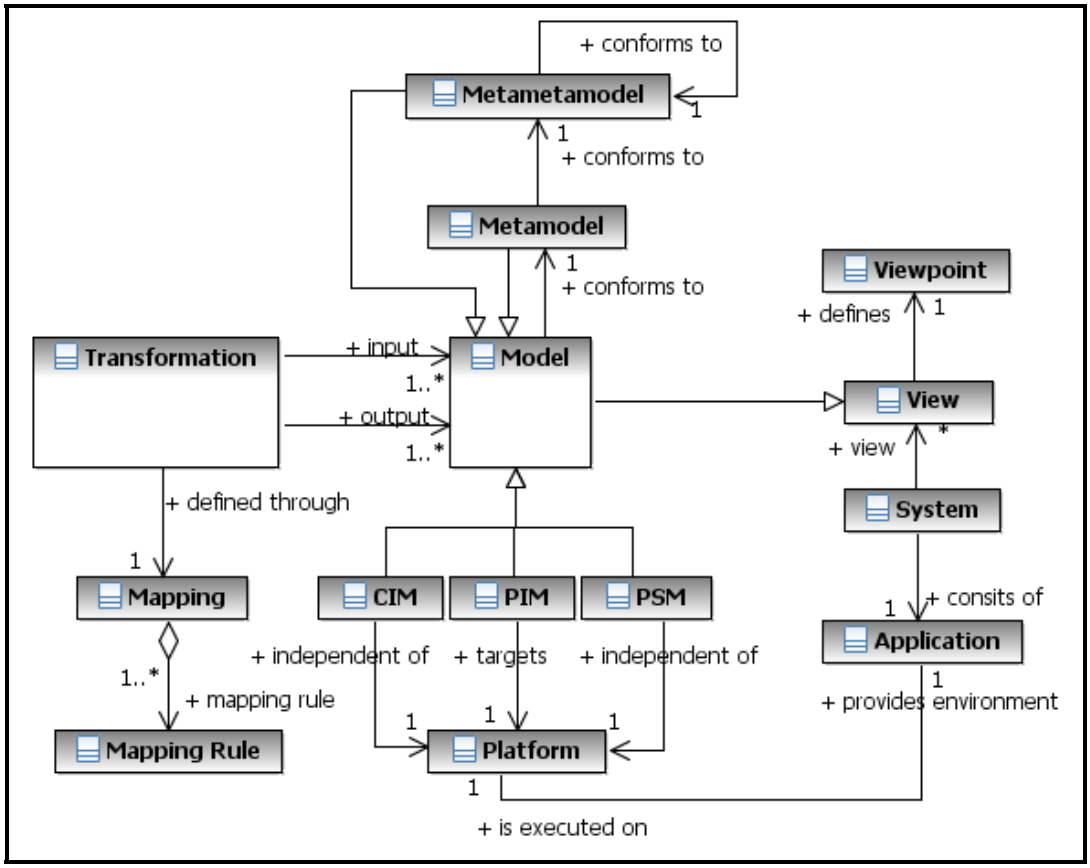


Fig. 2.2: The MDA metamodel.

2.2.1.1 Abstraction Levels

MDA defines three main abstraction levels of a system that supports a business-driven approach to software development. These abstraction levels are important (i) in order to achieve the independence from the software application platform, and (ii) for reasons of longevity in the rapid change of the requirements. Each of them is represented as model (cf. Fig. 2.2) that conforms to a certain metamodel of the particular abstraction level.

Computational Independent Model The computational independent model (CIM) focuses on the (i) abstract environment of the system and (ii) specific (often business) requirements of the system. The CIM represents the computational independent viewpoint and hides the structural and technical details related to the targeted execution platform. A CIM is normally generated by business analysts.

Platform-Independent Model A platform independent model (PIM) is a view of the system from a platform independent viewpoint. This viewpoint focuses on the platform-independent details, hiding platform-specific details. The PIM is suitable for architecture modeling on different platforms of similar types. Hence, it should be able to gather all necessary information needed for describing the system behavior in a platform independent way. In

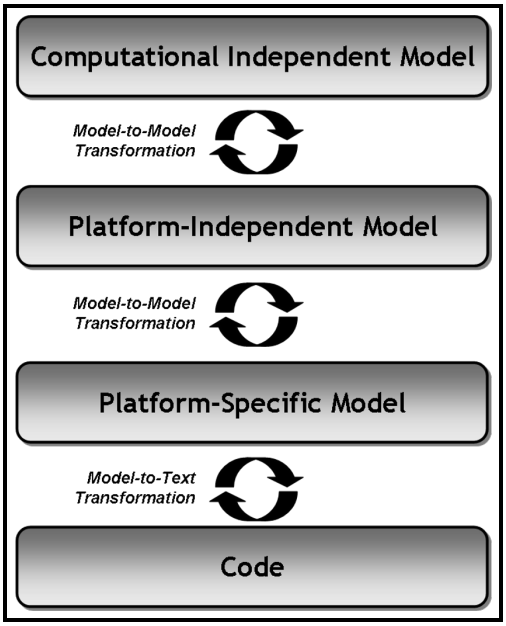


Fig. 2.3: The abstraction levels and their different model transformations.

model transformation architecture, a CIM is transformed to a PIM and refined. The refined PIM then represents the CIM requirements in terms of software services, behaviors and interfaces.

Platform-Specific Model A platform specific model (PSM) is a view of the system from the platform-specific viewpoint. In accordance to (Object Management Group; 2003a), it links the specifications in the PIM with the details that specify how the system uses a particular type of platform. The PSM represents the PIM taking into account the specific platform characteristics. Thus, the PIM is further refined to a set of PSMs each of them describes the realization of the software systems with respect to the chosen platform-specific software technology platforms.

Implementation/Code The implementation level illustrates the textual-written representations that can be executed on the chosen platform.

Even if these abstraction levels are clearly separated with respect to the level of platform-specific details, it is important to note that the abstraction level is relative in the sense that for a particular platform the model could be considered as PSM, but for another platform the model is rather a PIM. Brown (2004) summarized this circumstance as follows: "one person's PIM is another person's PSM".

2.2.1.2 MDA in AOSE

The AOSE community is highly interested in closing the gap between the (early) requirement and implementation phases of AOSE methodologies. Therefore, the principles of MDA are considered as valuable. Amor et al. (2004) presented the Malaca Agent Model (Amor et al.; 2004), where the

Malaca UML[9] Profile provides the stereotypes and constraints necessary to create Malaca models on UML modeling tools. The model transformation is realized from a TROPOS design model—as PIM—to a Malaca Model—as PSM. Guessoum (2005) a MDA-based approach for MASs is proposed by separating the application logic (described in a PIM) from the underlying technology (described in a PSM). Basing on Meta-DIMA, a MDA-based MAS development process defines the PIMs and PSMs by analyzing the multiagent applications, defining a library of metamodels, and designing the transformation rules to implement a metamodel from its description. A first step has been done by defining a PSM for the multiagent tool DIMA[10] and PIMs from the PASSI and Aalaadin/PASSI (Bernon et al.; 2005b) metamodels. In (Xiao and Greer; 2007), an agent-based MDA approach is presented that allows transferring business models into agent-based systems. The Agent Systems Engineering Methodology (ASEME) also applies principles in accordance to MDA by distinguishing between CIM, PIM and PSM models. However, the corresponding models are generated by the different process phases (i.e. requirements analysis, design and implementation phases) and are hence not artifacts from different languages. The developers of the CAFnE toolkit (Jayatilleke et al.; 2006) presented an MDA approach to extend the Prometheus methodology (Padgham and Winikoff; 2002b) in terms of automatically producing an executable implementation. Some of the just presented MDA approaches and their pros and cons are discussed and evaluated in Chapter 10.

2.2.2 Software Factories

Microsoft's Software Factories initiative (Greenfield et al.; 2004) shares many ideas with the OMG's MDA proposal. Among others, the idea of using models and model transformations plays a fundamental role. However, in contrast to using UML as standard modeling tool, Software Factories promote the usage of Domain-Specific Languages (DSLs[11]). At this, in accordance to (Greenfield et al.; 2004), the overall aim of a Software Factory is to develop a software product line that configures extensible tools, processes, and content using a software factory template based on a schema to automate the development and maintenance of variants of a product by adapting, assembling, and configuring framework-based components.

2.2.2.1 Software Factories in AOSE

Recently, several approaches (e.g. (Pe et al.; 2007; Nunes et al.; 2009)) investigated the integration of software product lines and MASs. The overall aim of this research direction is to combine software product lines and AOSE by incorporating their respective benefits and helping the industrial exploitation of agent technology.

[9] A UML Profile is in accordance to (Object Management Group; 1999) a predefined set of Stereotypes, TaggedValues, Constraints, and notation icons that collectively specialize and tailor UML for a specific domain or process (e.g. Unified Process Profile). A Profile does not extend UML by adding any new basic concepts. Instead, it provides conventions for applying and specializing standard UML to a particular environment or domain.

[10] DIMA (Guessoum and Briot; 1999) is a development and implementation platform developed in Java, where agents are considered as a set of dedicated modules able to percept and communicate.

[11] The term Domain-Specific Language is intensively discussed in Chapter 3.

2.2.3 Metamodelling

Models are one of the key artifacts of MDD. The interpretation of this term strongly depends on the domain they are used. For the domain of software engineering, we consider the definition given by Bézivin and Gerbé as a good starting point for our discussion.

Definition 2.2.1 (Model, Bézivin and Gerbé, 2001)

A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.

The key characteristic of a model is its ability to simplify a system as it abstracts from reality and thus can easier be understood by human beings. The second and last definition, we are discussing in this context already perfectly fits in our context of MDD.

Definition 2.2.2 (Model, Kleppe et al., 2003)

A model is a description of a (part of) systems written in a well-defined language. A well-defined language is a language with a well-defined form (syntax), and meaning (semantics). Both are required for the automated interpretation by a computer.

In accordance to Kleppe et al., a model bases on a well-defined language that provides (i) a syntax defining how the models have to look like and (ii) some sort of semantics defining the meaning of the corresponding models. Both, syntax and semantics, allow to interpret models by computers. However, the level of semantics that can be expressed in a model is limited.

In the context of MDD, *metamodels* are used to describe the syntax of a model. In order to understand what a metamodel is, it is useful to understand the difference between a metamodel and a model. Although a metamodel is also a model, in accordance to (Clark et al.; 2004b), a metamodel has two main distinguishing characteristics.

- A metamodel captures the essential features and properties of the language that is being modeled. Thus, a metamodel should be capable to describe a language's abstract syntax.
- A metamodel must be part of a metamodel architecture. This means that a metamodel is again a model that conforms to another metamodel, which itself is described by another metamodel. This allows all metamodels to be described by a single metamodel. This single metamodel, sometimes known as a meta-metamodel, is the key of metamodeling as it enables all modeling languages to be described in a unified way.

The real benefit of metamodeling is its ability to describe languages and models in a unified way. This means that they can be uniformly managed and manipulated thus tackling the problem of language diversity. For instance, mappings can be constructed between any number of languages, as long as they are described in the same metamodeling language (i.e. on the same metamodeling hierarchy). Another benefit is the ability to define rich languages that abstract from implementation specific technologies and focus on the problem domain. Using metamodels, many different abstractions can be defined and combined to create new languages that are specifically tailored for a particular application domain. How metamodels can be used to define the syntax of a language is also addressed in detail in Chapter 3.

- The *meta-metamodeling* layer (M3) provides the infrastructure for a metamodeling architecture. M3 defines the language for specifying metamodels, i.e. level for defining the definition of modeling elements. The elements of the M3 model categorize M2 elements (e.g. meta-class, meta-association and meta-attribute).

- The *metamodeling* layer (M2) describes an instance of a meta-metamodel. M2 defines the language for specifying models, i.e. level of modeling element definition. The elements specified on the M2 layer categorize M1 elements (e.g. classes, attributes, operations, associations, generalizations etc.)
- The *model* layer (M1) describes an instance of a metamodel. M1 defines a language to describe an information domain. The elements specified on the M1 layer categorize instances at the layer M0. Each element of M0 is an instance of the M1 element.
- The *instances* layer (M0) describes an instance of a model and thus the level of the running system. M0 defines a specific information domain.

MOF as part of M3 plays an important role in OMG's metamodel hierarchy as it provides concepts for the existing standards like the Software Process Engineering Metamodel (SPEM, (Object Management Group; 2007)), Common Warehouse Metamodel (CWM, (Poole and Mellor; 2001)), Ontology Definition Metamodel (ODM, (Object Management Group; 2005)), etc. Hence, it is the common foundation that provides the standard modeling and interchange constructs for defining metamodels. MOF can be divided into Essential MOF (EMOF) and Complete MOF (CMOF), where EMOF, on the one hand, is a compromise that favors simplicity of implementation over expressiveness, while CMOF, on the other hand, is more expressive, but also more complicated. The Eclipse Modeling Framework (EMF) started out as an implementation of MOF (more or less aligned on OMG's EMOF), but evolved from there based on the experience gained from implementing a large set of tools using it. EMF can be thought of as a highly efficient Java implementation of a core subset of MOF. Beside EMF, the Metadata Repository (MDR[12]), which is part of the NetBeans tools platform, provides another implementation of MOF.

2.2.3.1 Metamodeling in AOSE

A first attempt of the AOSE community toward the development of a unified metamodel was described in (Bernon et al.; 2005b). This metamodel was developed by merging the metamodels of ADELFE, PASSI, Gaia and PASSI and thus combines the strengths of each metamodel. For instance, the unified metamodel covers aspects like (i) cooperative behavior as described by the ADELFE metamodel, (ii) organizational behavior as specified by the Gaia metamodel and, (iii) FIPA-compliant communication structures as defined by the PASSI metamodel. A more recent approach toward a unified metamodel was discussed during an AOSE Technical Forum Group meeting in Ljubljana[13].

Other metamodeling approaches in AOSE are for instance MESSAGE (Caire et al.; 2002), which defines a methodology to specify telecom applications using agent technology or RICA (Role/Interaction/Communicative Action) (Serrano and Ossowski; 2004) aiming at integrating aspects of ACLs and organizational models on three different layers: On the first layer, generic concepts of the system (e.g. agent, role and action types) are specified, the second layer includes social aspects like norms and institutions. The last layer specifies agent interactions via communication. Moreover, the metamodel presented in (Beydoun et al.; 2005) proposes having a basic, but complete (w.r.t. the concepts that define MASs) metamodel, allowing the generation of systems in different agent platforms. Other metamodel approaches are, for instance, SODA (Molesini et al.; 2005) or MEnSA (Dalpiaz et al.; 2008). Pavón et al. (2006), for instance, presented an update to

[12] http://mdr.netbeans.org/

[13] The attendees agreed on a smaller core part compared to the first draft. In this metamodel, the agent participates in a communication and plays a role that has the ability to solve particular Tasks. Organizations also refer to roles. The cognitive agent is a specialization of agent as it is represented in an environment.

INGENIAS introducing the INGENIAS Development Kit (IDK), as a way to provide MDD tools for MAS development. It presents the IDK MAS Model Editor, a graphical tool for MAS model creation, and a modular approach adapts the editor and tools to new metamodels or target platforms.

Shortcomings All these metamodels cover important agent-related concepts. However, the only concept that shows up in each metamodel of ADELFE, Gaia and PASSI is the concept agent. Although agent-related aspects like interaction and behavior belong to most metamodels, different concepts are used to express their configuration. Metamodels like RICA or the unified metamodel neither include concepts for modeling the core building blocks of MASs nor are precise enough to generate an executable implementation from the design. Hence, in most cases, the automatic and generic transformation from abstract concepts to concrete code is not possible.

2.2.4 Model Transformation

For resolving interoperability issues between models and to bridge the various abstraction levels, model transformations certainly play an important role in the MDD approach. According to Kleppe et al. (2003), a model transformation is the automatic generation of a target model from a source model. Thus, a model transformation can be considered as mapping of a set of models onto another set of models or onto themselves. A mapping thereby defines correspondences between elements or concepts in the source and target models. This is done through so-called mapping rules that are descriptions of how one or more constructs in the source language are transformed into one or more constructs in the target language.

The automatic generation of a target model from a source model can either be performed *horizontally* or *vertically*, where horizontally means to define a mapping and synchronization of models at the same level of abstraction, PIM to PIM or PSM to PSM transformations and vertically means to generate lower-level models from higher-level models, i.e. PIM to PSM transformations. Even if horizontal and vertical transformations are distinct, a complete transformation chain normally combines both, as in a first step, one or more horizontally and vertically transformation steps may precede the final code generation that optionally is merged with manually written code. One reason why extending the generated code with manually written code might be necessary is that no transformation is perfect nor complete and one may want to change things manually. A second reason could be that the target model is more expressive compared to the source model. So the application developer needs to add details that could not be expressed in the source model.

Beside the distinction between horizontal and vertical transformations, Mens et al. (2005) also distinguish between *endogenous* and *exogenous* transformations. Endogenous transformations are transformations between models expressed in the same metamodel. Endogenous transformations are defined between different metamodels. Further characteristics of model transformations (e.g. *level of automation*, *complexity* and *preservation*) are discussed in (Mens et al.; 2005).

In accordance to (van Deursen et al.; 2007), techniques to transform models can be categorized into three different types: Transformations from model to code are used to implement modeling languages. Transformations from code to models are used to extract models from (legacy) code. Transformations from models to models are used to refactor models, to migrate models to a new modeling language, or to map higher-level models to lower-level models. For all three types of transformations, rules and application strategies are written in a special language, called the transformation specification language, which can be either graphical or textual (Czarnecki and Helsen; 2003).

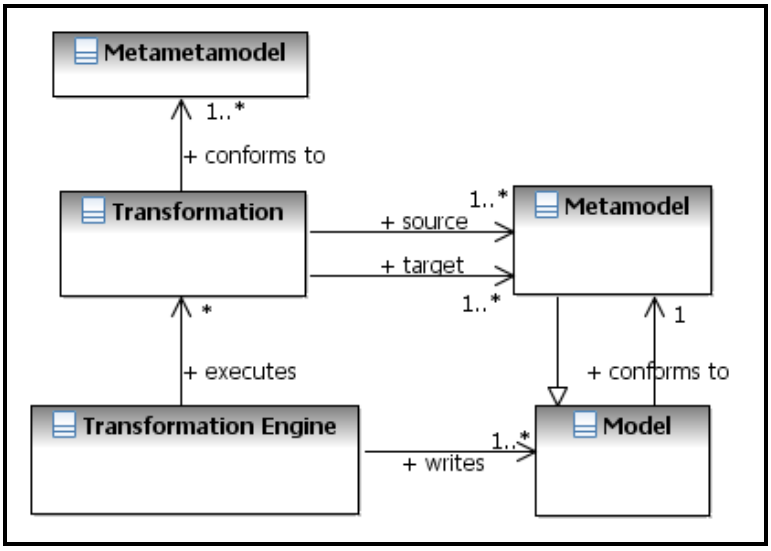


Fig. 2.4: The transformation metamodel.

Fig. 2.4 depicts the metamodel for model transformations. The *Transformation* refers to a non-empty set of source and target *Metamodels*. In the case of endogenous, the source metamodel and the target metamodel are the same, which could be considered as a kind of specialization. Both—*Transformation* and *Metamodel*—conform to the same *Metametamodel*. The *Transformation Engine* executes the *Transformation* by reading the input source *Models* and writing the target *Models* in accordance to the rules defined by the *Transformation*.

2.2.4.1 Model-to-Model Transformation

Model-to-model transformations are the key enabler for model transformation. Therefore, several tools or model transformation engines for implementing model-to-model transformations were proposed. Examples are Kermeta[14] (Falleri et al.; 2006), Atlas Transformation Language[15] (ATL, (Jouault et al.; 2008)), Bidirectional Object-oriented Transformation Language[16] (BOTL, (Marschall and Braun; 2003)), Yet Another Transformation Language[17] (YATL, (Patrascoiu; 2004)), AndroMDA[18] (Mizuta and Huang; 2005), MetaEdit+ (Kelly et al.; 1996), Tefkat[19] (Lawley and Steel; 2005), etc. Recently, the OMG proposed the MOF 2.0 Query/View/Transformation Specification (QVT, (Object Management Group; 2008a)) as standard for model transformations, which allows implementing transformations in a declarative and imperative manner. To distinguish between different model-to-model transformation approaches, Czarnecki and Helsen (2006) distinguish several categories of model-to-model transformation languages:

- *Direct manipulation approach:* This category of model transformation languages is the most simplest form and offers the user only limited support in implementing transformations.

[14] http://www.kermeta.org/
[15] http://www.eclipse.org/m2m/atl/
[16] http://sourceforge.net/projects/botl/
[17] http://is.tm.tue.nl/staff/rdijkman/yatl4mdr.html
[18] http://team.andromda.org/
[19] http://tefkat.sourceforge.net/

Often, the user needs to implement the mappings in a programming language like for instance Java.

- *Structure-driven approach:* The languages of this category consists of two phases. In the first phase, the structure of the output models are defined, whereas in the second phase, the content of the source model is extracted and put on the particular place at the target model. Example: Model Transformation Framework[20] (MTF)
- *Operational approach:* This category of languages are similar to *direct manipulation approaches*. The main feature these kinds of languages offers to the user is the additional operational support like, for instance, a query language like OCL (Object Constraint Language[21]). Example: Kermeta
- *Template-based approach:* This language category uses a template defined by the model transformation architect that is used to instantiate the generated format. Example: AndroMDA[22]
- *Relational approach:* This category of languages allows defining relations between the elements of two or more metamodels. Relationships are normally defined bi-directional, which allows transforming models into different directions enabling roundtrip engineering. Example: QVT[23]
- *Graph-transformation-based approach:* This category of languages operates on typed, attributed, labeled graphs. A study on graph transformations is given in (Ehrig et al.; 2005). Example: Graph Rewrite And Transformation[24] (GreAT, (Gray et al.; 2006))
- *Hybrid approach:* This category includes languages that combine techniques from other categories. Examples: ATL
- *Other approaches:* Any language category that does not fit into one of the previously presented categories. Example: Extensible Stylesheet Language Transformation[25] (XSLT)

2.2.4.2 Model-to-Text Transformation

Most of the available model-to-text transformation tools can be categorized as either template-based or visitor-based (Czarnecki and Helsen; 2003). The template-based approaches (e.g. MOFScript[26] (Oldevik; 2006; Oldevik et al.; 2005) AndroMDA[27] (Mizuta and Huang; 2005), etc.) replace particular parts of the target document with the information extracted from the source model. The visitor-based approach, in contrast, visits each concept in the source model and writes predefined text to a text stream accordingly. The most prominent visitor-based tool is Jamda[28]. An other approach is to use XSLT for implementing the model-to-text transformations. Using XMI, the source models can be serialized as Extensible Markup Language (XML) and imported in the particular platform.

[20] http://www128.ibm.com/developerworks/views
[21] http://www.omg.org/docs/ptc/03-10-14.pdf
[22] http://www.andromda.org/index.php
[23] http://en.wikipedia.org/wiki/QVT
[24] http://www.escherinstitute.org/Plone/tools/suites/mic/great
[25] http://www.w3.org/TR/xslt
[26] http://www.eclipse.org/gmt/mofscript
[27] http://andromda.org/
[28] http://sourceforge.net/projects/jamda

2.2.4.3 Model Transformations in AOSE

Bertolini et al. (2006) demonstrated how automatic transformations can be used to convert UML models on the various phases of the Tropos methodology (Bresciani et al.; 2004) to finally maintain a related implementation. The model transformation is defined by using the Tefkat model transformation engine. Kardas et al. (2009b) presented a model transformation to combine MASs and semantic Web services using the ATL language. As part of the implementation phase of ASEME, Spanoudakis and Moraitis explore the opportunity to define a model transformation between the AMOLA language (Spanoudakis and Moraitis; 2008a) and JADE. For this purpose, Spanoudakis and Moraitis explored the adequacy of the model transformation languages ATL and QVT. Moraitis and Spanoudakis (2006) demonstrated how systems designed following the GAIA methodology and its corresponding models can be converted to JADE for deployment. Analogously, the Gaia2Jade process proposes that the implementation phase should be performed in four stages: communication protocol definition, activities refinement, JADE behavior creation, and agent classes construction. However, the Gaia2Jade process is not provided as model transformation that automatically translates Gaia models to JADE. Instead, the transformation needs to be done manually. In (Rougemaille et al.; 2008), the authors illustrated a practical use of model transformations and code generation part of the ADELFE v.2 methodology. In (Spanoudakis and Moraitis; 2008b), a model transformation from capabilities to interactions is presented. Molesini et al. (2008) illustrate how to use model transformations to integrate an agent-based methodology and infrastructure to obtain a new software process. Another interesting approach is described in (Kardas et al.; 2009a), which introduces platform-specific modeling and code generation tools for the model-driven development of MASs.

Shortcomings of State of the Art Even if some of the presented approaches utilize model transformation engines like Tefkat, most of them are only defined on a conceptual level. Moreover, due to the fact that the underlying source metamodels are often too abstract, the model transformations only provide skeleton code, if the model transformation is generic at all. Chapter 10 evaluates the model transformations of existing MAS design approaches in terms of how well (i.e. automatic, manual) code generation is supported.

2.3 Bottom Line

This chapter introduces the interested reader into the areas of MAS and AOSE, on the one hand, and model-driven development on the other hand. Both areas are important when developing a basic vocabulary for MASs in a platform-independent manner.

In the first part of this chapter, the core building blocks of MASs are detailed, including the concepts of agent, organization, interaction, and environment. An agent is thereby considered as autonomous entity in the system that is able to act in a reactive and pro-active manner. Through coordination and interaction with other agents, complex problems are solved that go beyond the capacities of a single agent. Organizations are typically used in order to form social groups that establish complex problem solving entities. The careful definition of these terms lay the foundation to precisely establish a unified vocabulary for describing MASs that will form the core of the platform-independent language to be formalized in the forthcoming chapters.

The second part gives an overview of MDD in general and Model-driven Architecture, metamodeling, and model transformations in particular. Both, the principles of metamodeling and the

MDA idea to distinguish between platform-independent and platform-specific models are considered in the language development of DSML4MAS. Thereby, model transformations determine means for combining the newly created language with already existing platforms on the different abstract levels.

3. LD-AOSE: Language-Driven Agent-Oriented Software Engineering

The state of the art in designing MASs is to design agent-based systems based on existing AOSE methodologies and take the resulting design artifacts as base to manually implement the MASs using existing agent-oriented programming languages (AOPLs) or general purpose programming language like Java. The resulting gap between abstract design and concrete implementation may tend to the divergence of both, which makes again the design less useful for further work in maintenance and comprehension of the system (Bordini et al.; 2007a). A resent trend in AOSE is therefore to apply mechanisms from MDD (cf. Section 2.2) to close the gap between agent design and resulting implementation. However, MDD neither provides capabilities to rapidly design new languages and tools in a unified and interoperable manner as needed would be needed for adequate agent-oriented software development nor offers full potential of, for instance, the language-driven development approach, which is addressed by this chapter to propose the general research approach of this thesis in a precise and falsifiable manner.

Structure of this Chapter Section 3.1 introduces the term language-driven development and presents the main benefits of this software engineering approach. Section 3.2 gives an overview of DSML4MAS by briefly illustrating its main components and architecture. Section 3.3, finally, concludes this chapter by briefly summarizing the presented approach to lay the foundations for the future content of this thesis.

3.1 Language-Driven Development

A significant factor behind the difficulty of developing complex software is the wide conceptual gap between the problem and the implementation domains (France and Rumpe; 2007). The growing complexity of software is the motivation behind work on industrializing software development. In particular, current research in the area of MDD is primarily concerned with reducing the gap between problem and software implementation domains through the use of technologies that support the systematic mapping between problem-level abstractions and concrete software implementations.

MDD aims to be a major step forward in the way that systems will be developed in the future as it aims at capturing the enterprise modeling architecture defined on CIM level on the selected target platform. France and Rumpe (2007), however, identifies two major challenges MDD has to face:

- The abstraction challenge: How can one provide support for creating and manipulating problem-level abstractions as first-class modeling elements in a language?

- The formality challenge: What aspects of a modeling language's semantics need to be formalized in order to formal support the manipulation, and how should the aspects be formalized?

Although model transformations and in particular PIM to PSM transformations have an important role to play in the overall system development, there is an increasing interest in the wider application of MDD to what is termed by Clark et al. (2004b) as language-driven development (LDD).

The core idea of LDD is to focus on the development of languages and tools that are tailored to the special needs of the application developers to improve the system's development practice. LDD aims to raise the productivity of the software development process by concentrating on powerful language abstractions and development environments that support the engineering of languages and processes. It is fundamentally based on the ability to rapidly design languages and their supporting tools, based on the principles of metamodeling. Hence, in accordance to (Clark et al.; 2004a), it involves the application of MDD technologies to rapidly generate and integrate semantically rich languages and tools that address specific domain modeling requirements. The aim is to provide developers with rich modeling abstractions appropriate to their development needs and thus enabling them to clearly focus on the problem domain in isolation from implementation details (Clark et al.; 2004a).

Languages are an essential part of the development of systems. In this development process, languages are either used as a high-level modeling language that abstracts from implementation or tailored to a specific implementation platform. Beside general-purpose languages, like for instance UML or Java, providing suitable abstractions applicable across several domains, there exist domains that are too complex to describe with general purpose languages. In these situations, more domain-specific languages (DSL) (Cook et al.; 2007) are needed providing a highly specialized set of concepts clearly formalized to target a small problem domain. Apart from using languages to design and implement systems, they typically support many different capabilities (e.g. execution, validation, testing) that are an essential part of the development process.

LDD and MDD should not be considered as complementary approaches. Rather, LDD is a specialization that completes the MDD philosophy, by enabling all aspects of the development process to be captured in languages expressed by for instance (meta) models and a clear semantics. According to LDD, the language provides a formal framework that allows to build agile abstractions that can be changed and adapted if necessary. This in combination with the separation of PIM and PSM made by MDA will also have beneficial effect for the whole development language.

3.1.1 What is a Language?

Independent of the purpose, any language definition comprises three main parts, i.e. the *abstract syntax*, *semantics*, and *concrete syntax*. The characteristics of them are detailed in the remainder of this section. Their relationship within a language is depicted in Fig. 3.1.

3.1.1.1 Abstract Syntax

The abstract syntax defines rules that prescribe how the language constructs (e.g. words) are combined to more complex constructs (e.g. sentences). Such an abstract syntax is normally called grammar (e.g. Backus-Naur form).

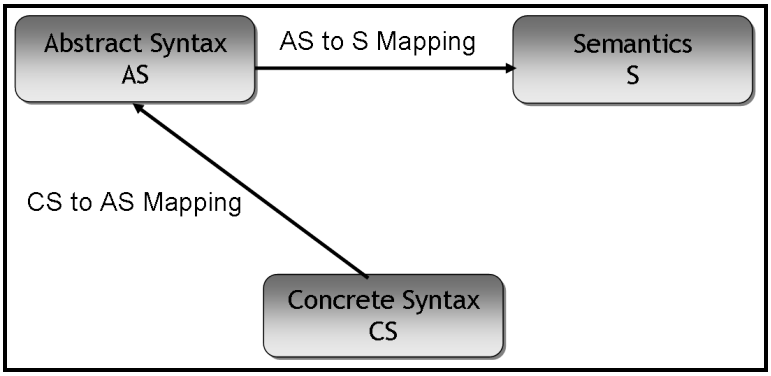


Fig. 3.1: The three core components of a language.

Similarly, in the context of LDD, the abstract syntax of a language describes the vocabulary of concepts provided by the language and how they are combined to create models or programs (Evans; 2006). Apart from addressing the set of provided concepts and their relationships, the abstract syntax may also include constraints expressed by, for instance, the Object Constraint Language (OCL) that prescribe under which circumstances a model is well-formed. These sort of rules are often referred to static semantics. The static semantics are especially necessary if additional tool support (e.g. model transformations) is provided that expects the correct structure and usage of the language in order to provide meaningful results. To ensure the models correctness, the models must be validated beforehand with respect to the given constraints.

In terms of MDD, the abstract syntax is described by a metamodel or UML profile that defines how the models should look like. MOF, for instance, defines the structure of UML by specifying concepts for classes, attributes and references. Apart from addressing the structure of a language, the abstract syntax serves as the basis for defining the (dynamic) semantics. This is illustrated in Fig. 3.1 through the *AS to S* mapping.

3.1.1.2 Concrete syntax

The concrete syntax is the set of notations that facilitate the presentation and construction of the language constructs (Evans; 2006). The concrete syntax could either be formulated in a textual or visual manner. A textual syntax allows describing the models in a structured textual form, whereas a visual syntax allows using a diagrammatical form. In the latter case, the notation is visualized through graphical symbols. UML, for instance, uses nodes and edges to represent some underlying model elements, while other languages may have different notations. One of the main benefits of the concrete syntax is that the complexity of the abstract syntax can be hidden behind the manner in which the graphical symbols are arranged.

3.1.1.3 Semantics

The semantics of the language define the meaning and the purpose of its elements. The semantics definition is a very important feature of a language as it clarifies what the actual elements represent and leaves no place for assumptions. Semantics are a key element in order to understand how to correctly use a language. Visual modeling languages such as UML offer only little support

to express the language's semantics. The semantics can, as previously mentioned, be further distinguished into static and dynamic semantics. The static semantics thereby specifies the structural meaning, whereas the dynamic semantics are concerned with the behavior of the language, thus the language's operational aspect.

In terms of MDD, even if the developers may have an understanding of the syntax of a language, the semantics are the key to clarify the language's and in particular the concept's meanings. In the context of MDD, semantics are often introduced when transforming a PIM to a specific (execution) platform that already offers some kind of (operational) execution semantics.

3.1.2 Benefits of Language-Driven Development

As previously mentioned, LDD and MDD should not be considered as complementary approaches, rather LDD should be considered as MDD approach. There are several benefits of LDD compared to traditional MDD (e.g. MDA) that make LDD an interesting approach also to investigate in the area of AOSE.

- Languages are precise and have a clear formal semantics. The formal semantics increase the application developer's understanding of the language and supports the execution, evaluation and testing on the generated models.
- Languages provide an adequate graphical notation that reduces modeling complexity and increases the application developer's understanding of the language's meaning.
- Languages can be combined and integrated in multiple ways through model transformations in accordance to MDD.
- Transformations between languages can be validated, as the behavior of models or programs written in one language can be checked against the behavior of the language it is translated to.

Apart from these benefits, there are certainly also potential disadvantages that need to be further discussed. First to mention is the complexity of the language design, the implementation of code generators, the precise formalization as well as defining a suitable concrete syntax. The higher costs for maintenance and education for the language users need to additionally be mentioned.

3.1.3 Domain-Specific (Programming) Languages

In accordance to Watt (1990), programming languages should fulfill several criteria like they (i) must be universal (every problem must have a solution that can be programmed in the language, if that problem can be solved at all by computer), (ii) must be implementable on a computer and (iii) should also be reasonably natural for solving problems, at least problems within its intended domain. However, further distinguishing features can be proposed to group programming languages like for instance declarative vs. imperative or functional vs. object-oriented.

The most interesting criterion for this thesis is to distinguish between general purpose and domain-specific. General purpose languages like UML, as the most prominent representative nowadays for designing visual software systems, were developed to solve a wide variety of problems from scientific computing up to business processing. It is important to recognize that UML as general purpose language is intended to be the one and only language, a universal standard by the OMG for object-oriented software development. There are, however, domains where engineers either do not understand UML, or the general concepts of UML are simply inappropriate for modeling effectively in the certain domain, possibly due to the fact that they might already have

their own standard languages or tools. UML as general-purpose modeling language is also rather limited in integrating domain-specific concepts (cf. Section 3.1.4.1).

Recent articles, such as (Cook; 2004), see domain-specific languages as "...the next step towards developing a technology for software manufacturing". Others, like (Iseger; 2005), claim that a pure DSL-based approach consistently results in productivity increases of 500-1000%, compared to the mere 35% found with MDA. To informally set the boundaries of the term *domain-specific language*, we present two defintions that nicely frams the core ideas.

Definition 3.1.1 (Domain-Specific Language, van Deursen et al., 2000)

A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Every DSL has, in accordance to the above definition given by van Deursen et al., a well-defined problem domain, which is the specific area of application in which the systems created with the DSL will be used. There are several interesting challenges involved with identifying and delineating the problem domain.

Definition 3.1.2 (Domain-Specific Language, Cook et al., 2007)

A Domain-Specific Language is a custom language that targets a small problem domain, which it describes and validates in terms native to the domain.

As aforementioned, DSLs are explicitly tailored to a particular target domain, not to cover all feasible problem domains. This idea is not novel, however, in former times these kinds of languages, such as HTML etc., were called special-purpose languages or little languages (Bentley; 1984). Like any language, a DSL could either be defined using a textual or graphical notation. For several reasons, a graphical notation has significant advantages over a textual notation. Most important, a graphical visualization using some sort of diagram can easily be interpreted by human beings in contrast to a hard readable textual visualization.

The general framework of a DSL is depicted in Fig. 3.2. Two roles have to be distinguished: The DSL developer defines an adequate level of abstraction of the domain, specifies the language constructs, the notation used as well as code generators. The DSL can then be used by the application developers by installing the DSL-specific IDE. The application developer then defines the application models by using the provided notation and applies the code generation facilities to produce code that can be afterwards manually refined. Together with the domain-specific framework, a working application is produced.

3.1.4 Domain-Specific Modeling Language

Analogously to domain-specific programming language, a Domain-Specific Modeling Language (DSML) is a visual modeling language designed for a specific purpose inside a certain problem domain. A DSML is a new approach to model-based software development. Like for DSLs in general, a domain in DSML is defined as the set of concepts and their relations within a specialized problem field. Necessary information that describes actual business processes and entities, the manner in which these entities interact, etc. constitutes domain knowledge that can only be obtained from domain experts developing the DSML.

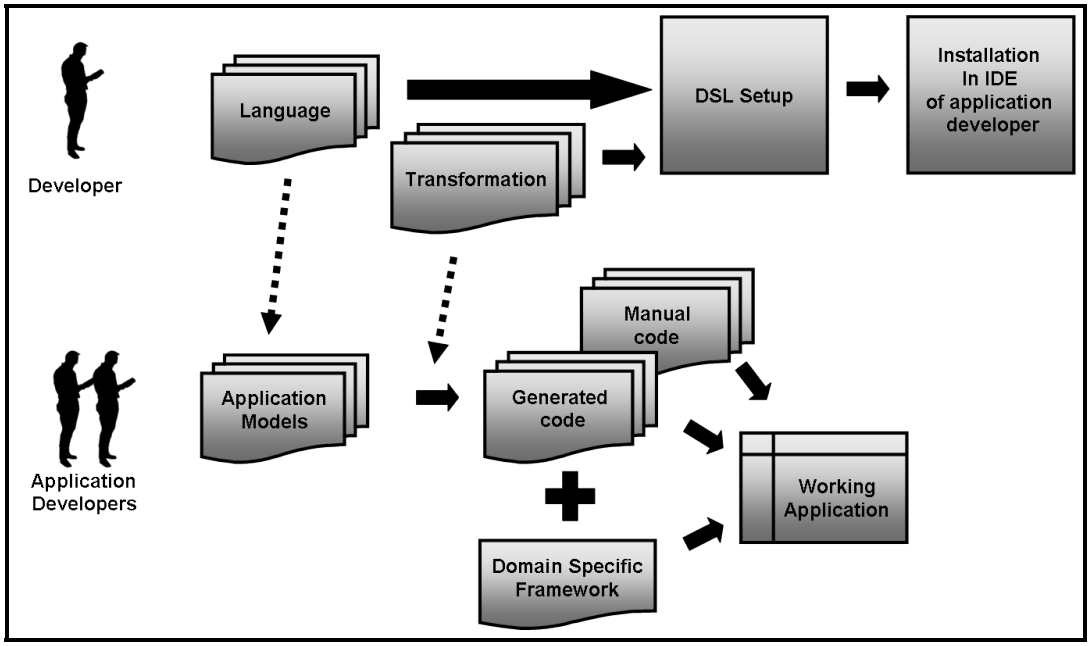


Fig. 3.2: An overview of a general DSL framework

A DSML gives the designer the freedom to use concepts and underlying logic that is specific to the target application domain. Hence, the DSML is completely independent of programming language concepts and syntax, which is a more flexible solution compared to MDA emphasizing the importance of a single and universal modeling language (i.e. UML) at the center of the development process. Moreover, these domain models (with appropriate visualization) are more easily readable and usable for the domain experts compared to UML diagrams for instance. This is certainly one of the main advantages of DSMLs, because developers usually have limited knowledge about the application domain, which can result in inaccurate working programs.

A DSML can be realized in two ways, either by customization of pre-existing languages through profiles or by creating a new language with a standardized meta-data architecture based on, like for instance, MOF. The first approach through customization is achieved by marking up UML concepts with existing stereotypes and tags to create a new domain concept. The second approach is based on the idea to create a brand new DSML from scratch. This involves applying metamodeling facilities and standards to create a metamodel of the DSML, which is used to generate adequate tool support for existing platforms.

3.1.4.1 UML is *not* the Solution

One option is certainly to define a general-purpose modeling language and teach the domain experts how to use it in different domains. Experience with using UML tells us that this is not often successful. The development of the UML started during the early 1990s. It emerged as a unification of the diagramming approaches for object-oriented systems developed by Grady

Booch, James Rumbaugh, and Ivar Jacobson. First standardized in 1997, it has been through a number of revisions, most recently the development of version 2 in 2005 and 2.2[1] in 2008.

UML can either be used as a means of creating informal documentation for the structure of an object-oriented design (i.e. as a sketch) or as first-class artifacts within a (model-driven) software development process. In the latter case, either manually or automatically, code skeletons are produced by systematically translating a UML model into source code by utilizing existing model transformations. Even if UML is considered as standard modeling language, several limitations can be identified, especially for designing a DSML:

- UML bases on a very large metamodel, which makes the whole it hard to use due to complexity. Most of UML usages only rely on a small subset of the entire metamodel such as UML Class diagrams.
- UML specification falls short in providing a precise and formal semantics. In contrast to its syntax, which is defined in an unambiguous manner, the semantics of its elements is either missing or specified in natural language. Hence UML's semantics tend to be not precise enough for the purpose of formal system verification nor for the correct usage by the application developers.
- UML concepts are often not suitable for modeling domain-specific applications as the concepts of the particular domain are either not existing or need to be defined through profiles.
- UML is not executable, which makes it impossible to transform a UML model using code generators to produce executable artifacts. Especially in the case of UML profile, the code generators need to be manually defined by the developers of the DSML.

UML started out as general purpose object-oriented modeling language. But as a consequence of its popularity, attempts were made to tailor for more and more highly specialized uses for which it was not originally intended. For instance, the recently emerging standards of SOA or agents under the umbrella of OMG are good examples, where UML profiles are used to express SOA and MAS-related concepts using basic UML vocabulary. Initially, when submitting the first proposal of the Agent Metamodel and Profile (Object Management Group; 2009a), we started with four different ways of modeling the core principles of agent-based systems through UML profiles. At the same time, the concepts are expressed in a single metamodel. This already demonstrates that finding the right UML concepts for a certain domain also in terms of semantics is sometimes a very difficult task.

The main advantage when applying UML profiles is that any existing tool supporting UML profiles can be used to represent the concrete syntax. This means that the usual UML notation is applied, and stereotyped by the particular domain-specific concept. However, the new domain-specific language is restricted to the concept provided by UML, its meaning (i.e. semantics) cannot be changed to better fit the new language's needs.

3.1.4.2 Metamodeling is *not* the Solution

Even if the principles of MDD and in particular model transformations are important towards closing the gap between design and implementation, several limitations can be identified. Especially when it comes to defining a DSML through metamodeling:

[1] Whenever we are talking of UML, we talk about version 2 or above

	Metamodel	UML
Adding new types	✓	×
Adding new attributes	✓	×
Adding new associations	✓	×
Adding new methods	✓	×
Concrete syntax	×	✓
Abstract syntax	✓	×
Semantics	Only static	Hard to refine
Tool support	×	✓

Tab. 3.1: UML vs. metamodel for defining DSMLs

- MOF as a meta-metamodeling language is not rich enough to capture semantic concepts in a platform independent way. Especially the operational semantics cannot be specified with MOF and more formal techniques have to be used instead.
- MOF itself does not provide means for expressing the notation of a language, in particular in a diagrammatical manner, which is necessary when defining a graphical modeling language.
- MOF itself does not provide abstractions for capturing tools in a generic fashion. This means that either the language designer has little control over the tool that supports the language, or these aspects must be encoded in a platform specific way.

Even if a formal semantics cannot be defined within a metamodel, at least the static semantics can be specified using OCL. However, in contrast to the development of DSMLs conforming to UML, where the language developer lean the domain concepts on UML concepts, the developer is free to define the domain concepts when using the principles of metamodeling. Consequently, attributes, types as well as associations can newly be established. However, to determine the concrete syntax, additional tool support is necessary that allows mapping concrete syntax to abstract syntax. Section 3.2.3 introduces tools supporting the specification of this mapping.

Table 3.1 sums up the pros and cons of metamodeling and UML profiles for the purpose of producing a DSML. Important to mention is that UML profiles only provide little support to change the semantics of UML for covering the special needs of the new developed language. In contrast—and this is certainly one of the major criteria for using profiles—tool support is provided by any existing UML tool as well as the concrete syntax can easily be specified. However as in the case of metamodeling, when it comes to code generation, often the model transformations have to be defined by hand. The major advantages of metamodeling is obviously that the language wanted to develop can easily be tailored to the special needs of its domain. Consequently, the abstract syntax can newly be created by adding new types, associations and methods. However, the tool support automatically offered, as well as, means for defining the concrete syntax is limited. Even if recent initiatives like the Graphical Modeling Framework (GMF) were started to resolve this lack.

Both approaches demonstrate their shortcomings for defining DSML, however, metamodeling in combination with adequate tool support like GMF to define the concrete syntax and tool support might be the better alternative. The real strong criterion is that a UML profile does not allow any changes on the existing UML core, which includes adding new classes or associations. However, especially when customizing UML for supporting agent-based systems, this would be

necessary as agent-based systems certainly add certain functionalities that cannot be expressed with object-oriented approaches. To add or refine the semantics of existing UML concepts is also rather difficult, which again favors the metamodeling approach.

3.2 Domain-Specific Modeling Language for Multiagent Systems

As mentioned earlier, the main objective of the Domain-Specific Modeling Language for Multiagent Systems (DSML4MAS) is to produce a graphical modeling language to support the non-agent expert in using MASs. Fig. 3.3 depicts the general idea of the DSML4MAS design.

The language developers (i.e. we) define the DSML4MAS language consisting of the three basic ingredients abstract syntax, concrete syntax and semantics. Based on the abstract syntax and semantics, we define model transformations to the two agent platform JACK and JADE. Together with a graphical modeling tool, we provide a setup that can easily be installed by the MAS developers. They create the models in accordance to DSML4MAS and apply the model transformations to generate code, which can, if necessary, be manually refined. Together with the respective agent platform, the MAS developers can run agent application.

Although different kinds of languages can be distinguished (i.e. modeling languages like UML or programming languages like Java), there are some common features that any language share and that are necessary to consider for the development of new languages. These core features of a language are, in accordance to (Langlois et al.; 2007), the *language*, *transformation*, *tool*, and an optional *process*. All of these ingredients are detailed in terms to DSML4MAS in the remainder of this dissertation, a first brief overview is given in the following.

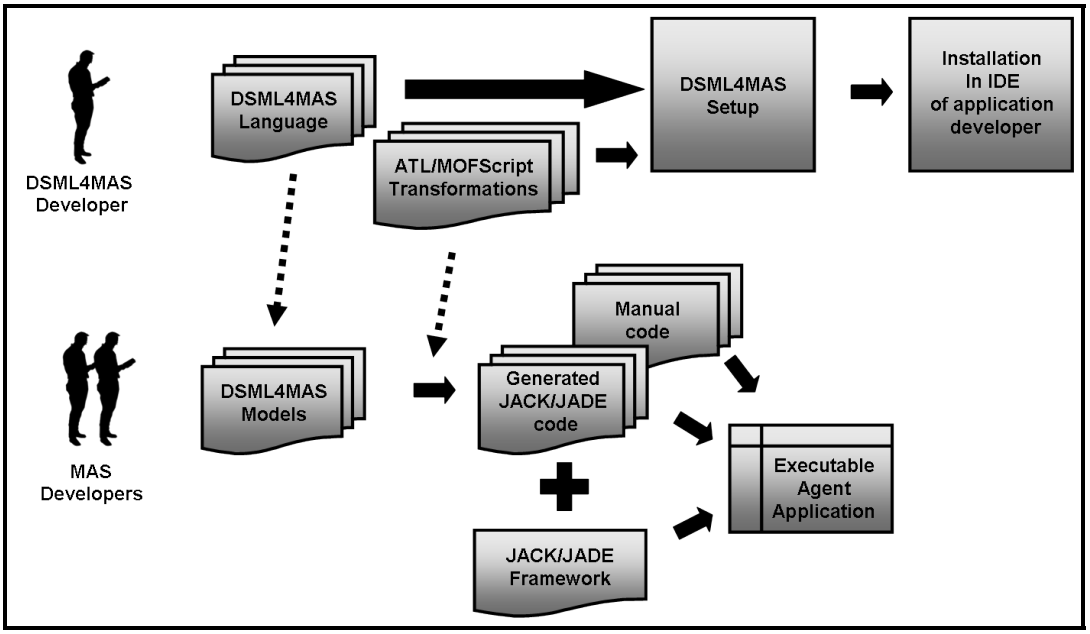


Fig. 3.3: An overview of our framework.

3.2.1 Language

In Section 3.1.4, we investigated two feasible options for defining a DSML in general and DSML4MAS in particular. The first option is to define a UML profile that specifies how to model MASs using the concepts provided by UML, the second option is to define a MOF-compliant metamodel that directly defines the domain-related concepts.

As previously stated, we consider the second approach more appropriate for defining DSML4MAS. The development of DSML4MAS is divided into four phases. In the first phase, an adequate level of abstraction must be found. This implies to formally define a language model that specifies the abstract syntax of the language. In terms of DSML4MAS, the language model is derived from the PIM4AGENTS metamodel. In the second phase, a suitable concrete syntax is defined, e.g. graphical symbols, which is used by users of the language. As the concrete syntax represents the concepts in the abstract syntax, usually there is a correspondence between concrete and abstract syntax elements. In the third and last phase, a generator is defined in accordance to MDD that translates the modeling language into an executable representation. For this purpose, the elements of the abstract syntax have to be mapped to instances of the abstract syntax, conforming to the formal language model of the agent-based language.

3.2.1.1 Abstract Syntax of DSML4MAS

The abstract syntax of a modeling language in general expresses the structure of its vocabulary (i.e. concepts) and the relations between them, which is supported by different metamodeling environments like Ecore (EMF), MetaGME (GME) (Davis; 2003), or XCore (XMF Mosaic) (Clark et al.; 2004b). Ideally, one unique meta-metamodel is defined as this enables reuse and integration with other tools or architectures. Following this, the architecture of DSML4MAS is built around the Ecore meta-metamodel. Ecore has been chosen, because it has proven its practical usability, as a wide range of tools and languages are based on Ecore. Hence, to describe the abstract syntax of DSML4MAS, we created a metamodel called PIM4AGENTS, which conforms to Ecore as meta-metamodel. Consequently, the concepts of PIM4AGENTS are instances of the EClass metaclass, their characteristics are described as EAttribute and, finally, their relationships are defined as EReference.

3.2.1.2 Semantics of DSML4MAS

An abstract syntax contains only little information about what the concepts in a language actually mean. Ecore, for instance, is not sufficiently precise to set out all relevant aspects of a specification. Beyond straightforward constraints (e.g. association multiplicities), there exist a range of complex and sometimes subtle restrictions that are not easily conveyed in graphical form. For instance, for specifying the dynamic semantics of a system in UML, natural language is often used.

PIM4AGENTS already includes parts of the static semantics, however, additional information is needed in order to capture the complete (also dynamic) semantics of a language, which is important in order to give the language a clear representation and meaning. Otherwise, assumptions may be made about the language that lead to its incorrect use. Even if the application developers have an understanding of the syntax of a language, the semantics is the key to clarify the language's and concepts' meanings. The semantics in the context of MDD is usually introduced when transforming a PIM to a specific platform that offers some kind of execution semantics, however, our

aim is to introduce a clear semantics already at the PIM level to ensure that the generated models can already be validated on a more abstract level.

Benefits of Formal Languages A formal specification of DSML4MAS'S semantics can be used for different types of analysis purposes, different situations and applications.

Syntax checking (see e.g. (Shen et al.; 2002)) can be performed against the PIM4AGENTS metamodel to ensure that the created models conform to the PIM4AGENTS metamodel. The syntax check is in our case enhanced by the metamodel of PIM4AGENTS and the visual modeling tool provided by DSML4MAS that only allows creating of models that naturally fit to PIM4AGENTS.

Well-Formedness checking (see e.g. (Shen et al.; 2002)) can be performed against more complex statements that can be deduced from the static semantics of the PIM4AGENTS metamodel, but are not directly part of the abstract syntax.

Consistency checking (see e.g. (Bernardi et al.; 2002; Schäfer et al.; 2001)) is performed against different viewpoints of a system to ensure that those are consistent. This could mean that, for instance, consistency checker guarantees that the behavior model allows to implement the corresponding interaction models of the MAS.

Model checking (see e.g. (Kwon; 2000; Lilius and Porres Paltor; 1999a,b)) is considered as dynamic analysis performed on the finite state models to identify whether functionalities like liveness, deadlock, fairness, or reachability hold for feasible executions of the model.

In case of DSML4MAS, we consider all of the presented types important. The first three categories to ensure that any model specified in accordance to PIM4AGENTS can be applied to the code generators in order to ensure the production of meaningful output. The last category to provide means to do further analysis on the produced PIM4AGENTS models to estimate how the system behaves at run-time. To particularize a formal specification, a number of different semantical forms exist.

Different Forms of Semantics The semantics of a language is a mathematical defined function from the language's syntax to its semantic domain (Harel and Rumpe; 2004). It expresses the language's computational behavior in either a static or/and dynamic manner. This mathematical function can be defined in different ways, depending on purpose of the semantics. The mathematical function can be expressed in an either axiomatic, denotational, or operational manner.

Axiomatic Semantics The axiomatic semantics (cf. (Hoare; 1969)) are given by means of axioms relating expressions in the language. The axioms can be based on some underlying logic like OCL and are often defined in terms of invariants, pre- and postconditions.

Denotational Semantics The denotational semantics (cf. (Stoy; 1977)) are given by means of mathematical functions translating the given specification into a well-understood model having a precise semantics.

Operational Semantics The operation semantics (cf. (Plotkin; 2004)) are given by means of defining states and transitions between states. The operational semantics are in particular

used to describe the dynamic part of the language by describing the meaning of programs in terms of their execution steps taken by an abstract machine. The operational semantics are a necessary requirement when performing model checking on the programs.

Several options exist for giving a language a formal specification. Most of them allow the specification from an axiomatic, denotational and operational view. Selected formal specification languages are discussed in the following.

Categorizes of Formal Languages Generally, the term formal methods refers to the use of mathematical techniques in the design and analysis of computer hardware and software. These methods may use formal specification languages to describe the architecture as well as the behavior of the system designed and mathematically-based analysis techniques to demonstrate that particular properties such as liveness are satisfied by the system. Two major categories of formal methods can be distinguished as follows: One the one hand, model-oriented approaches for modeling systems' states and data including formal methods such as Z (Woodcock and Davies; 1996; Spivey; 1992), Alloy (Jackson; 2006), B method (Abrial; 1996), Object-Z (Smith; 2000) and, on the other hand, process algebras for modeling system behaviors and interactions including for instance Communicating Sequential Processes (CSP) (Hoare; 1978), LOTOS (Eijk and Diaz; 1989), π-calculus (Milner; 1999). In the following, selected approaches are discussed in more detail focusing in particular on model-oriented approaches that are feasible candidates for formalizing the semantics of DSML4MAS.

Z The Z language is a state-oriented formal specification language based on set theory and predicate logic. A Z specification typically includes a number of state and operation schema definitions. A state schema encapsulates variable declarations and related predicates (invariants). The system state is determined by values taken by variables subject to restrictions imposed by state invariants. An operation schema defines the relationship between the "before" and "after" states corresponding to one or more state schemata. Complex schema definitions can be composed from the simple ones by schema calculus.

Allow Alloy is a formal object-oriented specification language based on first-order logic and a notation called relational calculus that give it a mathematical notation for defining objects and their relationships. Models generated with Allow are comparable to UML models combined with OCL, but have a simpler syntax, type system and semantics, used for an automatic analysis. An Allow specification contains several so-called paragraphs representing signatures (comparable to the schemata in Z) used for defining types. Each signature introduces a basic type and a collection of relations to other signatures. As previously mentioned, UML does not base on a formal semantics (i.e. an adequate semantic mapping for the full UML does not exist; only textual descriptions are given.) However, for instance, in (Anastasakis et al.; 2007) a model transformation from UML and OCL to Alloy is given that allows the creation of analyzable UML models.

Object-Z Object-Z (Smith; 2000) is an object-oriented specification language that supports features like classes, instance, inheritance, and polymorphisms. Object-Z bases on and extends the specification language Z with object-oriented specification support and bases on mathematical concepts (like for instance sets, functions, and first-order predicate logic) that permits rigorous analysis and reasoning about the specifications. The basic construct is the class, which encapsulates a state schema with all the operation schemata, which may

affect its variables. A class further includes invariants that specify restrictions on the variables. Object-Z has been especially used to formalize UML. In (Mann and Klar; 1988), for instance, the authors formalized a metamodel of object-oriented statecharts. A generic model transformation between UML and Object-Z has been defined in (Kim et al.; 2001). This model transformation allows mapping between UML concepts and Object-Z concepts and thus enables the generic creation of formal UML models. A formalization of UML state machines using Object-Z has been given in (Kim and Carrington; 2002). Apart from these works, Object-Z is also very popular in the Semantic Web service community. In (Wang et al.; 2007a), for instance, the authors use Object-Z for the formal specification for the Semantic Web Service Ontology (WSMO). Wang et al. (2007b) demonstrated how to utilize Object-Z to formalizes OWL-S.

Formal Agent-based Approaches In (Lapouchnian and Lespérance; 2006), a CASL[2] (short for Cognitive Agents Specification Language) specification for i* is given that allows the formal analysis and verification. This approach has its similarities with Formal Tropos (Fuxman et al.; 2004), which supports the formal verification of i* models through model checking. An extension is presented in (Decreus and Poels; 2009) to acquire correctly semantically annotated business process models. In (Hilaire et al.; 2004), the authors describe a formal specification of a mobile robot architecture. The most prominent approach is proposed in (d'Inverno and Luck; 2001b) that is based on the specification language Z (Spivey; 1992). However, the development of MASs is purely restricted to the formal specification, no graphical visualization nor automatic code generation is offered. The authors of (Brandão et al.; 2004) propose an approach in which Object-Z is extended for specifying MASs. The authors of (Xu and Zhang; 2005) demonstrate how to use Object-Z to formalize the role model of their methodology.

All these approaches demonstrate that the formal specification of MASs is required in order to make the system behavior explainable. However, the presented works solely focus on the formal specification, without providing mechanisms for modeling, code generation, etc. This is certainly one main advantage offered by DSML4MAS.

DSML4MAS's Semantics Approach For formalizing the semantics of DSML4MAS, we consider Object-Z as formal mechanism as it allows the specification in a state-based and object-oriented manner, which is close to how PIM4AGENTS has been defined. The static semantics are defined by formalizing the concepts' attributes and invariants. The dynamic semantics are defined by specifying a denotational and operational semantics. The denotational semantics are defined in terms of introducing additional semantic variables and invariants. The operational semantics are defined in terms of class operations and invariants restricting the operation sequences that are specified using the timed trace notation of the timed refinement calculus (Smith and Hayes; 2000). A very similar approach, from a formal Object-Z specification into UML Class diagrams is discussed in (Chen and Miao; 2004). To allow the integration into the graphical framework, the resulting Object-Z specification is, in a second step, manually transformed into a corresponding OCL formalization, which is then integrated into the DSML4MAS development environment.

[2] CASL is a formal specification language that combines theories of action and mental states expressed in situation calculus with ConGolog.

3.2.1.3 Concrete Syntax of DSML4MAS

The concrete syntax of DSML4MAS is defined by a notation model, which is used to store visual information necessary for drawing DSML4MAS diagrams. It is independent from the underlying abstract syntax and semantics model. To realize a link between concrete and abstract syntax, a mapping was defined, which serves as input for the generation of a visual editor. This mapping is illustrated in Fig. 3.1 by the *CS to AS* mapping.

3.2.2 Model Transformations

Apart from the modeling language itself, code generators are an important ingredient for any DSML as autonomic transformations significantly reduce cost and time, which have to be spent to offer executable code. In the remainder of this section, selected model transformation engines are reviewed, which provide the means for implementing and running model transformations.

3.2.2.1 State of the Art on Model Transformation Engines

As previously discussed in Section 2.2.4, several model transformation engines exist to implement model transformations. In the remainder of this section, we briefly analyze four selected model transformation engines, in order to come to a decision which model transformation engine(s) to incorporate into the DSML4MAS architecture.

XMF Mosaic XMF Mosaic (Xactium[3]) provides a model-driven development framework for modeling, executing, constraint checking and deploying languages and tools. It is a commercial tool platform that provides modeling and programming capabilities including full support for MDD and most of the QVT standards. It is based on standards like UML, even if only UML class diagrams (without compositions and aggregations) are supported. To specify complex transformations, an OCL-like language called XOCL can be used. Due to its general purpose framework, transformations do not dependent on any core metamodel. Furthermore, models from other tools basing on the XMI can be imported. XMF provides a collection of classes that form the kernel (called XCore). XCore includes class definitions for the basic types including Integer, Boolean and String and collection types for sets and sequences of values. XCore is object-oriented and provides basic notions of object and class. XMF Mosaic includes two types of mapping languages, i.e. XMap is a pattern oriented mapping language expressing unidirectional mapping, XSynch is used in bi-directional synchronizations between models.

The main shortcoming of XMF Mosaic is that models are described in an XMF Mosaic-based format, which makes the integration of other languages very hard and reduces the language's interoperability.

Model Transformation Framework (MTF) As part of their involvement in the QVT standardization, IBM has developed the Model Transformation Framework (MTF[4]) that mainly consists of (i) a language to define mappings between EMF models and (ii) a transformation engine capable of interpreting mapping definitions. MTF implements some of the QVT concepts (however, MTF is not QVT compliant) and provides a declarative language for model transformations, along with a

[3] http://albini.xactium.com/content/
[4] http://www128.ibm.com/developerworks/views

transformation engine. As plug-in for Eclipse, MTF supports the transformation of EMF models and additionally allows the integration of Java code to extend the mapping definition language. Beside the transformation engine, it provides a set of tools to run and debug transformations.

The main shortcoming is that MTF is not very intuitive and it is very hard to learn how to use MFT correctly. Both issues reduce usability drastically. Moreover, the structural-driven approach supported by MTF shows inflexibilities in implementing model-to-model transformations.

Atlas Transformation Language (ATL) The Atlas Transformation Language (ATL) aims at providing a set of transformation tools that include a transformation repository, sample transformations and an ATL transformation engine. ATL is a hybrid language designed to express model transformations as required by MDD. It has been developed as part of the Atlas Model Management Architecture (AMMA), which is implemented on top of EMF and available as a plug-in for Eclipse. It is based on declarative rule definitions, which define mappings between source models and target models and supported by a set of development tools like, for instance, an editor and debugger. Originally, ATL was designed to express model-to-model transformations, however, model-to-text transformations can be expressed as well. ATL is similar to the QVT submission in terms of semantics, but differs in syntax. It is open source, with an increasing user community, and currently under continuous development.

MOFScript MOFScript is a model-to-text transformation language, developed to provide a code generation tool for arbitrary metamodels. The corresponding tool, which is packaged as Eclipse plug-in, provides the means of editing, compiling, and executing model-to-text transformations. Similar to the ATL language, MOFScript bases on the Ecore meta-metamodel and can be utilized with a rough understanding of Java necessary for developing mappings.

3.2.2.2 DSML4MAS's Transformation Engines

The necessary model transformations of the DSML4MAS architecture should facilitate the automated translation of high-level models conforming to the PIM4AGENTS metamodel into more low-level MAS representations on the PSM layer. In (Hahn et al.; 2006c), the model transformation engines of XMF Mosaic, MTF, and ATL were evaluated in accordance to a model transformation evaluation framework proposed by Gronmo et al. in (Gronmo et al.; 2005). Based upon this evaluation, we decided to include the model transformation engines of the ATL language for model-to-model transformations and the MOFScript language for model-to-text transformations. Apart form characteristics like traceability and tool support, one major reason was that both model transformation languages—as described in Section 3.2.2.1 and Section 3.2.2.1—conform to the Ecore meta-metamodel, which can be considered as the de-facto standard for DSML. This easily allows the integration of existing languages and metamodels into the tool suite and hence furthers the interoperability of DSML4MAS.

3.2.2.3 DSML4MAS's Model Transformation Architecture

Based on the selected model transformation engines, the model transformation architecture of DSML4MAS (see Fig. 3.4) has been defined. It mainly consists the core language of DSML4MAS, their relationship to the abstraction levels PIM and PSM as well as their relationship to other languages defined through model transformations, either model-to-model or model-to-text.

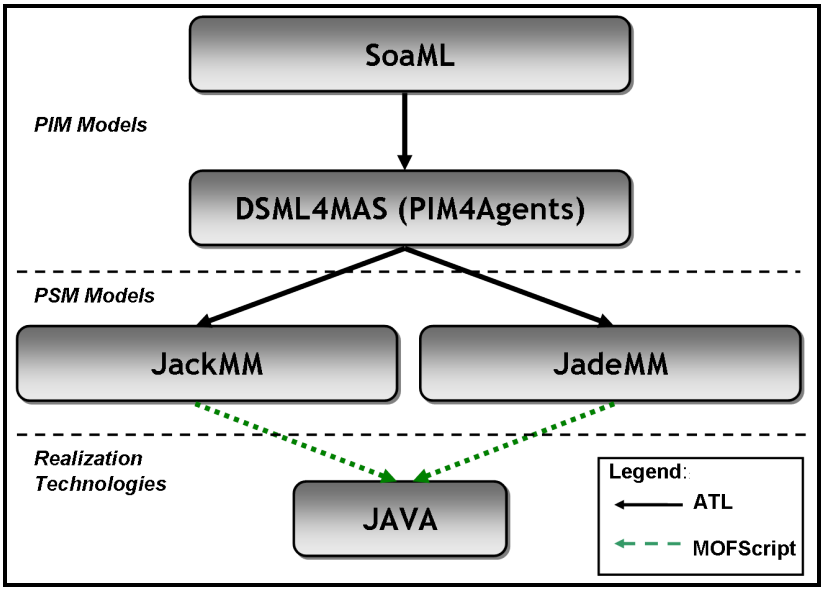


Fig. 3.4: The model transformation architecture of DSML4MAS.

On the highest level, the service models are situated, which are represented by SoaML. This architecture allows the realization of one of the main goals of dissertation namely to improve the interoperability between MASs and other software development approaches like service-oriented architectures by providing a generic approach mapping service models to MAS models, which are then transferred to the various execution platforms.

The lowest level shows the generated PSMs and their metamodels. These PSMs will be mapped to different technologies such as JACK and JADE to generate the executable artifacts. The architecture thus provides an integrated solution for service development that covers the life cycle of services from business goals and requirements to platform specific models for various platforms.

In the remainder of this section, we discuss the model transformation architecture in more detail by focusing on the abstraction levels, their core technologies, as well as, the model transformations needed. For this purpose, we distinguish between CIM to PIM transformations, PIM to PIM transformations, PIM to PSM transformations and finally, PSM to code (i.e. Java) transformations.

PIM to PIM Transformations To enhance the interoperability between Service-oriented Architectures (SOAs) and MASs, we developed a model-to-model transformation between SoaML and PIM4AGENTS, which automatically transfers SoaML models into PIM4AGENTS models. This transformation was implemented using ATL. The mapping details are discussed in Section 8.4.

PIM to PSM Transformations To provide code generation facilities in DSML4MAS, model transformations between PIM4AGENTS and two agent-based execution platforms are provided:

- Model transformation between PIM4AGENTS and JackMM: The model transformation between the PIM4AGENTS metamodel and the metamodel of Jack (JackMM) is presented in Section 7.3.1.

- Model transformation between PIM4AGENTS and JadeMM: The model transformation between the PIM4AGENTS metamodel and the metamodel of JADE (JadeMM) is presented in (Hahn et al.; 2009a; Gründel; 2009).

PSM to Code Model Transformations As a final step of the DSML4MAS model transformation architecture, the models in accordance to JackMM and JadeMM are mapped to executables artifacts for which the MOFScript model-to-text language is used.

- Model transformation between JackMM and Code: The generated Jack models conforming to the JACK metamodel (JackMM) are in a final step mapped into specific JACK code that can be imported by the JACK IDE. For this purpose we defined a model-to-text transformation using MOFScript which uses the generated JackMM models as input and produces the JACK-specific Gcode. When imported, the generated Gcode can easily be transferred into Java using the facilities provided by Jack and executed. Details are discussed in Section 7.3.2.
- Model transformation between JadeMM and Code: Like described in the case of JACK, for JADE we also defined a model-to-model transformation using MOFScript. However, in this case, we defined a transformation directly from JadeMM to Java code that can finally be executed. Details are given in (Fischer et al.; 2007).

3.2.3 Tool Support for Visualizing the Design

3.2.3.1 State of the Art on Tool Support

Creating a DSML from scratch is a very difficult job. The language engineer would have to take care of model representations, graphical editing, source code generation, and so on. To support the software engineer in the model representations and graphical editing, several alternative commercial and open source tools are provided, the most important ones are briefly discussed in the following.

Graphical Modeling Framework The Eclipse Graphical Modeling Framework (GMF[5]) provides a generative component and run-time infrastructure for developing graphical editors based on EMF and the Graphical Editing Framework (GEF) of Eclipse. It aims to simplify the combination of these two technologies by allowing GEF editors to be specified and generated using models.

XMF Mosaic XMF-Mosaic is a model-based development platform based on Eclipse (open source IDE) and XMF. XMF stands for (eXecutable Metamodelling Facility) and is an extension of the existing standards such as MOF, QVT or OCL, with executable metamodeling capabilities which provide the ability to create languages with all the key features like semantics and syntax, the ability to create metamodels which involves having a meta-architecture.

Visual Studio 2005 DSL Tools Visual Studio 2005 DSL Tools (Cook et al.; 2007) are a standard development kit that allows developers to use Microsoft's platform to build visual modeling tools that run inside of VisualStudio. Microsoft's vision is thereby to construct custom designed tools that can be used to model a certain problem domain. The steps necessary for the created of a

[5] public available at http://www.eclipse.org/gmf

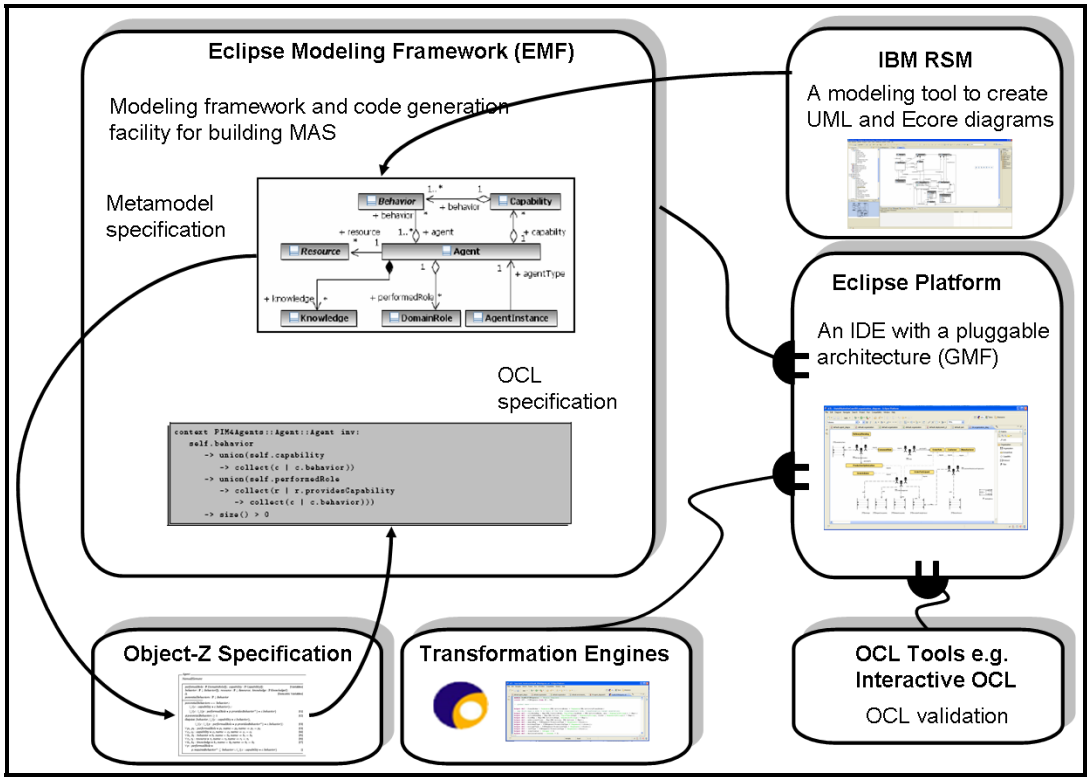


Fig. 3.5: The pluggable architecture of DSML4MAS.

custom design tool is to (i) create the model with the domain-specific notation and (ii) generate a running designer that is based on the domain model created.

Generic Modeling Environment The Generic Modeling Environment (GME) (Ledeczi et al.; 2001; Davis; 2003) is a meta-configurable tool that allows a DSML to be defined from a metamodel. Any model can be created using the DSML and may be translated into source code.

3.2.3.2 DSML4MAS's Tool Support

The DSML4MAS Development Environment (DDE) is the IDE for modeling in accordance to DSML4MAS (cf. Fig. 3.5). It bases on the GME, which bases again on the Ecore meta-metamodel and allows the graphical editing and validation of the generated design. Similarly, the developed model transformations were provided as plug-in to directly build executable code within the tool suite.

Reduction of Complexity To reduce the complexity of MAS design is one of the main objectives of the research area of AOSE. For this purpose, DDE offers several views on the MAS. Each view (e.g. agent view, protocol view, deployment view, etc.) focuses on a certain aspect and abstracts from others. Changes that affect several views are automatically propagated to the others.

Model validation Many design errors of a MAS can already be captured at the model level. DSML4MAS offers a formal semantics that can be used to check the syntactic correctness of the created models. For this purpose, constraints based on the OCL[6](OCL) have been manually derived from the formal Object-Z specification of DSML4MAS to check the static semantics of the models. These constraints are automatically evaluated during design time and support the developer to produce well-formed models, which is important to ensure that the code generators finally produce meaningful output.

Reusable Components DDE allows the user to reuse components (like plans, protocols, organizational structures, etc.) across several projects. This reduces development time and cost, and increases the quality of the components.

Extensibility DDE is seamlessly integrated into the Eclipse workbench. This implies that further extensions (e.g. transformations, views, model validation, etc.) can easily be defined and plugged into the Eclipse workbench. DDE thereby directly benefits from new developments around the very active Eclipse modeling project[7] and other Eclipse tools.

Open source The source code of DDE is published under LGPL (GNU Lesser General Public License) and available for download[8].

3.2.4 Process

To support the development process of agent-based system, DSML4MAS provides a semi-automatic process to build MAS applications. This process bridges the gap between the analysis phase and the implementation phase of DSML4MAS and hence includes different abstract levels (PIM and PSM) of the supported abstraction hierarchy. The different phases along the process are (semi-)automatically bridged through model transformations that are either *endogenous*, i.e. between viewpoints of the same abstraction level (PIM-to-PIM), or *vertical* i.e., between platforms on different abstraction levels (PIM-to-PSM). The methodology process model was formalized using the Eclipse Process Framework (EPF[9]) that aims at providing a customizable framework for software process engineering. For details on the DSML4MAS methodology, we refer to Chapter 5.

3.2.5 DSML4MAS's Architecture

The general architecture of DSML4MAS is depicted in Fig. 3.6. It consists of three main parts, i.e., the *artifacts*, the DSML4MAS *Development Environment*, and the *model repository*:

Artifacts In the DSML4MAS architecture, three sorts of artifacts are distinguished, which provide the necessary input for the DSML4MAS Development Environment. The DSML4MAS modeling language, the model transformations of the model transformation architecture as well as the agent-oriented programing languages.

The DSML4MAS language artifacts consisting of the PIM4AGENTS metamodel as abstract syntax, the Object-Z and OCL specification as formal semantics as well as the graphical

[6] http://www.omg.org/docs/ptc/03-10-14.pdf
[7] http://www.eclipse.org/modeling/
[8] http://sourceforge.net/projects/dsml4mas/
[9] available at http://www.eclipse.org/epf/

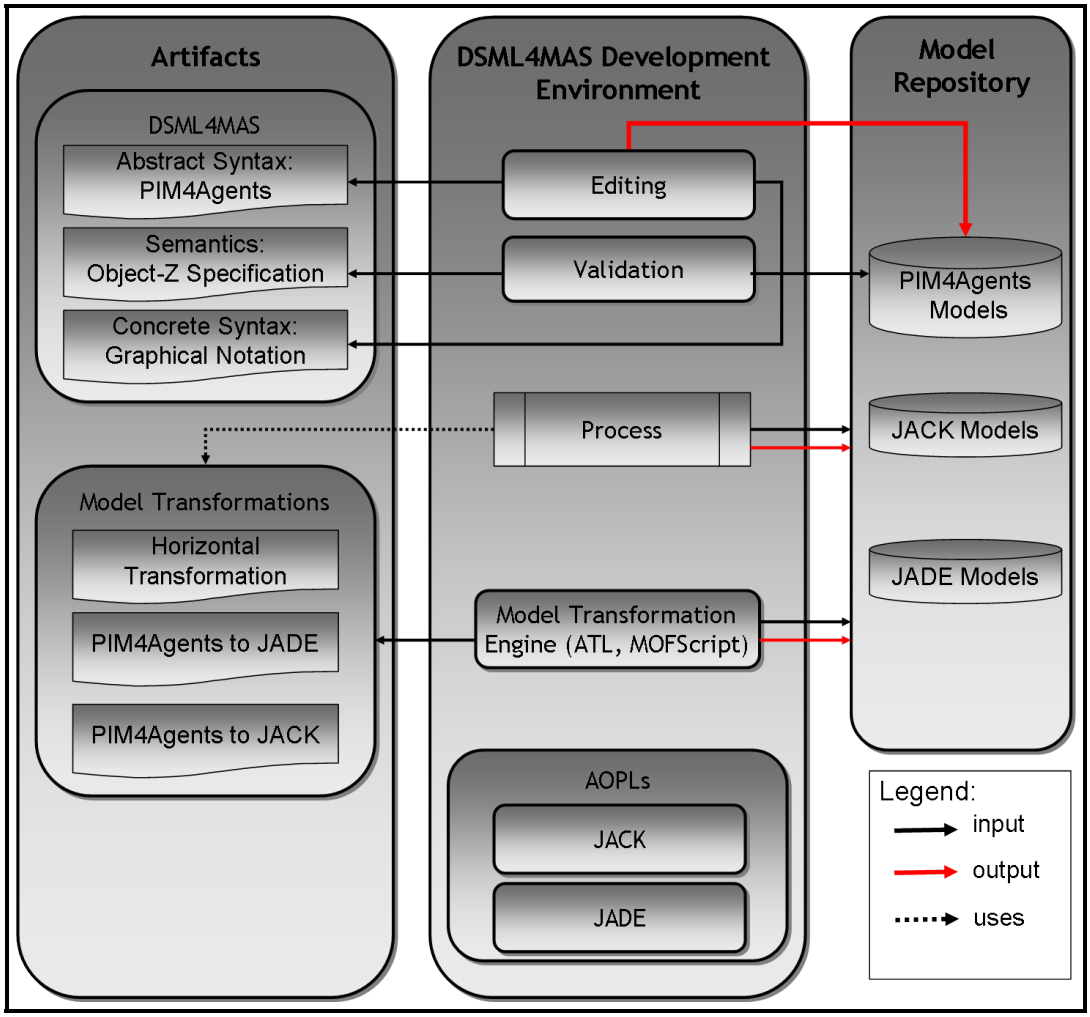


Fig. 3.6: The architecture of DSML4MAS.

notation as concrete syntax. The model transformation artifacts consist of (i) the model-to-model transformations between PIM4AGENTS on the one side and the metamodels of JACK and JADE on the other side, (ii) the model-to-text transformations between the metamodels of JACK and JADE and the corresponding textual representation interpreted by the AOPLs of JACK and JADE, which establish the third kind of artifact, and (iii) the horizontal transformations of DSML4MAS responsible for mapping viewpoints.

DSML4MAS Development Environment The DSML4MAS Development Environment provides mechanisms for graphical editing and validation of the design and integrated the model-driven process guiding the design as well as the model transformation engines as Eclipse plugins.

Model Repository Finally, the model repository contains the artifacts that are produced as part of the model-driven methodology. These artifacts includes the PIM4AGENTS models, as

well as the generated JACK and JADE models produced when applying the PIM-to-PSM model transformations. These artifacts can then be further imported and processed by the particular agent-based execution platform.

3.3 Bottom Line and Summary of Approach

This chapter presented the basic principles of language-driven development (LDD), which aims at focusing on the development of languages and tools that are tailored to the special needs of the developers to improve the system's development practice. The core ingredients of any language defined in accordance to LDD are an abstract syntax defining the vocabulary of the language, a concrete syntax defining its notations, as well as the semantics formalizing the meaning of the language. A Domain-Specific Modeling Language (DSML) as one concrete instance of LDD is a visual modeling language defined for a certain domain. For the purpose of realization, either UML profiles or metamodeling facilities can be applied. The former bases on customizing the existing UML, the latter allows building a new language from scratch. In the context of DSML4MAS due to the reason of creating new concepts with an unambiguous semantics, we favor the second option for building the domain-specific modeling language for MASs (DSML4MAS).

DSML4MAS is defined in accordance to LDD with the objective to close the gap between design and code. DSML4MAS'S core components include an abstract syntax defined by the PIM4AGENTS metamodel, a concrete syntax, and a formal semantics defined by Object-Z. The formal semantics is used to (i) further refine DSML4MAS'S abstract syntax and (ii) formalize the denotational and operational semantics. The syntax and semantics defined are used as a base to develop a graphical editor that finally formulates the concrete syntax. Syntax and semantics are expressed with OCL to guarantee that the developed models are well-formed. This is of special importance when applying the model transformation to the specific AOPLs. Besides the language features, the model transformation architecture of DSML4MAS includes two AOPLs, i.e. JACK and JADE that are provided to support the seamless execution of the generated design. The model transformation, as well as well-formedness rules are integrated into the DSML4MAS Development Environment, which is an Eclipse plug-in that is provided for download under http://sourceforge.net/projects/dsml4mas/.

Part II

Language Features of the Domain Specific Modeling Language for Multiagent Systems

4. Abstract Syntax and Semantics of DSML4MAS

As one of the core chapters of this dissertation, this chapter presents the abstract syntax and semantics of DSML4MAS. For this purpose, platform independent concepts, their attributes and semantics are discussed, which are, from a research perspective, necessary for designing MASs in a precise, rich and expressive manner. As already stated in (Lind; 2001), probably one of the hardest problems in developing a platform independent modeling language for MASs is to define the scope of the proposed constructs. The concepts should neither be too specific in that it covers only a small fraction of MAS applications, nor should they be too general, because unrelated details make the method less useful in the specific context. Therefore, the constructs are platform independent and base on general concepts relevant to any MAS and its models, problem-specific concepts are avoided as much as possible. To propose platform-specific concepts, Section 2.1 reviewed literature from the areas of agent software engineering, distributed AI and cognitive science in order to ground the adopted concepts on what other people in the agent community think.

Structure of this Chapter This chapter is organized as follows: Section 4.1 briefly introduces the different viewpoints we consider important in order to model MAS in a precise and adequate manner. Section 1.2 then gives a brief introduction into Object-Z to give the base for understanding the semantic descriptions of PIM4AGENTS. Afterwards, the abstract syntax and semantics of theses viewpoints are formalized in Sections 4.2 to 4.9. Finally, Section 4.10 concludes this chapter.

4.1 Eight Views on Designing Multiagent Systems

As discussed in Section 3.2, the abstract syntax of DSML4MAS is defined by the PIM4AGENTS metamodel. In order to support an evolution of PIM4AGENTS, it is structured into several views each focusing on a specific viewpoint of a MAS. Grouping modeling concepts in this manner allows the metamodel evolution by (i) adding new modeling concepts, (ii) extending existing modeling concepts, or (iii) defining additional viewpoints of MASs. The idea of dividing a system into different views is not new, even if it was revolutionized in the context of MDA (cf. Section 2.2.1).

In terms of MDA, complex systems are always seen from several different perspectives (viewpoints), and their separation into different views is a powerful means to reduce complexity and master their implementation. The viewpoint technique appears as a powerful means to address the system's complexity, and to organize the expertise of participants. The viewpoint technique, moreover, provides a means to represent and support each specific focus (using views), and to combine these focuses on models.

For information systems in general, Zachman (1987) developed a framework for enterprise architectures consisting of different aspects like data, function, etc. This idea of structuring architectures has also been adopted by the agent community (e.g. (Huget; 2002a; Lind; 2001)).

Lind, for instance, distinguished between seven views necessary for modeling MASs. These are the system view, environment view, role view, interaction view, society view, architecture view, and task view. In PIM4AGENTS, we propose slightly different views, which constitute the core modeling building blocks of DSML4MAS:

Agent viewpoint defines how to model single autonomous entities, the capabilities they have to solve tasks and the roles they play within the MAS. Moreover, the agent viewpoint defines to which resources an agent has access to and which kind of behaviors it can use to solve tasks—either in a reactive or proactive manner.

Organization viewpoint defines how single autonomous agents are arranged to more complex organizations that may be defined on the base of various different structures, each of them may be adequate for a certain problem solving scenario. *Organizations* in PIM4AGENTS can be either an autonomous acting entity like an agent, or simple groups that are formed to take advantage of the synergies of its members, resulting in an entity that enables products and processes that are not possible for any single individual.

Role viewpoint covers the abstract representations of functional positions of autonomous entities within an organization or other social relationships. In general, a role in PIM4AGENTS can be considered as set of features defined over a collection of entities participating in a particular context. The features of a role can include (but are not limited to) activities, permissions, responsibilities, and protocols. A role is a part that is played by an entity and can, as such, be specified in interactive contexts like collaborations.

Interaction viewpoint focuses on the exchange of messages between autonomous entities or organizations. Thereby, two opportunities are offered: On the one hand, the exchange of messages can be described from the internal perspective of each entity involved, or on the other hand, from a global perspective in terms of agent interaction protocols focusing on the global exchange of messages between entities.

Behavior viewpoint describes how the internal behavior of intelligent entities can be defined in terms of combining simple actions to more complex control structures or plans that are used for achieving predefined objectives or goals. The behavioral viewpoint contains basic concepts from workflow languages as well as particular tailored concepts for describing more agent-oriented processes.

Environment viewpoint contains any kind of entity that is situated in the environment and the resources that are shared between agents, roles or organizations to meet their objectives. The core environment mainly deals with how to define objects in terms of their attributes and operations.

Multiagent viewpoint contains the core building blocks for describing MASs. In particular, the agents situated in the MAS, the roles they play within collaborations, the kinds of behaviors for acting in a reactive and proactive manner, and the sorts of interactions needed for coordinating with other agents.

In addition to these core viewpoints to model and design agent-oriented software systems, the following extensions are provided to support the deployment and integration of (Semantic) Web services:

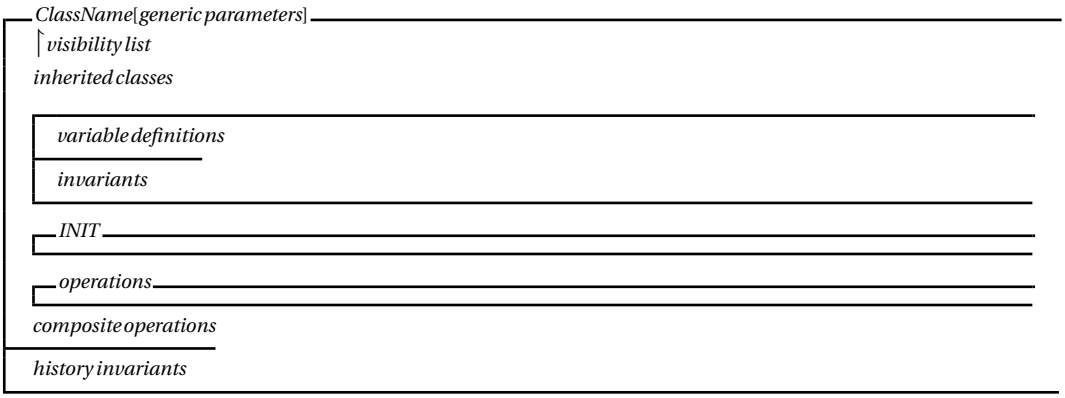


Fig. 4.1: A partial Object-Z class schema representation.

Deployment viewpoint describes the run-time agent instances involved in the system and how these are assigned to social structures like organizations and interactions.

Service-oriented Environment viewpoint describes how semantic services as a special kind of objects are described in terms of preconditions, postconditions and effects. A detailed discussion on this viewpoint is given in (Hahn et al.; 2008b).

These presented viewpoints provide a common baseline for agent-based computing to support an expressive and precise design of MASs, rich enough for full code generation. In the remainder of this chapter, we carefully examine the different views, their concepts and semantics.

4.1.1 A brief Introduction into Object-Z

as previously mentioned, Object-Z is an object-oriented specification language that supports features like classes, instance, inheritance, and polymorphism.

The most important features of an Object-Z specification are class schemata (see Fig. 4.1) that take the form of a named box with optionally a list of generic parameters. Furthermore, a class schema includes (i) a list of visibility that restricts the access to variables and operations, (ii) a list of inherited classes, (iii) a list of variable definitions and invariants, (iv an initial state schema INIT that specifies the initial state of the objects of the class, (v) a list of operations that specify pre- and post conditions, (vi) a list of composite operations of the class, as well as (vii) a set of history invariants that constrain the order of the operations.

The state schema in Object-Z consists of the set of declared variables and the corresponding class invariants. The operation schemata specify operations relating pre and post conditions of the object. Input variables are annotated by a question mark (?), output variables by an exclamation mark (!). A list marked with δ declares the set of variables that are changed by the operation.

Furthermore, Object-Z provides a set of operators that allow the combination of operations. This list of operators include the sequence operator ($op1 \fatsemi op2$), the conjunction operator ($op1 \wedge op2$), the choice operator ($op1 \,[]\, op2$), the parallel operator ($op1 \parallel op2$), as well as the operation enrichment ($schema \bullet op$).

The general idea is to specify for each concept of PIM4AGENTS an Object-Z class. This class consists of three parts: class attributes, class invariants, and class operations. The class attributes defines the syntax of the DSML4MAS. This mainly corresponds to the information contained in the PIM4AGENTS metamodel and defines whether a model is well-formed. For transformation purposes, UML's aggregation is mapped to Ⓢ, whereas the UML's composition is mapped to ©. The class invariants define the static semantics of DSML4MAS and and thus specify whether a model is meaningful or not. The class operation defines the dynamic semantics and declares whether a model can be interpreted and executed.

Denotational semantics are defined by introducing additional variables (we call these *semantic variables* to distinguish them from the variables that formalize the abstract syntax), which are used to define the semantics and invariants in Object-Z classes. Operational semantics are specified in terms of class operations (we call these operations *metaoperations*) and invariants restricting the operation sequences. We use the timed trace notation of the timed refinement calculus to define these invariants with Object-Z. With this approach, we give a mutually consistent (formal) denotational and operational semantics of the PIM4AGENTS behavioral viewpoint.

4.2 Multiagent System Viewpoint

A MAS consists, in accordance to the discussion in Section 2.1.2, of a collection of autonomous agents possibly situated in a dynamic and uncertain environment and able to engage in rich, high-level social interactions by potentially building flexible organizational structures.

To meet these requirements from an engineering point of view, in PIM4AGENTS, we introduce the multiagent system viewpoint depicted in Fig. 4.2. This viewpoint allows defining MASs on a very abstract level by introducing the core building blocks—like for instance *Agents*, *Interactions*, *Behaviors* etc.—necessary for designing and implementing MASs in precise manner. In the remainder of this section, we carefully debate on the multiagent system viewpoint and its concepts.

4.2.1 MultiagentSystem

In accordance with Section 2.1.2, the multiagent system view of PIM4AGENTS (see Fig. 4.2) comprises the core concepts of MASs, i.e. *Agent*, *Capability*, and *AgentInstance* (from the agent viewpoint, cf. Section 4.3), *DomainRole* (from the role viewpoint, cf. Section 4.5), *Behavior* (from the behavior viewpoint, cf. Section 4.7), *Interaction* (from the interaction viewpoint, cf. Section 4.6), *Message* and *Environment* (from the environment viewpoint, cf. Section 4.8). Furthermore, the *MultiagentSystem* inherits the attribute *name* from *NamedElement* (cf. Section A.1.1). The abstract syntax of *MultiagentSystem* is defined as follows:

Definition 4.2.1 (MultiagentSystem in PIM4AGENTS)

A MultiagentSystem is defined by a 9-tuple M = (name, agent, instance, role, behavior, interaction, capability, environment, message), where:

- *name: represents the unique identifier of the MultiagentSystem concept*
- *agent: represents all different kinds of Agent types situated in the MAS*
- *instance: represents all run-time AgentInstances available in the running system*
- *role: illustrates all different kinds of DomainRoles available to be played by Agents*
- *behavior: typifies internal Behaviors that are used by Agents for achieving goals*

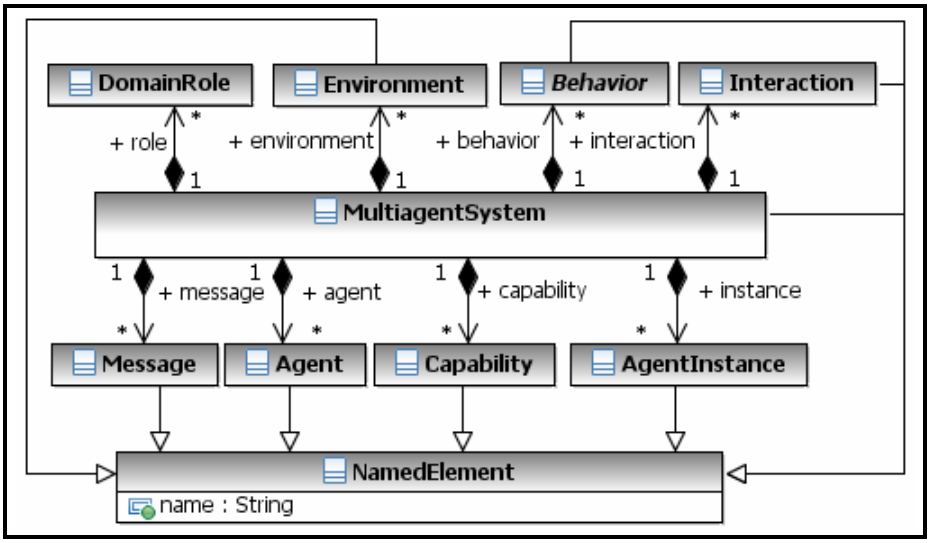


Fig. 4.2: The partial metamodel reflecting the multiagent system viewpoint of DSML4MAS.

- *interaction: typifies external behaviors (i.e. Interactions) used for the exchange of Messages between Agents of the MAS*
- *capability: defines all sorts of Capabilities that can be possessed by any entity within the MAS*
- *environment: constitutes the collection of Environments and contained Resources that can be accessed by any kind of entity (i.e. Agent or Organization) in the MAS*
- *message: represents the kind of Messages that are sent between Agents, possibly in accordance to the Interactions referred by the interaction attribute.*

Definition 4.2.1 specifies the abstract syntax of *MultiagentSystem* and thereby provides the minimal requirements to conform to Definitions 2.1.1 and 2.1.2 given in Section 2.1.2. The *MultiagentSystem* contains *Agents* that act in an autonomous manner and access capabilities for solving particular problems. Furthermore, it includes a set of *Environments* providing the basic infrastructure to allow computation and integration means for communication through the concept of *Interaction*.

The semantics of *MultiagentSystem* is expressed by Schema 4.2.1. Beside the primary variables already introduced in Definition 4.2.1 (e.g. agent, instance, role, etc.), further invariants were given refining the static semantics:

Invariants I1 to I8 state that all *Agents*, *AgentInstances*, *Capabilities*, *Interactions*, *DomainRoles*, *Behaviors*, *Environments* and *Messages* are unique, meaning that any two instances of them must be different with respect to their names, otherwise they are considered as equal. Furthermore, the *MultiagentSystem* is restricted to include at least one *Agent*[1] (cf. Invariant I9). Hence, the simplest form of a MAS conforming to PIM4AGENTS would contain one *Agent*, which owns certain *Behaviors* to achieve its goals. Consequently, the *MultiagentSystem* must not contain any *Message*, *Capability*, *Interaction*, *Environment* nor *DomainRole*. Obviously, when designing complex, real-world agent applications, these concepts are obligatory to model in an efficient and precise manner.

[1] However, if only one *Agent* is contained by the *MultiagentSystem*, we would rather call it an agent system.

MultiagentSystem

NamedElement

$agent : \mathbb{P} \downarrow Agent©$; $instance : \mathbb{P}\, AgentInstance©$; $capability : \mathbb{P}\, Capability©$ $interaction : \mathbb{P} \downarrow Interaction©$; $role : \mathbb{P}\, DomainRole©$ $behavior : \mathbb{P} \downarrow Behavior©$; $environment : \mathbb{P} \downarrow Environment©$; $message : \mathbb{P}\, Message©$	[Variables]
$\forall d_1, d_2 : agent \bullet d_1.name = d_2.name \Rightarrow d_1 = d_2$	[I1]
$\forall i_1, i_2 : instance \bullet i_1.name = i_2.name \Rightarrow i_1 = i_2$	[I2]
$\forall m_1, m_2 : message \bullet m_1.name = m_2.name \Rightarrow m_1 = m_2$	[I3]
$\forall c_1, c_2 : capability \bullet c_1.name = c_2.name \vee c_1.behavior = c_2.behavior \Rightarrow c_1 = c_2$	[I4]
$\forall e_1, e_2 : environment \bullet e_1.name = e_2.name \Rightarrow e_1 = e_2$	[I5]
$\forall r_1, r_2 : role \bullet r_1.name = r_2.name \Rightarrow r_1 = r_2$	[I6]
$\forall b_1, b_2 : behavior \bullet b_1.name = b_2.name \Rightarrow b_1 = b_2$	[I7]
$\forall i_1, i_2 : interaction \bullet i_1.name = i_2.name \Rightarrow i_1 = i_2$	[I8]
$\#agent \geq 1$	[I9]

Schema 4.2.1: Class schema of *MultiagentSystem*

4.2.2 Message

Messages are an essential means in MASs to describe the communication between agents. In accordance to (Caire et al.; 2002), a message is an object communicated between agents. Transmission of a message takes finite time and requires an action to be performed by the sender and also the receiver. The attributes of a message specify the sender, receiver, a speech act (categorizing the message in terms of the intent of the sender) and the content.

In PIM4AGENTS, we distinguish between two sorts of messages, i.e. *Message* and *ACLMessage*. A *Message* can be considered as an object that is communicated between *Agents* within a certain *Behavior*. It comprehends the particular content intended to be exchanged by the sending and receiving *AgentInstances*. The actions necessary for sending (i.e. *Send* activity, cf. Section 4.7.14) and receiving (i.e. *Receive* activity, cf. Section 4.7.14) *Messages* are part of the *Plan's* constructs.

In contrast, an *ACLMessage* (cf. Section 4.6.6) is in particular used inside interactions to describe the message between *Actors*, which are the generic place holders within *Interactions*. A *Message* normally realizes an *ACLMessage* (cf. Section 4.6.6), however, the latter further includes the idea of performatives (i.e. speech acts). The abstract syntax of *Message* is given in Definition 4.2.2.

Definition 4.2.2 (Message in PIM4AGENTS)

A Message is given by a 6-tuple M = (name, sender, receiver, content, ontology, aclMessage), where:

- *name: defines the name of the Message*
- *sender: represents the AgentInstance that sends this particular Message*
- *receiver: represents the AgentInstance that receives this particular Message*
- *content: specifies the content of the Message*
- *ontology: represents the set of ontologies useful to understand the Message's content*
- *aclMessage: links the Message optionally to an ACLMessage, which is realized by the Message.*

The class schema of *Message* is depicted in Schema 4.2.2. Beside the primary variables introduced in Definition 4.2.2, it includes four invariants: Invariant I1 ensures that any *Message* realizes at

most one *ACLMessage*. The same holds for the *sender* and *receiver* variables, meaning that at most one sender and receiver must be defined (cf. Invariant I2).

Message

NamedElement

$sender, receiver : \mathbb{P}\, AgentInstance$	[Variables]
$content : \mathbb{P}\, Knowledge;\ ontology : \mathbb{P}\, Ontology;\ aclMessage : \mathbb{P}\, ACLMessage$	
$\#aclMessage \leq 1$	[I1]
$\#sender \leq 1 \land \#receiver \leq 1$	[I2]
$sender \cap receiver = \varnothing$	[I3]
$aclMessage \neq \varnothing \Rightarrow \#sender = \#receiver = 0$	[I4]

Schema 4.2.2: Class schema of *Message*

Moreover, Invariant I3 states that in any case the sending and receiving *AgentInstances* must be disjoint. If a *Message* refers to an *ACLMessage*, finally, the number of sending and receiving *AgentInstances* is restricted to 0 as in this case the *AgentInstances* receiving and sending the particular message are indirectly given through the variable *instancesBound* (cf. Schema 4.9.4) of the *Actor's ActorBinding*.

4.3 Agent Viewpoint

Summarizing the discussion about the definition of agent in Section 2.1.3, we can conclude that an agent is an intelligent component that is situated in an environment able to act in an autonomous manner and to interact with other agents. For the purpose of communicating and achieving its design objectives, an agent has certain forms of behaviors available that are either pre-defined by the system designer or composed at design time.

How these requirements are met in PIM4AGENTS is represented in the agent viewpoint which is depicted in Fig. 4.3. This viewpoint includes the concepts *Agent, Knowledge* and *Capability*, as well as *AgentInstance* (from the deployment viewpoint, cf. Section 4.9), *Behavior* (from the behavioral viewpoint, cf. Section 4.7), *DomainRole* (from the role viewpoint, cf. Section 4.5), and *Resource* (from the environment viewpoint, cf. Section 4.8) and thus merges the core viewpoints of PIM4AGENTS.

The concept of *Agent* as type defines a classification of autonomous entities that can adapt to and interact with their environment. It mainly specifies individual agent features such as knowledge, behavior, roles etc. that characterize its *AgentInstances*. This means that the *Agent* type is a metaclass for specifying physical design-level agents. An *AgentInstance*—the run-time agent—is defined as the autonomous entity also known as the agent, which is situated in an agent-based software system. It can be humans, machines, software, or any other entity that act as in an agent-based manner. A software agent, then, is an entity that interacts with its environment and has some degree of autonomy.

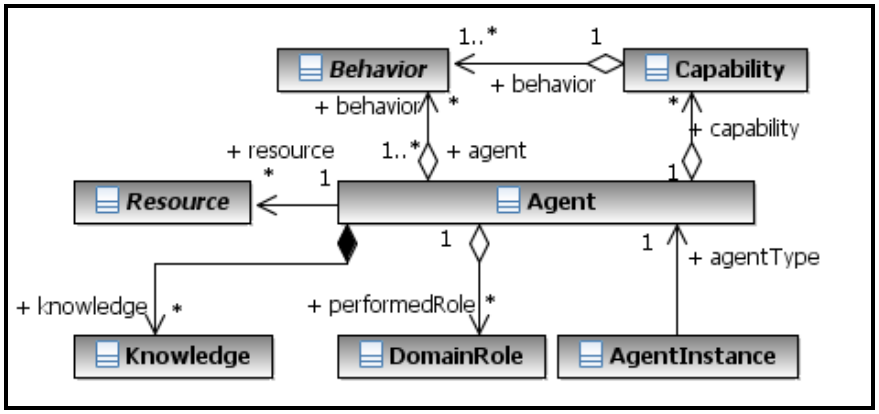


Fig. 4.3: The metamodel reflecting the agent viewpoint of PIM4AGENTS.

4.3.1 Agent

The agent viewpoint in PIM4AGENTS is centered on the concept of *Agent*, the type defining the autonomous entity capable of acting in the environment. An *Agent* has access to a limited set of *Resources* from its surrounding *Environment* (cf. Section 4.8). These *Resources* may include information or ontologies the *Agent* has access to. Furthermore, it can (i) perform particular *DomainRoles* that define in which specific context it is acting within an *Organization* and (ii) make use of certain *Behaviors* that specify how particular tasks are achieved. Apart from *Behaviors*, the *Agent* may also make use of *Capabilities* that are utilized to group particular *Behaviors* needed in certain domains. Moreover, private *Knowledge* represents the information (i.e. beliefs) an *Agent* could have about the world used for among others for making decisions. Finally, to support the run-time level, certain run-time entities called *AgentInstances*—representing a particular *Agent* type—can be introduced needed when executing the design made with DSML4MAS with an agent-oriented programming language. The abstract syntax of the concept *Agent* is given by Definition 4.3.1:

Definition 4.3.1 (Agent in PIM4AGENTS)

An Agent is given by a 6-tuple A = (name, performedRole, capability, behavior, resource, knowledge), where:

- *name: defines the name of the Agent*
- *performedRole: defines the DomainRoles the Agent performs in a social context like Organizations*
- *capability: contains different kinds of grouped Behaviors an Agent makes use of in particular contexts*
- *behavior: specifies the internal behavioral elements the Agent could execute for achieving internal goals*
- *resource: represents the kind of Resources, services or information the Agent makes use of*
- *knowledge: represents the kind of Knowledge available to an Agent for—among others—the purpose of deliberation and reasoning.*

The term agent as formalized in Definition 4.3.1 nicely corresponds to the weak notion of agency given in Definition 2.1.5 by Wooldridge and Jennings. In PIM4AGENTS, an agent (i) may act in

an autonomous manner in its environment—represented by the resources the agent has access to—through applying plans (i.e. behavioral elements) that do not necessarily need the intervention of human beings, (ii) may react on changes in the environment through plans which may again change the state of the environment on which other agents have to adapt, (iii) may act in a proactive manner by selecting the most adequate plan for achieving a certain goal, and (iii) may able to interact and communicate with other agents within *Organizations*.

Even if most of the agent-oriented metamodels, which will be discussed in Chapter 10, do not provide any means for defining run-time instances, however, we consider the distinction between types and instances of agents in MASs of particular importance. In PIM4AGENTS, the concept *Agent*—as types defined at design time—is used as a kind of classifier to provide means to classify *AgentInstances*—entities acting at run-time—by features and characteristics. This classification is important, because it enables the definition of a set of entities that share one or more capabilities and/or features in common (Odell et al.; 2005).

Agent

NamedElement

$perfomedRole : \mathbb{P}\, DomainRole\textcircled{S}$; $capability : \mathbb{P}\, Capability\textcircled{S}$ [Variables]

$behavior : \mathbb{P} \downarrow Behavior\textcircled{S}$; $resource : \mathbb{P} \downarrow Resource$; $knowledge : \mathbb{P}\, Knowledge\textcircled{C}$

Δ [Semantic Variables]

$potentialBehaviors : \mathbb{P} \downarrow Behavior$

$potentialBehaviors == behavior \cup \bigcup \{c : capability \bullet c.behavior\} \cup$
$\quad \bigcup \{c : \bigcup \{p : performedRole \bullet p.providesBehavior^{+}\} \bullet c.behavior\}$ [I1]

$\#potentialBehaviors \geq 1$ [I2]

$\text{disjoint} \langle behavior, \bigcup \{c : capability \bullet c.behavior\},$
$\quad \bigcup \{c : \bigcup \{p : performedRole \bullet p.providesBehavior^{+}\} \bullet c.behavior\} \rangle$ [I3]

$\forall p_1, p_2 : performedRole \bullet p_1.name = p_2.name \Rightarrow p_1 = p_2$ [I4]

$\forall c_1, c_2 : capability \bullet c_1.name = c_2.name \Rightarrow c_1 = c_2$ [I5]

$\forall b_1, b_2 : behavior \bullet b_1.name = b_2.name \Rightarrow b_1 = b_2$ [I6]

$\forall r_1, r_2 : resource \bullet r_1.name = r_2.name \Rightarrow r_1 = r_2$ [I7]

$\forall k_1, k_2 : knowledge \bullet k_1.name = k_2.name \Rightarrow k_1 = k_2$ [I8]

$\forall p : performedRole \bullet p.requiresBehavior^{+} \subseteq behavior \cup \bigcup \{c : capability \bullet c.behavior\}$ [I9]

Schema 4.3.1: Class schema of *Agent*

The semantics of the *Agent* concept is formalized in Schema 4.3.1. Its declarative part consists of the variables (i.e. *performedRole*, *capability*, *behavior*, *resource* and *knowledge*) discussed in Definition 4.3.1. Beside those primary variables, the semantic variable *potentialBehaviors* is defined, which specifies all kinds of behaviors the AgentInstance may adopt during run-time. The variable *potentialBehaviors* consists of (i) *Behaviors* the *Agent* may have direct access to through the variable *behavior*, (ii) the kinds of *Behaviors* grouped by the *Capabilities* through the variable *capability*, or (iii) *Behaviors* specific to any kind of *Role* the *Agent* performs (e.g. *Capabilities* a *DomainRole* offers to an *Agent* to perform).

The reason for introducing the variable *potentialBehaviors* is to ensure that an *Agent* is equipped with at least one behavior, expressed by the term $\#potentialBehaviors \geq 1$ of Invariant I1. This constraint is important to ensure that any *Agent* designed with DSML4MAS is able—at least on a conceptual level—to act in a reactive and/or proactive manner as requested in Definition 2.1.5. This would be impossible without having suitable behavioral elements. Moreover, the

potentialBehaviors are further restricted by Invariant I2 in the form that any *Behavior* either used directly by the *Agent*, through the *Capabilities* or performed *DomainRoles* have to be different. Moreover, the Invariants I4 to I8 assure that all *DomainRoles*, *Capabilities*, *Behaviors*, *Resources* and *Knowledges* an *Agent* refers to are unique, meaning that any two instances of them must be different with respect to their names. Finally, any *Behavior* required by a performed *DomainRole* must either be provided by the *Agent's Behavior* or *Capabilities*.

4.3.2 Capability

A *Capability* in PIM4AGENTS allows to group *Behaviors* that, conceptually, have a correspondence with regard to what they allow a particular entity—which could either be an *Agent* or *Role*—to do. The abstract syntax of *Capability* is defined as follows:

Definition 4.3.2 (Capability in PIM4AGENTS)

A Capability is given by a pair C = (name, behavior), where name defines the name of the Capability and behavior specifies the kinds of Behaviors that are grouped by this Capability.

The class schema of the *Capability* concept is given below. It includes the variable *behavior* (as specified in Definition 4.3.2) and refine the semantics of a *Capability* by Invariant I1 stating that at least one *Behavior* needs to be addressed by a *Capability*. Moreover, any *Behavior* grouped by a *Capability* must be different (cf. Invariant I2).

Capability

NamedElement

$behavior : \mathbb{P}_1 \downarrow Behavior$Ⓢ	[Variables]
$\#behavior \geq 1$	[I1]
$\forall b_1, b_2 : behavior \bullet b_1.name = b_2.name \Rightarrow b_1 = b_2$	[I2]

Schema 4.3.2: Class schema of *Capability*

4.3.3 Knowledge

Knowledge is used in PIM4AGENTS to represent the agent-specific beliefs used inside plans. The abstract syntax of an *Agent's Knowledge* is defined as follows:

Definition 4.3.3 (Knowledge in PIM4AGENTS)

A Knowledge is given by a triple K = (name, type, value), where name defines the name of the Knowledge, type defines its type and value its initial value.

A formal specification of *Knowledge* is given in Schema A.2.1, which mainly consists of the primary variables given in Definition 4.3.3.

The concept of an *Agent* defines the autonomous entity with a MAS, however, in order to define social units establishing a framework for interaction and cooperation, further concepts are needed. Therefore, in PIM4AGENTS, the concept of *Organization* is introduced, which is together with its related viewpoint, discussed in detail in the forthcoming section.

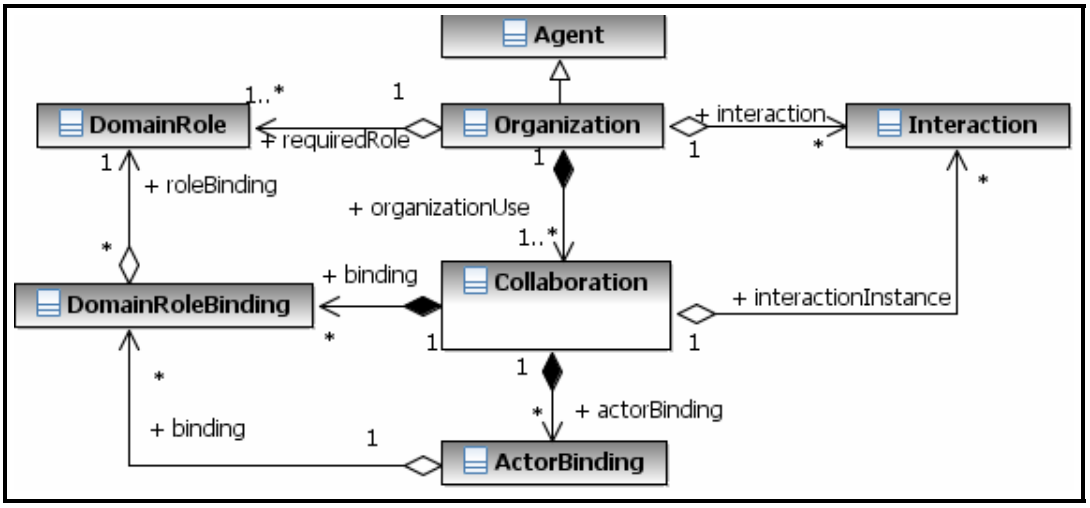


Fig. 4.4: The metamodel reflecting the organization viewpoint of PIM4AGENTS.

4.4 Organization Viewpoint

Section 2.1.5 debates the notion of organization in MAS. The definition given in (Wooldridge et al.; 2000) nicely summarizes this debate. Wooldridge et al. see an organization as a collection of roles, that stand in certain relationships to each other, and that take part in systematic institutionalized patterns of interactions with other roles.

The authors' viewpoint nicely conforms to the manner how we see the term organization in PIM4AGENTS. In particular, the organization viewpoint emphasizes on the concept of *Organization*, which specifies the general social structure through the concept of *DomainRoles*. As part of an *Organization*, several *Collaborations* then define how the particular *DomainRoles* finally interact with each other. This is done by specifying how the *Organization's* forms of interaction are utilized by the *DomainRoles*.

The metamodel of the organization viewpoint is depicted in Fig. 4.4. Beside the concepts of *Organization* and *Collaboration*, the metamodel of the organization viewpoint further includes the concepts of *Agent* (cf. Section 4.3.1) from the agent viewpoint, *DomainRole* (cf. Section 4.5.2) from the role viewpoint and *Interaction* (cf. Section 2.1.6) from the interaction viewpoint. Moreover, for defining bindings between *DomainRoles* and *Actors*, the organization viewpoint contains the concepts of *DomainRoleBiding* and *ActorBinding* from the deployment viewpoint (cf. Section 4.9).

4.4.1 Organization

In PIM4AGENTS, an *Organization* defines the social structure *Agents* can take part in and is commonly formed to regulate, foster, support, or facilitate the interaction between its members. In other words, an *Organization* enables a purposeful and specialized domain in which *Agents* may collaborate through *Interactions*. The structure of an *Organization* in terms of its scope of responsibility is specified by *DomainRoles*. These are required by an *Organization* to achieve a certain set of goals by dividing each goal into sub-goals and delegating the responsibility of achieving each particular sub-goal to its required *DomainRoles*.

The concept of *Collaboration* then particularizes how the participating *DomainRoles* collaborate. A *Collaboration* is characterized through *Interactions* that define (i) how the *Organization* communicates with other *Agents* be them atomic *Agents* or complex *Organizations* and (ii) how organizational members represented by the required *DomainRoles* are coordinated. The manner in which *AgentInstances* are bound to *DomainRoles* is expressed through the concept of DomainRoleBinding, which is part of the deployment viewpoint (cf. Section 4.9). The abstract syntax of *Organization* is defined as follows:

Definition 4.4.1 (Organization in PIM4AGENTS)

An Organization is given by a 9-tuple O = (name, requiredRole, interaction, organizationUse, performedRole, capability, behavior, resource, knowledge), where:

- *name: defines the name of the Organization*
- *requiredRole: defines the DomainRoles the Organization needs to assign in order to achieve the intended goals*
- *interaction: represents the different forms of Interaction used inside the Organization to coordinate its DomainRoles*
- *performedRole: defines the DomainRoles performed by the Organization*
- *organizationUse: refers to the Collaborations instantiating the Organization*
- *capability: depicts the Capabilities this organizational type applies to act in a reactive and/or proactive manner*
- *behavior: includes any Plan used to orchestrate the Organization's DomainRoles*
- *resource: describes all different kinds of Resources part of the Environment the Organization has access to*
- *knowledge: defines the Knowledges the Organization and its members have access to.*

An *Organization* in PIM4AGENTS can be used in two different manners: On the one hand, the *Organization* can be used to provide a social structure to foster the interaction of its members neither being an autonomous entity itself nor having capabilities to behave in a reactive nor proactive manner. On the other hand, as an *Organization* is a special kind of *Agent*, it can be considered as an autonomous and intelligent entity that can interact with other *Agents* or *Organizations*. Hence in the latter case, like an *Agent*, an *Organization* may perform *DomainRoles* to interact with other *Agents* and may make use of Behaviors to act in an autonomous, proactive and/or reactive manner. In either case, the *Organization* may be composed of other *Organizations*, providing the ability to create hierarchies in terms of sub-organizations, sub-sub-organizations, etc. In this way, an *Organization* provides building blocks that enable conceptual and social scalability.

The formal semantics of an *Organization* is given in Schema 4.4.1. The corresponding schema inherits from the *Agent* schema and additionally includes in its declarative part the variables *requiredRole*, *interaction* and *organizationUse*. The static semantic is formalized by the following invariants:

For any kind of collaboration and interaction inside an *Organization*, at least two entities are needed. If the *Organization* acts itself as intelligent *Agent* and, hence, posses any kind of Behavior for sending and receiving messages, it can collaborate with its required *DomainRoles* through its performed *DomainRoles*. Consequently, in this case, a single required *DomainRole* is sufficient to meet the requirement of having at least two entities for collaboration.

However, if an *Organization* does not possess any kind of Behavior (either directly or through the concept of *Capability*), it requires at least two *DomainRoles* for the purpose of collaboration. This is ensured by Invariant I1. If an *Organization* does not possess any Behavior, we consider the

Organization

Agent

$$
\begin{array}{l}
requiredRole : \mathbb{P}_1\ DomainRole\textcircled{S};\ interaction : \mathbb{P} \downarrow Interaction\textcircled{S} \\
organizationUse : \mathbb{P}_1\ Collaboration\textcircled{S} \qquad \text{[Variables]} \\
\hline
behavior \cup \bigcup \{c : capability \bullet c.behavior\} = \varnothing \Rightarrow \#requiredRole \geq 2 \wedge performedRole = \varnothing \qquad \text{[I1]} \\
\forall o : organizationUse \bullet \{dr : o.binding \bullet dr.roleBinding\} \subseteq requiredRole \cup performedRole \qquad \text{[I2]} \\
\forall o : organizationUse \bullet o.interactionInstance \subseteq interaction \qquad \text{[I3]} \\
\forall c_1, c_2 : organizationUse \mid c_1 \neq c_2 \bullet \{dr : c_1.binding \bullet dr.roleBinding\} = \\
\qquad \{dr : c_2.binding \bullet dr.roleBinding\} \Rightarrow \\
\quad \exists i : AgentInstance \mid \\
\qquad i \in \{ai : \bigcup \{m : c_1.binding \bullet m.membership\} \bullet ai.agentInstance\} \vee \\
\qquad i \in \{ai : \bigcup \{m : c_2.binding \bullet m.membership\} \bullet ai.agentInstance\} \bullet \\
\qquad i \notin \{ai : \bigcup \{m : c_1.binding \bullet m.membership\} \bullet ai.agentInstance\} \cap \\
\qquad\quad \{ai : \bigcup \{m : c_2.binding \bullet m.membership\} \bullet ai.agentInstance\} \qquad \text{[I4]} \\
\forall o : organizationUse \bullet \bigcup \{drb : \bigcup \{ab : o.actorBinding \bullet ab.binding\} \bullet drb.roleBinding\} \subseteq \\
\qquad requiredRole \cup performedRole \qquad \text{[I5]} \\
\forall p : performedRole \bullet p.requiresBehavior^+ \subseteq behavior \cup \\
\qquad \bigcup \{c : capability \bullet c.behavior\} \cup \\
\qquad \bigcup \{r : requiredRole \bullet r.requiresBehavior^+ \cup r.providesBehavior^+\} \qquad \text{[I6]}
\end{array}
$$

Schema 4.4.1: Class schema of *Organization*

Organization as simple social structure like a community, where *Agents* only take part in for the purpose of interaction. Hence, in this case, the *Organization* itself should not be considered as intelligent entity at all, but as social space providing mechanisms for interaction.

Furthermore, Invariant I2 restricts the set of *DomainRoles* a *Collaboration* uses to the set of either required or performed *Organization's DomainRoles*. In the same manner, Invariant I3 restricts the *Interactions* referred to by a *Collaboration* to the *Interactions* that can be applied by its *Organization*. Invariant I4 states that any two different *Collaborations* of an *Organization*, which require the same set of *DomainRoles*, must at least refer to one different AgentInstance through the Membership concept (cf. Section A.6.1). Otherwise, they are considered as equal. Moreover, the *Collaboration's ActorBindings* must only refer to *DomainRoles* (through the *DomainRoleBindings*), which are addressed by the *Organization* as either performed or required (cf. Invariant I5). Finally, Invariant I6 guarantees that any Behavior required by the provided *Organization's DomainRoles* must either be provided by its own Behaviors, *Capabilities* or by the provided and required Behaviors of its required *DomainRoles*. This invariant ensures that any goal assigned to an *Organization* can able be achieved by the *Organization* itself or its members.

Even if the concept of *Organization* lays the foundation for *social ability*, it only manifests the structure of the *Organization* by characterizing which *DomainRoles* are part and which *Interactions* are used by the *DomainRoles* addressed. However, it does not make assumptions about which of its *DomainRoles* interact in which manner in its social context. For this purpose, the concept of *Collaboration* is utilized that is discussed in detail in the following.

4.4.2 Collaboration

In PIM4AGENTS, the concept of *Collaboration* defines the relationship between *DomainRoles* required by an *Organization* and *Actors* of an *Interaction*. This means in particular that the *Collaboration* names the *DomainRoles*—through the *DomainRoleBindings*—that interact with each other within the boundaries of a certain *Interaction*. An *Interaction* in PIM4AGENTS (cf. Section 2.1.6) is considered as pattern guiding the communication between entities within *Organizations* for coordinating their members. For the purpose of combining the organization and interaction viewpoint, the concept of *ActorBinding* (cf. Section 4.9.4) of the deployment viewpoint is utilized that assigns *DomainRoles* to *Actors*. The abstract syntax of *Collaboration* is defined as follows:

Definition 4.4.2 (Collaboration in PIM4AGENTS)

A Collaboration *is given by a 4-tuple C = (name, binding, interactionInstance, actorBinding), where:*

- *name: defines the name of the Collaboration*
- *interactionInstance: depicts the different types of Interactions the Collaboration instantiates*
- *binding: defines the Collaboration's bindings between AgentInstances and DomainRoles*
- *actorBinding: describes the Collaboration's bindings between Actors of is Interactions and DomainRoles of its Organization.*

A *Collaboration* specifies one *DomainRoleBinding* for each *DomainRole* its *Organization* either provides or requires. A *DomainRoleBinding* (cf. Section 4.9.3) defines the bindings between several *AgentInstances* and one *DomainRole*. The relationship between both can thereby either be determined at design time by explicitly assigning *AgentInstances* to *DomainRoles* or during run-time. In the latter case, special activities are provided to allow the dynamic assignment of *AgentInstances* within Plans.

The formal semantics of *Collaboration* is given in Schema 4.4.2. Beside the primary variables *interactionInstance*, *binding* and *actorBinding*, we further refine the semantics by the following twelve invariants.

Invariant I1 supports that all *ActorBindings* the *Collaboration* makes use of are unique; the same is specified for *DomainRoleBindings* in Invariant I2. Even if not all *Organization's DomainRoles* must be addressed by a *Collaboration*, at least two of them are needed for the purpose of interaction. This is formalized by Invariant I3. In addition, Invariant I4 guarantees that the set of *Actors* part of any *Interaction* the *Collaboration* makes use of (through the variable *interactionInstance*) is equal to the *Actor* referred to by the *Collaboration's ActorBindings*. In combination with Invariant I5, denoting that at least one *Interaction* must by part of a *Collaboration*, it is obvious that for any *Actor* of a *Protocol* at least one *ActorBinding* must be specified.

Furthermore, Invariant I6 states that the set of *AgentInstances* bound to *Actors* through the variable *instancesBound* (see Section 4.9.4) acting as subactors of the same superactor are disjoint, i.e., any AgentInstance must not be part of two or more subactors of the same superactor. This constraint is necessary to ensure that *AgentInstances* bound to a *subactor* only receive *ACLMessages* that are unambiguous, i.e. the corresponding *AgentInstances* know how to react on the particular messages. Invariant I7 guarantees that if *ActorBindings* of subactors are specified, the *AgentInstances* bound to the subactors are subsets of the set of *AgentInstances* bound to the superactor. Further information on the concept of *Actor* can be found in Section 4.5.3.

In the context of a *Collaboration*, Invariant I8 further ensures that if two *DomainRoles* have a conflict with each other (we refer to Section 4.5.1 for details on conflicts between *Roles*), the same

Collaboration

NamedElement

$interactionInstance : \mathbb{P} \downarrow Interaction\textcircled{S}, binding : \mathbb{P}\, DomainRoleBinding\textcircled{C}$ $actorBinding : \mathbb{P}\, ActorBinding\textcircled{C}$	[Variables]
$\forall ab_1, ab_2 : actorBinding \bullet ab_1.name = ab_2.name \Rightarrow ab_1 = ab_2$	[I1]
$\forall b_1, b_2 : binding \bullet b_1.name = b_2.name \Rightarrow b_1 = b_2$	[I2]
$\#binding \geq 2$	[I3]
$\bigcup\{i : interactionInstance \bullet i.actor\} = \bigcup\{a : actorBinding \bullet a.actor\}$	[I4]
$\#interactionInstance \geq 1$	[I5]
$\forall ab_1, ab_2 : actorBinding \mid ab_1 \neq ab_2 \bullet ab_1.actor.superactor = ab_2.actor.superactor$ $\Rightarrow \text{disjoint}\langle ab_1.instancesBound, ab_2.instancesBound\rangle$	[I6]
$\forall a_1, a_2 : actorBinding \mid a_2.actor \in a_1.actor.subactor \bullet$ $a_2.instancesBound \neq \varnothing \Rightarrow a_2.instancesBound \subset a_1.instancesBound$	[I7]
$\forall b_1, b_2 : binding \mid b_1 \neq b_2 \bullet$ $b_1.roleBinding \in b_2.roleBinding.conflictsWith^+ \lor$ $b_2.roleBinding \in b_1.roleBinding.conflictsWith^+$ $\Rightarrow \{m : b_1.membership \bullet m : agentInstance\} \cap$ $\{m : b_2.membership \bullet m : agentInstance\} = \varnothing$	[I8]
$\forall a_1, a_2 : actorBinding \mid a_1 \neq a_2 \bullet$ $a_1.actor \in b_2.actor.conflictsWith^+ \lor a_2.actor \in b_1.actor.conflictsWith^+$ $\Rightarrow a_1.instancesBound \cap a_2.instancesBound = \varnothing$	[I9]
$\forall b_1, b_2 : binding \mid b_1 \neq b_2 \bullet b_1.roleBinding \neq b_2.roleBinding$	[I10]
$\forall a_1, a_2 : actorBinding \mid a_1 \neq a_2 \bullet a_1.binding \cap a_2.binding = \varnothing$	[I11]
$\forall a : actorBinding \bullet r : a.binding \subseteq binding$	[I12]

Schema 4.4.2: Class schema of *Collaboration*

AgentInstance must not be bound to both *DomainRoles* at the same time. The same holds for *Actors*, which is expressed in Invariant I9.

Moreover, in accordance to Invariant I10, the *DomainRoles* of the *Collaboration's DomainRoleBindings* are disjoint. In other words, a *DomainRole* must not be addressed by two or more *DomainRoleBindings* within the same *Collaboration*. The same holds for *ActorBindings* that must not address the same *DomainRoleBindings* (cf. Invariant I11). Finally, Invariant I12 ensures that any *DomainRoleBinding* addressed by an *ActorBinding* is part of the *Collaboration's DomainRoleBindings* (i.e. expressed through the variable *binding*).

DomainRoles and their bindings are important concepts of a *Collaboration*. The abstract syntax of *DomainRoles* and *Actors* are part of the role viewpoint presented in the next section.

4.5 Role Viewpoint

In accordance to (Dignum and Dignum; 2007), roles identify the activities and services necessary to achieve social objectives and enable to abstract from the specific entity that will eventually perform them. From a society design perspective, roles provide a necessary abstraction for agents in the system, and from the agent design perspective, roles specify the expectations of the society in terms of the agent's behavior in the society.

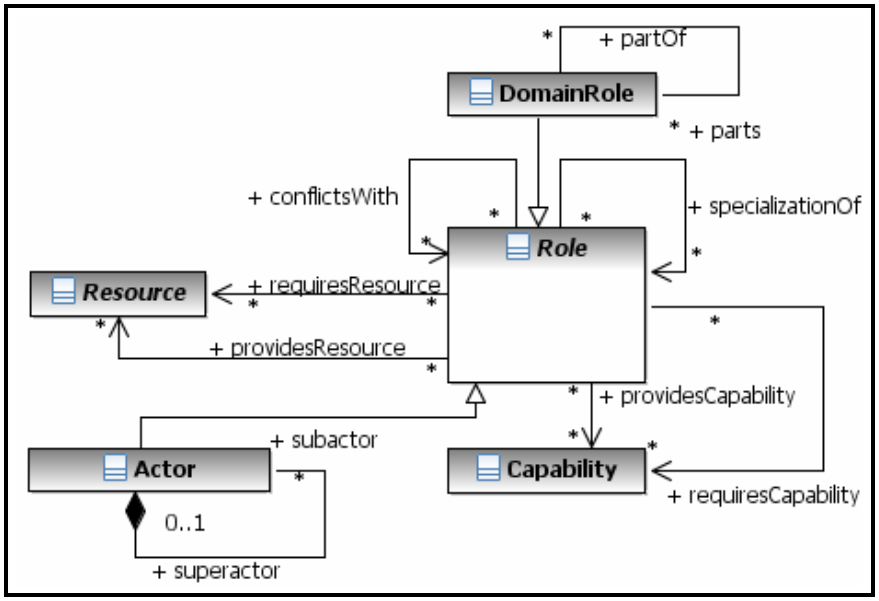


Fig. 4.5: The metamodel reflecting the role viewpoint of PIM4AGENTS.

Roles can be considered as an abstract representation of a functional position of an agent. *Roles* are normally used in the context of social groups of agents like organizations (cf. Section 2.1.5). Correspondingly, in PIM4AGENTS, roles are used in order to (dynamically) position agents within a social context either of the form of an *Organization* or *Interaction*. In particular, the position of an agent within a social context is defined by the (i) characteristics of a role in terms of functionalities, which might be offered or required and (ii) specializations, which need to be distinguished.

The metamodel of the role viewpoint is depicted in Fig. 4.5. The concept of *Role* has two specializations, i.e. *DomainRole* and *Actor*. The former defines the position inside an *Organization*, whereas the latter defines the position of *Agents* (or their run-time representatives) within an *Interaction*. Beside the different kinds of roles, the role viewpoint further includes the concepts of *Capability* (cf. Section 4.3.2 from the agent viewpoint) and Resource (cf. Section A.5.2 from the environment viewpoint).

4.5.1 Role

In PIM4AGENTS, a *Role* is considered as an abstraction of the social and normative behavioral repertoire of the *Agent* or *Organization* in a given social context. It defines what the entities performing this position are expected to do. This is reflected by the particular Behaviors either provided or required and wrapped by *Capabilities*. The abstract syntax of *Role* in PIM4AGENTS is defined as follows:

Definition 4.5.1 (Role in PIM4AGENTS)

A Role *is given by a 7-tuple R = (name, providesResource, requiresResource, providesCapability, requiresCapability, conflictsWith, specializationOf), where:*

- *name: defines the name of the Role*

- *providesResource: depicts the Resources the Role provides to the Agents performing this Role*
- *requiresResource: illustrates the Resources the Role requires from the Agents performing this Role*
- *providesCapability: represents the Capabilities the Role provides to its performing Agent*
- *requiresCapability: displays the Capabilities the Role requires from its performing Agents*
- *conflictsWith: declares that the source Role is in conflict with the target Role, i.e., no AgentInstance is bound to both Roles at the same time*
- *specializationOf: declares that the source Role is a specialization of the target Role with respect to Capabilities and Resources.*

Conforming to Wooldridge (2000b) and Ferber and Gutknecht (1998), a role is normally associated with a set of capabilities defining the functionalities offered in order to fulfill its responsibilities. In PIM4AGENTS, these *Capabilities* are either required or provided, where required means that the *Agents* performing this *Role* must own certain Behaviors matching with the *Role's* requirements. Correspondingly, provided means that the *Role* offers certain functionalities that can be utilized by the *Agents* performing this *Role*. Apart from *Capabilities* representing the functional aspects by specifying how to comply with particular responsibilities, a *Role* also needs to deal with data aspects. This is in PIM4AGENTS mainly covered through the reference to Resources either defining which kinds of Resources are directly provided by the *Role* or required to be provided by the *Agent* performing this *Role*.

In order to define role hierarchies, generalizations between *Roles* can be defined through the *specializationOf* relationship. At this, the source *Role* is a specialized version of the target *Role* (cf. Fig. 4.5). We consider this kind of inheritance as mechanism for incrementally design with PIM4AGENTS. Thereby, new classes may be derived from one or more existing classes. This is particularly significant in the effective reuse of existing specifications. This feature is used later on as part of the endogenous model transformation that generates *DomainRoles* on the base of *Actors*. Apart from the *specializationOf* relationship, under certain conditions, when two *Roles* cannot be taken by an AgentInstance at the same time, we say that these two *Roles* have a conflict, which is expressed through the *conflictsWith* relationship.

Up to now, we mainly dealt with describing the features and properties of a *Role*, without considering the concept of role assignment that relates agent instances to roles in a group (Rupert et al.; 2007). As emphasized by (Odell et al.; 2002), roles can be assigned in at least two ways, either endogenously by self-organization or exogenously by the system designer at design time. Even if the principles of self-organization has been previously investigated (e.g. (Hahn et al.; 2007a, 2006a; Hahn; 2004)). The focus in PIM4AGENTS is clearly on defining patterns of roles that can be imposed at design time and filled at run-time within *Organizations*. The (static) role assignment is done through the RoleBinding concept (cf. Section 4.9.2) of the deployment viewpoint.

The semantics of a *Role* is depicted in Schema 4.5.1. Apart from the primary variables given by Definition 4.5.1, further secondary variables are defined:

- The variable $specializationOf^+$ constitutes the set of *Roles* from which this particular *Role* directly or recursively inherits from (see Invariant I1)
- The variable $conflictsWith^+$ recursively specifies all conflicting *Roles* (cf. Invariant I2)
- The variables $requiresBehavior^+$ and $providesBehavior^+$ define all kinds of Behaviors that are either *Capabilities* required or provided by the *Role* itself, or from any other *Role* this *Role* inherits from (Invariants I3 and I4)
- The variables $providesResource^+$ and $requiresResource^+$ correspondingly illustrate the inherited set of provided and required Resources (cf. Invariants I5 and I6)

Role

NamedElement

$requiresCapability, providesCapability : \mathbb{P}\ Capability$	[Variables]
$requiresResource, providesResource : \mathbb{P} \downarrow Resource;\ conflictsWith, specializationOf : \mathbb{P} \downarrow Role$	
Δ	[Semantic Variables]
$providesResource^+, requiresResource^+ : \mathbb{P} \downarrow Resource$	
$specializationOf^+, conflictsWith^+ : \mathbb{P} \downarrow Role$	
$requiresBehavior^+, providesBehavior^+ : \mathbb{P} \downarrow Behavior$	
$specializationOf^+ == specializationOf \cup \bigcup\{s : specializationOf \bullet s.specializationOf^+\}$	[I1]
$conflictsWith^+ == conflictsWith \cup \bigcup\{s : specializationOf \bullet s.conflictsWith^+\}$	[I2]
$providesBehavior^+ == \bigcup\{p : providesCapability \bullet p.behavior\} \cup \bigcup\{s : specializationOf \bullet s.providesBehavior^+\}$	[I3]
$requiresBehavior^+ == \bigcup\{p : requiresCapability \bullet p.behavior\} \cup \bigcup\{s : specializationOf \bullet s.providesBehavior^+\}$	[I4]
$providesResource^+ == providesResource \cup \bigcup\{s : specializationOf \bullet s.providesResource^+\}$	[I5]
$requiresResource^+ == requiresResource \cup \bigcup\{s : specializationOf \bullet s.requiresResource^+\}$	[I6]
$requiresBehavior^+ \cap providesBehavior^+ = \varnothing$	[I7]
$providesResource^+ \cap requiresResource^+ = \varnothing$	[I8]
$self \notin conflictsWith^+ \land self \notin specializationOf^+$	[I9]
$\forall r : specializationOf^+ \bullet r \notin conflictsWith^+$	[I10]
$\forall r : conflictsWith^+ \bullet r \notin specializationOf^+$	[I11]

Schema 4.5.1: Class Schema of *Role*

Moreover, the denotational semantics state that (i) the Behaviors of $requiresBehavior^+$ and $providesBehavior^+$ must be disjoint (cf. Invariant I7), in the same manner as, (ii) the Resources of $providesResource^+$ and $requiresResource^+$ are disjoint (cf. Invariant I8). A *Role* can neither be part of its overall conflicting *Roles* ($conflictsWith^+$) nor part of its overall generalized *Roles* ($specializationOf^+$) (cf. Invariant I9). If a *Role* is part of the set of generalized *Roles* ($specializationOf^+$), it must not be in any conflict ($conflictsWith^+$) (cf. Invariant I10). Finally, if a *Role* is in any conflict with a *Role* ($conflictsWith^+$), it must not be part of its set of generalized *Roles* ($specializationOf^+$) (cf. Invariant I11).

4.5.2 DomainRole

As aforementioned in Section 4.4.1, a *DomainRole* is, on the one hand, performed by *Agents* and, on the other hand, required by *Organizations*. At this, the concept of *DomainRole* enables grouping of atomic or composed entities to define a social structure and thus clearly define the domain this sort of *Role* has expertise in with respect to the provision of functionalities to achieve a certain objective. An informal definition of *DomainRole* is as follows:

Definition 4.5.2 (DomainRole in PIM4AGENTS)

A DomainRole is defined by a 9-tuple D = (name, requiresResource, providesResource, requiresCapability, providesCapability, conflictsWith, specializationOf, partOf, parts), where:

- *name: defines the name of the DomainRole*
- *partOf: represents the Roles this DomainRole is part of*
- *parts: defines the aggregated set of Roles this DomainRole is composed of.*

The variables requiresResource, providesResource, requiresCapability, providesCapability, conflictsWith, and specializationOf are used in the same manner as specified by Definition 4.5.1.

The formal semantic of *DomainRole* is depicted in Schema 4.5.2. The class schema of *DomainRole* inherits from the *Role* schema. Furthermore, it includes as part of its declarative part the variables *parts* and *partOf* to express the relationship to other *DomainRoles*. The semantics of the *parts* and *partOf* relationships are now defined in terms of the *DomainRoles'* provided and required Behaviors and Resources. In addition, the secondary variables $parts^+$ and $partOf^+$ are defined to illustrate the complete chain of either sub- or super-roles. Their semantics are given by the Invariant I1 and I2.

Invariants I3 and I4 ensure that any kind of sub-role provides or requires a subset of the provided and required Behaviors of its parent roles. Combining all subroles' provided and required Behaviors fulfills the requirements of the parent roles in terms of its provided and required behaviors. The same holds for the required and provided Resources (cf. Invariants I6 and I5). However, none of the sub-roles provide or require all of the provided or required Behaviors and Resources, respectively. Consequently, if a parent *DomainRole* refers to subroles, it must include at least two *DomainRoles*. This circumstance is expressed by Invariant I7.

A *DomainRole* is, moreover, neither part of its $parts^+$ nor $partOf^+$ sets (see Invariant I8). Moreover, Invariant I9 assures that (i) the set of $parts^+$ and $partOf^+$ are disjoint and (ii) a *DomainRole* must not inherit from any *DomainRole* out of $partOf^+$ (i.e. $partOf^+$ and $specializationOf^+$ are disjoint).

Finally, Invariant I10 guarantees that any kind of *Actor*, as specialization of *Role*, is neither part of $parts^+$ nor $partOf^+$ and, in addition, must not be declared as super-role. Hence, even if the concepts of *DomainRole* and *Actors* are both specializations of *Role*, the relationship between both is solely expressed through the *ActorBinding* concept (cf. Section 4.9.4).

4.5.3 Actor

An interaction protocol is a mechanism to illustrate the conversation between agents from a global perspective within a community. It can be more easily described using generic roles instead of describing the interaction between the particular agent instances that take part in the conversation. These generic roles solely act as place holders that are filled with the selected agent instances at run-time. In PIM4AGENTS, we use the concept of an *Actor* as such generic placeholder that defines in which manner the fillers are interacting.

Like a *DomainRole*, the *Actor* concept is a specialization of the *Role* concept (cf. Fig. 4.5.1)) and thus may provide and require particular *Capabilities* or Resources that are necessary for exchanging messages within interactions. The abstract syntax of the *Actor* concept is given in Definition 4.5.3.

Definition 4.5.3 (Actor in PIM4AGENTS)

An Actor is given by a 10-tuple A = (name, requiresResource, providesResource, requiresCapability, providesCapability, conflictsWith, specializationOf, superactor, subactor, activeState), where:

- *name: defines the name of the Role*
- *superactor: represents the super-actor of this Actor*
- *subactor: illustrates all sub-actors of this Actor*
- *activeState: determines the set of MessageFlows in which the Actor is active. The activeState relationship is depicted in Fig. 4.6*

DomainRole

Role

$parts, partOf : \mathbb{P}\, DomainRole$	[Variables]
Δ	[Semantics Variables]
$parts^+, partOf^+ : \mathbb{P}\, DomainRole$	
$parts^+ == parts \cup \bigcup \{p : parts \bullet p.parts^+\}$	[I1]
$partOf^+ == partOf \cup \bigcup \{p : partOf \bullet p.partOf^+\}$	[I2]
$providesBehavior^+ \subseteq \bigcup \{p : parts \bullet p.providesBehavior^+\} \land \forall p : parts \bullet p.providesBehavior^+ \cap providesBehavior^* \neq \varnothing$	[I3]
$requiresBehavior^+ \subseteq \bigcup \{p : parts \bullet p.requiresBehavior^+\} \land \forall p : parts \bullet p.requiresBehavior^+ \cap requiresBehavior^* \neq \varnothing$	[I4]
$providesResource^+ \subseteq \bigcup \{p : parts \bullet p.providesResource^+\} \land \forall p : parts \bullet p.providesResource^+ \cap providesResource^* \neq \varnothing$	[I5]
$requiresResource^+ \subseteq \bigcup \{p : parts \bullet p.requiresResource^+\} \land \forall p : parts \bullet p.requiresResource^+ \cap requiresResource^* \neq \varnothing$	[I6]
$\#parts \geq 2$	[I7]
$self \notin parts^+ \land self \notin partOf^+$	[I8]
$parts^+ \cap partOf^+ = \varnothing \land partOf^+ \cap specializationOf^+ = \varnothing$	[I9]
$\nexists a : Actor \bullet a \in parts \lor a \in partOf \lor a \in specializationOf^+$	[I10]

Schema 4.5.2: Class Schema of *DomainRole*

The variables requiresResource, providesResource, requiresCapability, providesCapability, conflictsWith, and specializationOf are used in the same manner as specified by Definition 4.5.1.

Similar to a *DomainRole* and its *parts* relationship, an *Actor* could be further refined to express that various interaction traces might be feasible within one *Interaction*. Hence, if for an *Actor* more than one trace exists, the system designer has to split the *Actor* and its fillers in a manner that an *Actors'* trace is unique with respect to sending and receiving messages. The formal semantics of this actor sub-actor relationship is discussed in the following in more detail.

The class schema of *Actor* is depicted in Schema 4.7.1. It inherits from the class schema of *Role* and includes three variables, i.e. *subactor*, *superactor*, and *activeState*. In addition, the secondary variable $subactor^+$ is introduced that includes all *Actors* that are kind of this particular type, i.e. it recursively unions the *Actor* itself and all $subactor^+$ of its *subactors* (cf. Invariant I1). Correspondingly, the secondary variable $superactor^+$ recursively unions all *superactors* (cf. Invariant I2). In addition, the variables *messageSent* and *messageReceived* include any *ACLMessage* that is either sent or received by this *Actor*. The *messageSent* variable is defined as the set of *ACLMessage*s that are sent by this *Actor* within any active *MessageFlow* (Invariant I3). Correspondingly, the *messageReceived* variable includes the *ACLMessages* received in active *MessageFlows* (Invariant I4).

Critically, each *Actor* must at most refer to one super-actor (cf. Invariant I5). As the *subactor* reference should be considered as a kind of specialization, each *Actor* must either have no *subactor* or more than one *subactors* (cf. Invariant I6). Furthermore, if an *Actor* has *subactors*, these *subactors* refer again to the *Actor* as *superactor* (see Invariant I7). The eight invariant states that any two *subactors* must have different names. Followed by Invariant I9, further ensuring that an *Actor* is neither part of its own *subactors* (i.e. $subactor^+$) nor part of any $superactor^+$. Finally, Invariant I10 restricts the set of generalized *Roles* to the type *Actor*.

Actor

Role

$subactor : \mathbb{P}\, Actor©;\ superactor : \mathbb{P}\, Actor;\ activeState : \mathbb{P}\, MessageFlow$	[Variables]
Δ	[Semantic Variables]
$subactor^+ : \mathbb{P}_1\, Actor;\ superactor^+ : \mathbb{P}\, Actor;\ messageSent, messageReceived : \mathbb{P}\, ACLMessage$	
$subactor^+ == self \cup \bigcup \{s : subactor \bullet s.subactor^+\}$	[I1]
$superactor^+ == superactor \cup superactor.superactor^+$	[I2]
$messageSent == \bigcup \{m : \bigcup \{ms : \bigcup \{mf : activeState \bullet mf.forkOperator\} \bullet ms.messageSplit^*\} \bullet m.message\}$	[I3]
$messageReceived == \bigcup \{m : \bigcup \{ms : \bigcup \{mf : activeState \bullet mf.joinOperator\} \bullet ms.messageSplit^*\} \bullet m.message\}$	[I4]
$\#superactor \leq 1$	[I5]
$\#subactor = 0 \vee \#subactor \geq 2$	[I6]
$subactor \neq \varnothing \Rightarrow \forall a : subactor \bullet a.superactor = self$	[I7]
$\forall sa_1, sa_2 \in subactor \bullet sa_1 = sa_2 \Rightarrow sa_1.name = sa_2.name$	[I8]
$self \notin \bigcup \{s : subactor \bullet s.subactor^+\} \wedge self \notin superactor^+$	[I9]
$\nexists d : DomainRole \bullet d \in specializationOf^+$	[I10]

Schema 4.5.3: Class Schema of *Actor*

4.6 Interaction Viewpoint

The ability to communicate is one of the core characteristics of agents and group of agents in MASs (cf. Section 2.1.6). Two types of interactions can thereby be distinguished: Protocol-based interactions versus flexible interactions, where PIM4AGENTS emphasizes on the former. The vocabulary used to express interactions in PIM4AGENTS is defined by the metamodel of the interaction viewpoint (cf. Fig. 4.6). This interaction viewpoint includes the concepts *Interaction*, *Protocol*, *MessageFlow*, *MessageScope*, *ACLMessage*, *TimeOut*, *ExchangeMode*, and *Performative*, as well as, *Actor* (from the role viewpoint) and *Message* (part of the multiagent viewpoint).

4.6.1 Interaction

Even if the interaction within a MAS should not necessarily be reduced to the exchange of messages, the least common denominator of protocol-based and flexible interactions is the exchange of messages between two or more entities. This nicely corresponds to the manner in which *Interactions* are generally considered in PIM4AGENTS.

Definition 4.6.1 (Interaction in PIM4AGENTS)

An Interaction *is given by a triple I = (name, message, actor), where:*

- *name: defines the name of the interaction*
- *actor: represents the entities that exchange ACLMessages in this Interaction*
- *message: denotes the ACLMessages exchanged by the corresponding Actors.*

The class schema of *Interaction* is given in Schema 4.6.1. Beside the variables *message* and *actor* brought out by Definition 4.6.1, the following invariants are formalized.

The first category of invariants deals with the issue that within an *Interaction*, both *ACLMessages* as well as *Actors* must be unique (expressed by Invariant I1 and I2). This means that if either two

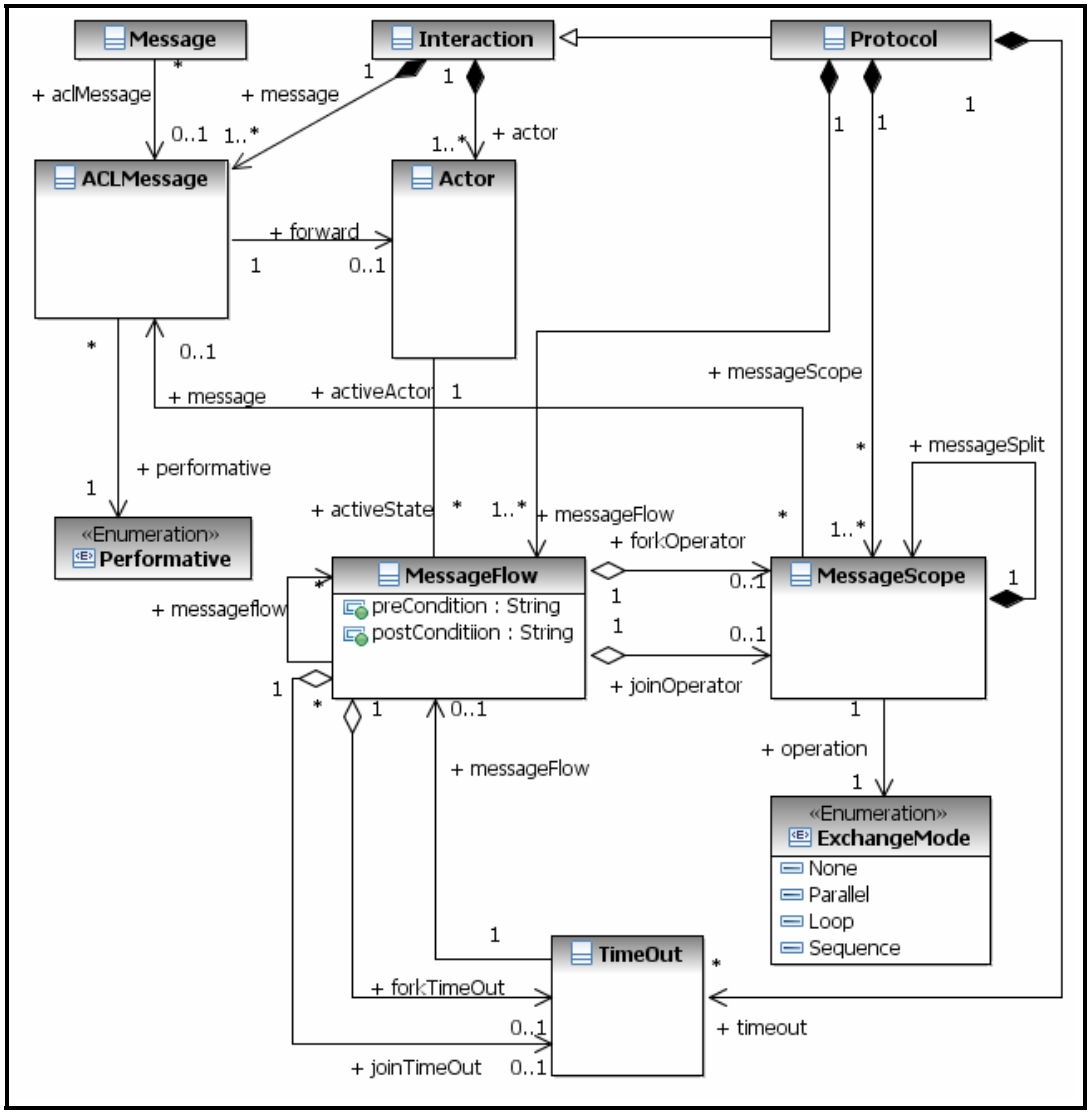


Fig. 4.6: The metamodel reflecting the interaction aspect of PIM4AGENTS.

Actors or two *ACLMessages* are designed having the same name, they are considered as equal. The next category of invariants restricts the kinds of approved interactions. Invariant I3 states that at least two *Actors* must be defined for any kind of interaction, which again must exchange at least one *ACLMessage*. The latter invariant is already expressed through the cardinalities of the *message* containment of the *Interaction* concept. Invariant I4 ensures that any *Interaction* only refers to *Actors* that do not act as sub-actor for any other *Actor*. This is expressed by the *superactor* attribute of an *Actor* (cf. Section 4.5.3). The reasons for restricting *Interactions* in this manner are that (i) sub-actors are considered as a kind of specialization concerning its *Actor's* position within the *Interaction* and (ii) sub-actors are contained by their super-actors. Finally, Invariant I5 ensures that the *forward Actor* of an *ACLMessage* is part of the *Interaction's Actors*.

Interaction

$message : \mathbb{P}_1\, ACLMessage©;\ actor : \mathbb{P}_1\, Actor©$	[Variables]
$\forall m_1, m_2 : message \bullet m_1.name = m_2.name \Rightarrow m_1 = m_2$	[I1]
$\forall a_1, a_2 : actor \bullet a_1.name = a_2.name \Rightarrow a_1 = a_2$	[I2]
$\#actor \geq 2$	[I3]
$\forall a : actor \bullet a.superactor = \varnothing$	[I4]
$\forall m : message \bullet m.forward \subseteq actor$	[I5]

Schema 4.6.1: Class schema of *Interaction*

4.6.2 Protocol

Agent interaction protocols (AIPs) define (i) a communication pattern between several parties as an allowed sequence of messages and (ii) the constraints of the content of these messages to form a conversation of a particular type. Flexible interactions, on the other hand, focus on the relaxation of the constraints that exist in using predefined protocols, potentially leading to more fault-tolerant and hence robust communications, which are driven by the interests of the communication participants and not by predefined sequences of message patterns. The PIM4AGENTS metamodel allows realizing both interaction styles, even if the focus is certainly on AIPs. Leon-Soto et al. (2009), however, demonstrate how to extend PIM4AGENTS to allow a more flexible combination of conversations of protocols.

In accordance with Fig. 4.6, a *Protocol* in PIM4AGENTS refers to (i) a set of *Actors* that interact within the *Protocol*, (ii) a set of *ACLMessages* exchanged by the parties concerned, (iii) a set of TimeOuts that define the time constraints for sending and receiving *ACLMessages*, (iv) a set of *MessageScopes* that define the *ACLMessages* and the order how they arrive, and (v) a set of *MessageFlows* that specify how the exchange of *ACLMessages* is proceed. To express that *AgentInstances* playing the same *Actor* at run-time may behave differently, the *Actor* can be split into sub-actors. At this, sub-actors of an *Actor* are determined at design time, but normally filled with the particular *AgentInstances* performing this position at run-time. The abstract syntax of *Protocol* is given by Definition 4.6.2.

Definition 4.6.2 (Protocol in PIM4AGENTS)

A Protocol is given by a 6-tuple P = (name, actor, messageFlow, messages, messageScope, timeout), where:

- *name: defines the name of the Protocol*
- *actor: refers to Actors that are involved in this Protocol*
- *messageFlow: refers to MessageFlows constituting the different kinds of states the Protocol's Actors are involved in*
- *messages: defines the ACLMessages sent and received within the Protocol*
- *messageScope: depicts the order in which ACLMessages are sent/received within the Protocol*
- *timeout: specifies the latest point in time a particular or set of ACLMessages have to be sent and received.*

The class schema of *Protocol* is given in Schema 4.6.2. Apart from the primary variables introduced by Definition 4.6.2, it further includes the following semantic variables:

Protocol

Interaction

$messageFlow : \mathbb{P}_1\ MessageFlow©;\ messageScope : \mathbb{P}\ MessageScope©;\ timeout : \mathbb{P}\ TimeOut©$ [Variables]

Δ [Semantic Variables]

$usedTimeouts : \mathbb{P}\ TimeOut;\ usedActors : \mathbb{P}\ Actor$

$usedMessages : \mathbb{P}\ ACLMessage;\ usedMessageScopes : \mathbb{P}\ MessageScope$

$usedMessageScopes == \bigcup\{mf : messageFlow \bullet mf.forkOperator.messageSplit^*\}$
$\quad \cup \bigcup\{mf : messageFlow \bullet mf.joinOperator.messageSplit^+\} = messageScope$ [I1]

$usedTimeouts == \bigcup\{ms : usedMessageScopes \bullet ms.forkTimeOut\} \cup$
$\quad \bigcup\{ms : messageScope \bullet ms.joinTimeOut\} = timeout$ [I2]

$usedActors == \bigcup\{mf : messageFlow \bullet mf.activeactor\} = actor$ [I3]

$usedMessages == \bigcup\{m : \cup\{ms : usedMessageScopes \bullet ms.messageSplit^*\} \bullet m.message\} = message$ [I4]

$\forall\ mf_1, mf_2 : messageFlow \bullet mf_1.name = mf_2.name \Rightarrow mf_1 = mf_2$ [I5]

$\forall\ ms_1, ms_2 : messageScope \bullet ms_1.name = ms_2.name \Rightarrow ms_1 = ms_2$ [I6]

$\forall\ to_1, to_2 : timeout \bullet to_1.name = to_2.name \Rightarrow to_1 = to_2$ [I7]

$\bigcup\{m : \bigcup\{mf : messageFlow \bullet mf.forkOperator.messageSplit^*\} \bullet m.message\}$
$\quad = \bigcup\{m : \bigcup\{mf : messageFlow \bullet mf.joinOperator.messageSplit^*\} \bullet m.message\}$ [I8]

$\forall\ mf_1, mf_2 : messageFlow \mid$
$\quad \bigcup\{m : mf_1.forkOperator.messageSplit^* \bullet m.message\} \cap$
$\quad\quad \bigcup\{m : mf_2.joinOperator.messageSplit^* \bullet m.message\} \neq \varnothing$
$\quad\quad \bullet\ mf_1.activeActor \neq mf_2.activeActor$ [I9]

Schema 4.6.2: Class schema of *Protocol*

- The semantic variable *usedTimeouts* represents TimeOuts that are referred by a MessageScope through the variables *joinTimeOut* and *forkTimeOut*.
- The semantic variable *usedActors* depicts *Actors* that are referred by a *MessageFlow* through the variable *activeActor*.
- The semantic variable *usedMessages* denotes *ACLMessages* that are referred by a MessageScope through the variable *message*.
- The semantic variable *usedMessageScopes* describes MessageScopes that are referred by a *MessageFlow* through the variables *forkOperator* or *joinOperator*.

Beside these secondary variables, the following invariants refined the semantics of *Protocol*:

- Invariant I1 specifies that *usedMessageScopes* is equal to *messageScope* in order to ensure that any MessageScope is actively used within the *Protocol*.
- The set of *usedTimeouts* must consist of the same elements as *timeouts* (i.e. the TimeOuts referred by the *Protocol*) expressed by Invariant I2. Consequently, this means that all *Protocol's* TimeOuts must be actively used within the *Protocol*.
- Like for the semantic variable *usedTimeouts*, Invariant 3 restricts the elements of *usedActors* by stating that the set of *Actors* referred by the *activeActor* variable of *MessageFlow* must be equal to the *Actors* defined by the *Protocol* (i.e. the *actor* variable of *Interaction* from which the *Protocol* inherits).
- In the same way, Invariant I4 specifies that the *usedMessages* variable contains the same *ACLMessages* as depicted by the *message* variable of *Interaction* from which *Protocol* concepts inherits. This means that any *ACLMessage* not defined by the *Interaction* cannot be used in this particular *Interaction*.

Additionally, Invariant I5 specifies that two instances of a *MessageFlow* within a *Protocol* must have different names, otherwise they are considered as equal. Invariants I6 and I7 state that in the context of a *Protocol* this unique occurrence similarly holds for the set of *MessageScopes* (i.e. *messageScope*) and *TimeOuts* (i.e. *timeout*). Invariant I8, moreover, ensures that any *ACLMessage* sent through a *MessageFlow's forkOperator* is also received through a *MessageFlow's joinOperator*. Finally, Invariant I9 ensures that an *ACLMessage* cannot be sent and received by one and the same *Actor*.

4.6.3 MessageFlow

A *MessageFlow* in PIM4AGENTS defines the sequence in which *ACLMessages* are sent and received by the *Actors* involved in a *Protocol*. It further defines time constraints (i.e. the latest point in time) in which *ACLMessages* must be sent and received. The abstract syntax of a *MessageFlow* is defined as follows:

Definition 4.6.3 (MessageFlow in PIM4AGENTS)

A MessageFlow is given by a 9-tuple M = (name, activeActor, forkOperator, joinOperator, forkTimeOut, joinTimeOut, messageflow, preCondition, postCondition), where:

- *name: defines the name of the MessageFlow*
- *activeActor: represents the Actor sending/receiving ACLMessages in this MessageFlow*
- *forkOperator: defines the order of incoming ACLMessages*
- *joinOperator: defines the order of outgoing ACLMessages*
- *forkTimeOut: specifies the time constraints for receiving ACLMessages*
- *joinTimeOut: specifies the time constraints for sending ACLMessages*
- *messageflow: illustrates state transitions, i.e. transitions between two MessageFlows of the same Actor*
- *preCondition declares the Protocol's state before MessageFlow execution*
- *postCondition declares the Protocol's state after MessageFlow execution.*

A *MessageFlow* can be considered as a state within a *Protocol*. If this state fulfills a certain precondition, ACLMessages are either sent and/or received by the *Actors* active in this state. At this, the *MessageFlow's fork-* and *joinOperators* define the state transitions. The formal specification of *MessageFlow* is given in Schema 4.6.3. In accordance to the abstract syntax given in Definition 4.6.3, the declarative part includes the variables *joinOperator*, *forkOperator*, *joinTimeOut*, *forkTimeOut*, *activeActor*, *messageflow*, *preCondition*, and *postCondition*.

Beside the declarative part, several invariants are defined: Invariant I1 states that for any *MessageFlow* at most one *forkOperator* and *joinOperator* must be defined. However, either *forkOperator* or *joinOperator* must be defined (see Invariant I2). A *MessageFlow* without any *joinOperator* can be considered as start state, without *forkOperator* as end state, where a *Protocol* could own more than one start and end state. Like in the case of the MessageScope attributes, Invariant I3 ensures that for each *MessageFlow* at most one *forkTimeOut* and one *joinTimeOut* is permitted, however, other than *forkOperator* and *joinOperator*, a *MessageFlow* does not need to refer to any TimeOut. Lastly, in accordance to Invariant I4, a direct transition between *MessageFlows* expressed through the *messageflow* reference is only allowed if the *activeActors* of the target *MessageFlow* are of the same kind as the *activeActor* of the source *MessageFlow*.

MessageFlow

NamedElement

$joinOperator, forkOperator : \mathbb{P}\ MessageScope$ $joinTimeOut, forkTimeOut : \mathbb{P}\ TimeOut;\ activeActor : Actor$ $messageflow : \mathbb{P}\ MessageFlow;\ preCondition, postCondition : \mathbb{B}$	[Variables]
$\#forkOperator \leq 1 \land \#joinOperator \leq 1$	[I1]
$forkOperator \neq \varnothing \lor joinOperator \neq \varnothing$	[I2]
$\#forkTimeOut \leq 1 \land \#joinTimeOut \leq 1$	[I3]
$messageflow \neq \varnothing \Rightarrow \{m : messageflow \bullet m.activeActor\} \subseteq activeActor.subactor^{+} \cup activeActor.superactor^{+}$	[I4]

Schema 4.6.3: Class schema of *MessageFlow*

4.6.4 MessageScope

The previous section formalized the concept *MessageFlow*, which represents the *Protocol's* states. To enter and leave a *MessageFlow*, special kinds of transitions are offered that are called MessageScopes. These *MessageScopes* define the order in which *ACLMessages* are exchanged. For this purpose, *ACLMessages* are connected via an *operation* defining in which manner (e.g. parallel, sequence, etc.) the *ACLMessages* are sent or received, respectively. The abstract syntax is given by Definition 4.6.4:

Definition 4.6.4 (MessageScope in PIM4AGENTS)

A MessageScope is given by a 4-tuple M = (name, message, operation, messageSplit), where:

- *name: defines the name of the MessageScope*
- *message: refers to ACLMessages that are sent and received in the current state of the Protocol*
- *operation: defines in which manner ACLMessages are sent and received*
- *messageSplit: allows defining compositions of operations*

MessageScope

NamedElement

$messageSplit : \mathbb{P}\ MessageScope©;\ operation : ExchangeMode$ $message : \mathbb{P}\ ACLMessage;\ break : \mathbb{P}\ Break©$	[Variables]
Δ $messageSplit^{+} : \mathbb{P}\ MessageScope;\ messageSplit^{*} : \mathbb{P}_1\ MessageScope$	[Semantic Variables]
$messageSplit^{+} == messageSplit \cup \{m : messageSplit, n : MessageScope \mid n \in m.messageSplit^{+} \bullet n\}$	[I1]
$messageSplit^{*} == \{self\} \cup messageSplit^{+}$	[I2]
$\forall ms_1, ms_2 : messageSplit \bullet ms_1.name = ms_2.name \Rightarrow ms_1 = ms_2$	[I3]
$\forall b_1, b_2 : break \bullet b_1.name = b_2.name \Rightarrow b_1 = b_2$	[I4]
$operation = None \Rightarrow messageSplit = \varnothing \land \#message = 1$	[I5]
$operation \neq None \Rightarrow messageSplit \neq \varnothing \land \#message = 0$	[I6]

Schema 4.6.4: Class schema of *MessageScope*

The class schema of a *MessageScope* is given in Schema 4.6.4. It includes the variables *messageSplit*, *message*, and *operation* which corresponds to literals specified by the enumeration *ExchangeMode*. The different literals of *ExchangeMode* are discussed in more detail in Section 4.6.5.

Apart from these primary, further semantic variables are given: The variable $messageSplit^+$ defines the transitive closure of all *messageSplits* the *MessageScope* recursively refers to (cf. Invariant I1). Accordingly, the variable $messageSplit^*$ unions the $messageSplit^+$ variable and the *MessageScope* itself (cf. Invariant I2). Any *MessageScope* referred to by the *messageSplit* variable or *Break* must be unique (cf. Invariant I3-I4). Moreover, Invariants I5 and I6 ensure that any *ACLMessage* must only be referred to by a *MessageScope* of operation type *None*. In this case, the *MessageScope's messageSplit* is empty. Otherwise, the *MessageScope* must not refer to any *ACLMessage*, but the *messageSplit* variable contains other branching *MessageScopes*.

4.6.5 ExchangeMode

The enumeration ExchangeMode represents the different manners of trace execution. For ordering messages within *Protocols*, PIM4AGENTS distinguishes between the following alternatives:

Sequence prescribes a sequencing of traces, i.e., all ACLMessages within this *Interaction* are timely ordered.

Parallel prescribes that several traces are executed concurrently within this *Interaction*, i.e., the order in which the corresponding ACLMessages are displayed is not relevant.

Loop prescribes that a particular trace is executed again and again. How often this trace is finally executed depends on either the Break condition or the *MessageFlow's* postcondition, which are both boolean expressions. The Loop is executed as long as these guards are satisfied.

Apart from these general literals, the pattern of an exclusive decision (i.e. XOR semantics) in *Protocols* is supported through the *messageflow* attribute of *MessageFlow* that allows distinguishing between exclusive states of an *Actor*.

4.6.6 ACLMessage

As aforementioned in Section 4.2.2, PIM4AGENTS distinguishes between two sorts of messages, i.e. *Message* and *ACLMessage*. The former defines the kind of information that is exchanged between two *AgentInstances* and the latter specifies the message exchange between *Actors* in *Interactions*. Both concepts are linked to each other as an *ACLMessage* is realized by sending and receiving *Messages* within a *Plan*. The abstract syntax of *ACLMessage* is indicated in Definition 4.6.5.

Definition 4.6.5 (ACLMessage in PIM4AGENTS)

An ACLMessage is given by a triple A = (name, forward, performative), where:

- *name: defines the name of the ACLMessage*
- *forward: represents the Actor to whom the ACLMessage is forwarded*
- *performative: defines the Performative of the ACLMessage.*

The formal semantics of ACLMessage is given in Schema 4.6.6. Beside the primary variables (cf. Definition 4.6.5), the Invariant I1 ensures that the *forward Actor* does not receive this ACLMessage through a *MessageFlow* and its MessageScopes.

ACLMessage

NamedElement

$forward : \mathbb{P}\, Actor$; $performative : Performative$ [Variables]

disjoint$\langle forward, \{mf : MessageFlow, me : \bigcup \{mes : mf.joinOperator.messageSplit^* \bullet mes.message\}$
$\mid self \in me \bullet mf.activeActor\}\rangle$ [I1]

Schema 4.6.5: Class schema of *ACLMessage*

As previously presented in Section 4.2.2, two core messaging standards are accepted in the agent community to specify certain performatives and parameters used inside messages. The original FIPA-ACL performatives are accept-proposal, agree, cancel, cfp, confirm, disconfirm, failure, inform, inform-if, inform-ref, not-understood, propose, query-if, query-ref, refuse, reject-proposal, request, request-when, request-whenever, and subscribe. A subset of them is implemented in PIM4AGENTS through the Performative enumeration:

Performative ::= *Request* | *Failure* | *Cancel* | *Agree* | *NotUnderstood* | *CFP* | *Refuse* | *Propose*
| *AcceptProposal* | *RejectProposal* | *Inform*

In addition to the concepts discussed so far, the interaction viewpoint further incorporates the concepts of Break and TimeOut. A more detailed description of the concepts' meanings as well as their formal semantics can be found at Appendix A.

The concepts of the interaction viewpoint mainly serve for designing the interaction between entities (i.e. *Actors*) of the MAS. Hence, the interaction viewpoint mainly centers on the global behavior, internal processes of the agent can only be defined on an abstract level in terms of sending and receiving messages. Hence, a more expressive vocabulary for designing an agent's internal processes is required and given by the behavioral viewpoint.

4.7 Behavior Viewpoint

As suggested by Definition 2.1.5, behavioral elements are of particular importance to allow agents to behave in an autonomous, reactive, pro-active and social manner. To meet these requirements, PIM4AGENTS combines different behavioral elements in the behavioral viewpoint.

The behavioral viewpoint can be divided into two parts: the first and core part describes how behaviors and plans are structured, whereas the second part focuses on the concrete concepts (either of complex or atomic form) provided to define the workflow of a certain behavior. The core part (depicted in Fig. 4.7) covers the main aspects of internal processes that are *Behavior*, *Plan*, *ControlFlow*, *InformationFlow*, *Activity*, *StructuredActivity*, *Task* and *Knowledge*.

4.7.1 Behavior

A Behavior is an abstract class that defines the internal behavior of an *Agent* to achieve. The body of a *Behavior* defines the order in which certain activities are executed to achieve an *Agent's* overall

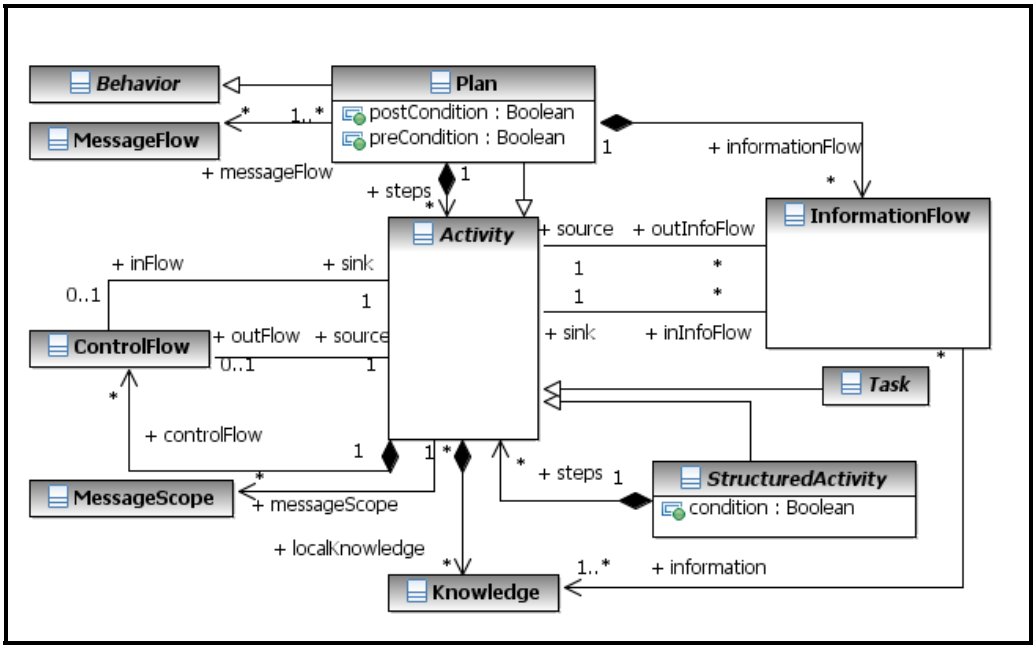


Fig. 4.7: The core metamodel of the behavioral viewpoint of PIM4AGENTS.

goal. A *Plan* is a concrete refinement of *Behavior*. The Object-Z specification of *Behavior* is given in Section A.4.1.

4.7.2 Plan

A plan, in accordance to (d'Inverno and Luck; 2001a), consists of six components: (i) an invocation condition (or triggering event), (ii) an optional context (a situation formula) that defines the pre-conditions of the plan, i.e., what must be believed by the agent for a plan to be executable, (iii) the plan body, which is a tree representing a kind of flow-graph of actions to perform, (iv) a maintenance condition that must be true for the plan to continue executing, (v) a set of internal actions that are performed if the plan succeeds and finally, (vi) a set of internal actions that are performed if the plan fails.

A *Plan* in PIM4AGENTS is considered as mechanism to specify an *Agent's* internal processes. It represents a super class connecting the agent viewpoint with the behavioral viewpoint. *Plans* are executed by *Agents* in order to achieve certain predefined goals. Even if the modeling of goals is not directly supported by the current version of PIM4AGENTS, indirectly, a Plan supports achieving goals through performing *Activities*. For the purpose of combining these *Activities*, a workflow-like language is provided.

In a broad sense, workflow specifications can normally be understood from a number of different perspectives: The control flow perspective or process perspective describes activities and their execution ordering through representing the flow of execution. *Activities* in elementary form are atomic units of work, and in more complex form modularize an execution order of a set of activities. The data perspective defines how data is processed on the control flow perspective.

For this purpose, information flows between activities and local variables of the workflow. The operational perspective describes the elementary actions executed by activities.

In the style of workflow specifications, a *Plan* in PIM4AGENTS includes a set of *Activities* that define the basic actions, *ControlFlows* that define the order in which *Activities* are executed and *InformationFlows* defining the manner in which information flows between *Activities*. The abstract syntax of the *Plan* concept is given by Definition 4.7.1.

Definition 4.7.1 (Plan in PIM4AGENTS)

A Plan is given by a 11-tuple P = (name, steps, controlFlow, messageFlow, preCondition, postCondition, outFlow, inFlow, localKnowledge, messageScope, informationFlow), where:

- *name: defines the name of the Plan*
- *steps: defines all associated basic or more complex Activities used for achieving goals*
- *controlFlow: represents the set of ControlFlows connecting the Plan's Activities*
- *messageFlow: denotes the MessageFlows implemented by the Plan*
- *preCondition: represents the state of the Plan in terms of facts holding before execution*
- *postCondition: represents the state of the Plan in terms of facts holding after execution*
- *outFlow: specifies the ControlFlow that links this source Plan with a target Activity*
- *inFlow: specifies the ControlFlow that links a source Activity with this target Plan*
- *localKnowledge: depicts all Knowledges that are global accessible in the Plan*
- *messageScope: depicts the MessageScopes implemented by this Plan*
- *informationFlow: depicts how data flows between Plans.*

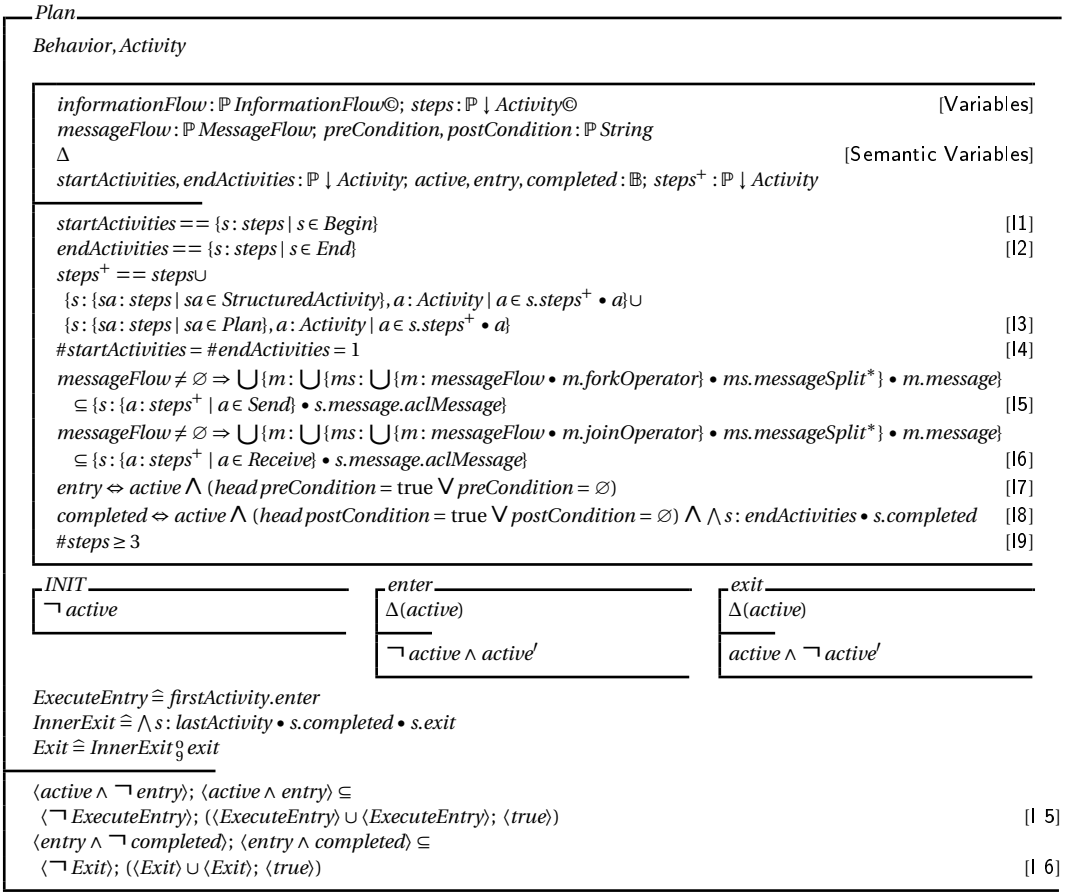
A *Plan* specifies an *Agents'* internal process in terms of implicit goals that need to be achieved. For this reason, it refers to a set of *ControlFlows* and *InformationFlows* part of the process description and contains a set of *Activities* that are linked to each other via a *ControlFlow* to specify the execution order. As a *Plan* is again a specialization of *Activity* (cf. Section 4.7.5), it may comprise sub-plans that are connected with either other sub-plans or *Activities* through the *outFlow* and *inFlow* references. Beside the *Activities* owned by a *Plan*, it may also contain *Knowledge*, which can only be accessed within its scope and is exchanged through *InformationFlows* and *Messages*.

To provide a link between an external *Protocol* and internal *Plan* that is implementing the interaction specified, a *Plan* refers to a set of *MessageFlows* (cf. Section 4.6.3). Having this link supports checking whether the *Plan* provides all functionalities needed to fulfill the requirements of the particular *MessageFlow* in terms of receiving and sending *Messages*. Finally, the variables *preCondition* and *postCondition* set the constraints that should be fulfilled before and after execution, respectively.

The semantics of a *Plan* are given in Schema 4.7.1. The corresponding class schema inherits from the class schemata of *Behavior* (cf. Schema A.4.1) and *Activity* (cf. Schema 4.7.5). Its declarative part consists of the primary variables declared in Definition 4.7.1.

In addition, the following semantic variables are defined used for describing the operational semantics: These variables are *startActivities*, *endActivities* of either type *Begin* or *End* (cf. Invariant I1 and I2 for a definition), *active*, *entry* of the type boolean, as well as $steps^+$ of type *Activity* denoting the set of all direct (i.e. *steps*) and indirect (i.e. any *step* part of the Plan's *steps* etc.) contained *Activity*. Hence, the $steps^+$ variable defines the transitive closure of the *step* relation expressed in Invariant I3.

In addition, the declarative part consists of several invariants: Firstly, a *Plan* has in accordance to Invariant I4 exactly one start and end activity (i.e., the set of *startActivities* and *endActivities*



Schema 4.7.1: Class schema of *Plan*

has exactly one element). If a *messageFlow* reference is defined, the *ACLMessage* sent and/or received by the corresponding *MessageFlow* must be sent and received through the *Send* and *Receive* activities, respectively (cf. Invariants I5 and I6). At this, the *Messages* referred within a *Plan* indicate the *ACLMessages* of the *MessageFlow* (cf. Section 4.2.2).

The operational semantics of a *Plan* are: When a *Plan* is entered, it becomes *active* meaning that the operation *enter* changes the variable *active* from false to true. When a *Plan* is exited, it becomes inactive meaning that the operation *exit* changes the variable *active* from true to false. Initially, the variable *active* is set to false (see operation *INIT*). Furthermore, a *Plan* is *entry* if it is *active* as well as the *preCondition* either evaluates to true or is not defined (cf. Invariant I7) and a *Plan* is *completed* if it is *active*, the ending *Activities* (i.e. *endActivities*) of the contained *Activities* (i.e. *steps*) are completed, and the *postCondition* either evaluates to true or is not defined (cf. Invariant I8).

Finally, at least three *Activities* must be part of a *Plan*, i.e. the *startActivities*, the *endActivities* (both restricted to 1), and an *Activity* expressing the Plan's functionality.

Additionally, we define three operations *ExecuteEntry*, *InnerExit*, and *Exit* which are used by the ControlFlow transitions when they enter or exit an *Activity* (see the Object-Z class ControlFlow). Since executing a *Plan* should start with executing the *startActivities*), the operation *ExecuteEntry* invokes the operation *enter* of the first activity within the *Plan*. This guarantees that the containing *Activities* of a *Plan* are always *active* when the *Plan* itself is *active*. The *Exit* operation simply exits the *Plan* by invoking the operations *InnerExit* and *exit*.

Finally, we define the operational sequence in terms of invariants using the timed trace notation (see (Smith and Hayes; 2000)). The invariants are described in the following notation: $\langle \neg\, var \rangle$; $\langle var \rangle \subseteq \langle \neg\, op \rangle$; $(\langle op \rangle \cup \langle op \rangle$; $\langle true \rangle)$. This term has the semantic that the operation *op* occurs immediately when the variable *var* evaluates to true. In the *Plan* context, the invariant defined by Invariant I5 states the operation sequence when a *Plan* is entered. The invariant ensures that the operation *ExecuteEntry* occurs immediately when the *Plan* is *active* and has been entered i.e. *entry* evaluates to true). In this case, the first activity of *steps* is executed. The invariant defined by Invariant I6 states the operation sequence when a *Plan* is exited. The invariant ensures that a *Plan* is only exited if the work on it has been completed. If this is the case, the operation *Exit* is immediately invoked.

4.7.3 ControlFlow

Two different sorts of flows are distinguished in the behavior viewpoint: *InformationFlow* and *ControlFlow*. The latter refers to a source and sink *Activity* and defines the temporal execution dependencies between *Activities*, i.e. the *ControlFlow* transition specifies the exact order of *Activities* to be executed. We assume transitivity of *ControlFlow* relations, but not symmetry and reflexivity. The former, in contrast, defines how information flow from *Activity* to *Activity* through their *Knowledges* (cf. Section 4.7.4). The following definition illustrates the informal specification of ControlFlow.

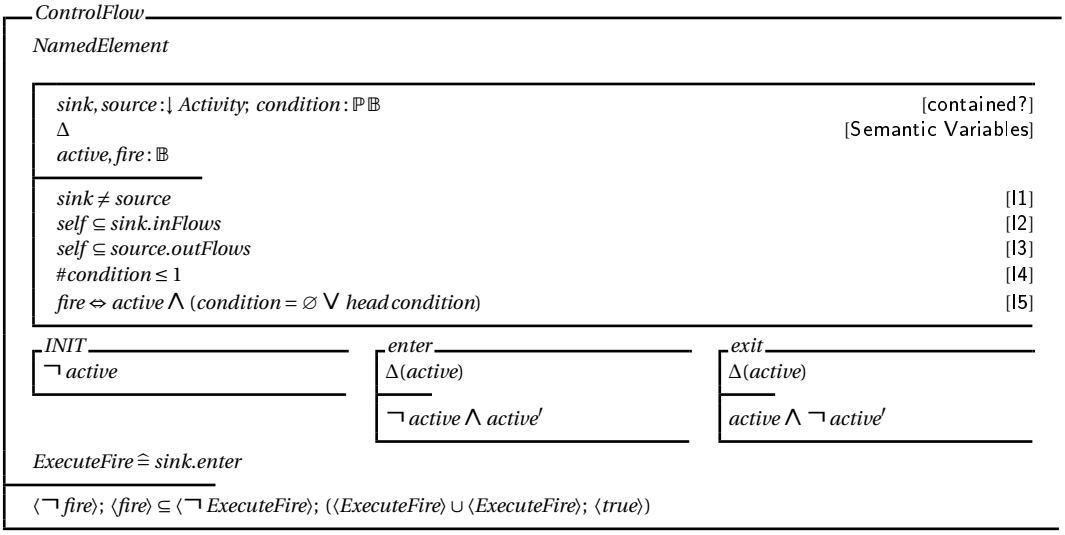
Definition 4.7.2 (ControlFlow in PIM4AGENTS)

A ControlFlow is given by a 4-tuple C = (name, sink, source, condition), where:

- *name: defines the name of the ControlFlow*
- *source: specifies the source Activity of this ControlFlow*
- *sink: refers to the sink Activity of this ControlFlow*
- *condition: defines the boolean expression denoting whether the control is transferred from the source to the sink Activity*

The semantics of *ControlFlow* is given in Schema 4.7.3. The class schema inherits from the class schema of *NamedElement* (cf. Schema A.1.1) and additionally consists of the variables *source*, *sink* and *condition*). The *sink* and *source* variables represent the *Activities* that are linked to each other through the *ControlFlow*, where both *Activities* must be unequal (see Invariant I1). A *ControlFlow* itself is again part of the *inFlow* or *outFlow* of the particular *source* and *sink Activity*. This is expressed by the Invariants I2 and I3 of the corresponding class schema.

The *condition* is defined as a set of boolean expressions denoting in which case the control is transfered from *source* to *sink*. Beside these primary variables, the semantic is further refined by the following invariants. Invariant I4, for instance, states that the set of *conditions* is restricted to one element, meaning that either a *condition* is not defined (which means in this context that the condition is true) or exactly one *condition* is given, which could be a complex logical expression.



Schema 4.7.2: Class schema of *ControlFlow*

Apart from these two primary variables, furthermore, two semantic variables of boolean type are introduced: (i.e. *active*, and *fire*), where *fire* evaluates to true iff both, *active* and *condition*, evaluate to true (see Invariant I5). In addition, four operations (i.e. *enter*, *exit*, *ExecuteExit*, and *ExecuteFire*) are defined. When a *ControlFlow* is entered, it becomes *active* meaning that the operation *enter* changes the value of *active* from false to true. When a *ControlFlow* is existed it becomes inactive meaning that the operation *exit* changes the variable *active* from true to false. The operation *ExecuteExit* guarantees that the source *Activity* is exited, whereas the operation *ExecuteFire* guarantees that the sink *Activity* is entered.

The operation sequences of the *ControlFlow* are defined as follows: Invariant I2 states the operation sequence when a *ControlFlow* is exited. The invariant ensures that the source *Activity* is exited immediately after the variable *active* evaluates to false. Invariant I3, additionally, ensures that the sink *Activity* is entered immediately after the variable *fire* evaluates to true.

4.7.4 InformationFlow

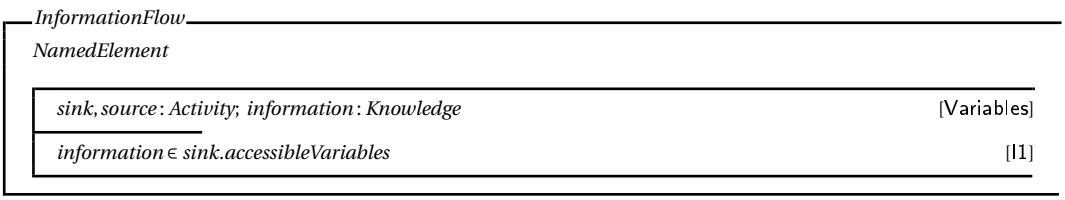
Unlike a *ControlFlow*, an *InformationFlow* defines how information is exchanged among the *Activities* in a *Plan*. Thereby, it specifies a unidirectional communication path between exactly two *Activities* to define how information and data is flowing between them. The information is expressed as *Knowledge* (cf. Section 4.3.3), the *Agent* has access to inside the *Plan*. The abstract syntax of the concept *InformationFlow* is given in Definition 4.7.3.

Definition 4.7.3 (InformationFlow in PIM4AGENTS)

An InformationFlow is given by a 4-tuple I = (name, source, sink, variable), where:

- *name: defines the name of the InformationFlow*
- *source: specifies the source Activity of this InformationFlow*
- *sink: specifies the sink Activity of this InformationFlow*

- *information: represents the information passed from the source Activity to the sink Activity*



Schema 4.7.3: Class schema of *InformationFlow*

An *InformationFlow* specifies through the *information* property the data transfer between two *Activities*. The class schema of *InformationFlow* is given in Schema 4.7.4. Invariant I1 states that any *information* transmitted through an *InformationFlow* must be part of the sink activity's scope of *Knowledge* expressed through the secondary variable *accessibleVariables* defined in Schema 4.7.5.

4.7.5 Activity

An *Activity* is an abstract class that defines the most common state within the behavioral aspect. Any two *Activities* may be connected through a *ControlFow* and/or *InformationFlow*. Furthermore, we distinguish between two sorts of *Activities*: *StructuredActivities* (cf. Section 4.7.6) define more complex control structures, whereas *Tasks* (cf. Section 4.7.13) are mainly used to define atomic activities. Definition 4.7.4 summarizes the abstract syntax of *Activity*.

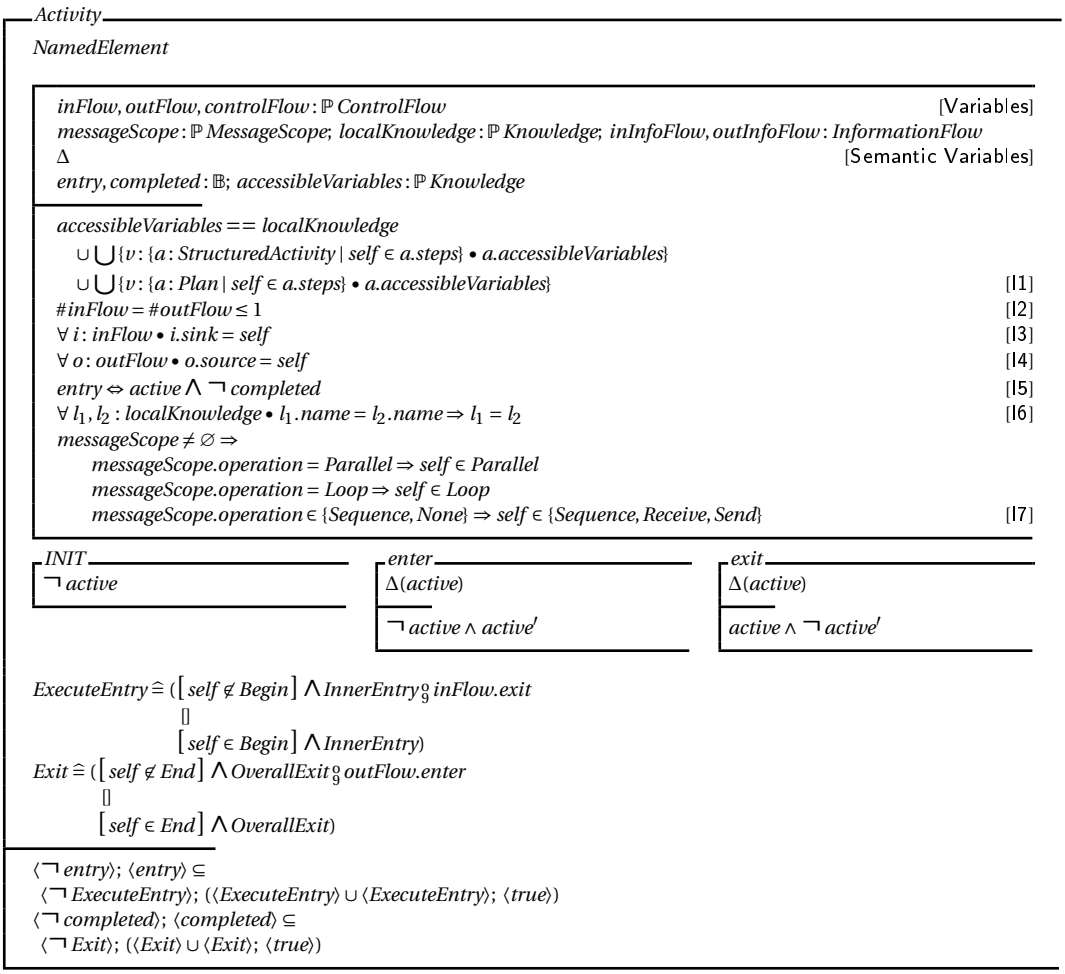
Definition 4.7.4 (Activity in PIM4AGENTS)

An Activity is given by a 6-tuple A = (name, controlFlow, outFlow, inFlow, localKnowledge, inInfoFlow, outInfoFlow, messageScope), where:

- *name: denotes the name of the Activity*
- *controlFlow: depicts the ControlFlows used inside the Activity*
- *outFlow: stands for the set of Flows that exits this Activity*
- *inFlow: stands for the set of Flows that enters this Activity*
- *localKnowledge: represents any kinds of Knowledge that can be accessed in the scope of this Activity*
- *inInfoFlow: represents the ingoing InformationFlows*
- *outInfoFlow: represents the outgoing InformationFlows*
- *messageScope: depicts any kind of MessageScope that is implemented by this Activity*

The formal semantics of *Activity* is given in Schema 4.7.5. Apart from the primary variables introduced in Definition 4.7.4, the declarative part of the schema additionally includes the three semantic variables, *entry*, *completed*, and *accessibleVariables*. The set of *accessibleVariables* is defined as the union of any *localKnowledge* of the Activity itself and the *accessibleVariables* of any Plan or *StructuredActivity* that recursively contains this *Activity* (cf. Invariant I1).

In addition, we restrict the upper bound of *ControlFlows* that can be defined as *inFlow* or *outFlow* of an *Activity* to one (cf. Invariant I2). In accordance to Invariant I3 and I4, the source



Schema 4.7.4: Class schema of *Activity*

and sink *Activities* of the ingoing and outgoing *ControlFlows* is the *Activity* itself. The semantics variable *entry* is defined as the conjunction of the variable *active* and the negation of *completed* (see Invariant I5). Invariant I6 states that any *Knowledge* of an *Activity* must be different, otherwise it is considered as equal. Finally, the Invariants I7 to I9 restricts the link between *MessageScope* and *Activity* in that manner that if the *operation* of a *MessageScope* is *ExchangeMode:Parallel*, the *Activity* must be of the form *Parallel*. If the *operation* is of type *ExchangeMode:Loop*, the *Activity* is of type Loop, etc.

The operational semantics of an *Activity* is defined as follows: When an *Activity* is entered, it becomes *active* meaning that the operation *enter* changes the variable *active* from false to true. When an *Activity* is exited, it becomes inactive meaning that the operation *exit* changes the variable *active* from true to false. Initially, the variables *active*, *activityDone*, and *innerComplete* are set to false (see operation *INIT*).

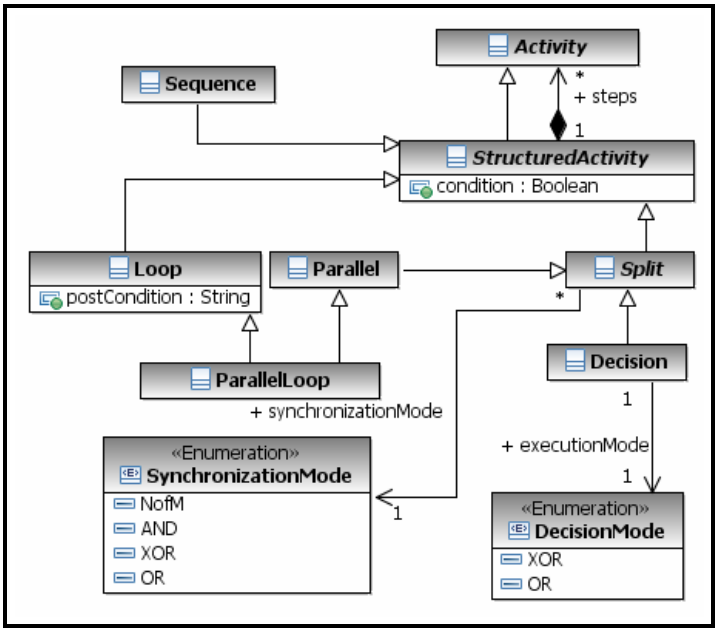


Fig. 4.8: The specializations of a StructuredActivity

Moreover, we define the operations *ExecuteEntry*, *Exit*, and *OverallExit*, where the operation *ExecuteEntry* invokes the *InnerEntry* operation possibly followed by invoking the *exit* operation of the *inFlow*. Certainly, the *inFlow* is only aborted if the *Activity* has an ingoing *ControFlow*. The operation *InnerEntry* is defined in the specializations of the concept *Activity*. The operation *Exit* invokes the operation *OverallExit* followed by possibly invoking the next *ControlFlow* defined by the *outFlow* variable. The operation *OverallExist* invokes again the operation *InnerExit* that is again refined in the different specializations of an Activity (see for instance Section 4.7.13).

Finally, we define the operational sequence in terms of invariants using the timed trace notation. Invariant I5 defines the operation sequence when an *Activity* is entered. The invariant ensures that the operation *ExecuteEntry* occurs immediately when the *Activity* is *entry*. The semantics defined by Invariant I6 refine the operation sequence when an *Activity* is exited. The invariant ensures that an *Activity* is only exited if the work on the *Activity* has been completed. If this is the case, the operation *Exit* is immediately invoked.

Section 4.7, up to now, mainly dealt with the basic functionalities of *Plans* used to describe the internal behavior of *Agents*. Summing up, a *Plan* is composed of Activities that are linked through *ControlFlows* and *InformationFlows*. As aforementioned, we distinguish between two sorts of *Activities*, i.e. *StructuredActivities* and *Tasks*.

In the remaining section, we focus on the concepts of *StructuredActivity* and *Task* and their specializations. The metamodel of the behavioral viewpoint focusing on complex activities is depicted in Fig. 4.8. It includes the concepts *StructuredActivity*, *Sequence*, *Split*, *Loop*, *Parallel*, *Decision*, and *ParallelLoop*, as well as *Activity*. Furthermore, this part of the behavioral metamodel consists of the enumerations *DecisionMode* and *SynchronizationMode*.

4.7.6 StructuredActivity

A *StructuredActivity* is an abstract class that introduces more complex control structures into the behavioral viewpoint. It inherits from *Activity*, but additionally, includes a set of *Activities*. Thus, the *StructuredActivity* allows structuring *Activities* recursively. The abstract syntax of *StructuredActivity* is as follows:

Definition 4.7.5 (StructuredActivity in PIM4AGENTS)

A StructuredActivity is given by a 8-tuple S = (name, steps, controlFlow, condition, localKnowledge, inFlow, outFlow, messageScope), where:

- *name: defines the name of the StructuredActivity*
- *steps: refers to the set of contained Activities*
- *condition: represents a set of boolean expressions to define the conditions under that the StructuredActivity is executed*

The variables controlFlow, localKnowledge, inFlow, outFlow, and messageScope are used in the same manner as defined by Definition 4.7.4.

The semantics of *StructuredActivity* is given in Schema 4.7.5. Apart from the abstract syntax given in Definition 4.7.5, it further introduces secondary variables like *active* that states that any *Activity* contained by the *StructuredActivity* (i.e. expressed through the variable *step*) is active (cf. Invariant I10). Moreover, the variables *strartActivities* and *endActivities* give the first (cf. Invariant I1) and last *Activities* (cf. Invariant I2) with respect to the control flow of a *StructuredActivity*. Both, the *startActivities* and *endActivities* only contain one element each (cf. Invariant I6). The secondary variable $steps^+$ represents all *Activities* contained by this *StructuredActivity* recursively (cf. Invariant I3). Finally, *activeControlFlows* depicts all outgoing *ControlFlows* of a *startActivity* that have either an empty condition or the condition evaluates to true (cf. Invariant I11). Moreover, the variable *completed* is re-defined within the *StructuredActivity*, It is *completed*, iff all *endActivities* are completed (see Invariant I12).

Other invariants are: The *condition* must at most consist of single logical expression (cf. Invariant I7), and the *steps* (cf. Invariant I4) and *controlFlows* (cf. Invariant I5) are considered as unique. In addition, Invariant I8 states that any *controlFlow* must only link two *Activities* that are contained by the *StructuredActivity*. Similarly, Invariant I9 restricts the *inFlow* and *outFlow* of a contained *Activity* to those *ControlFlows* that are part of the *controlFlow* variable. Finally, as expressed by Invariant I13, a *StructuredActivity* contains at least three other Activities, i.e. the *startActivities*, the *endAtivities*, and an *Activity* including a certain functionality.

4.7.7 Split

The concept of *Sequence*, as discusses in Section A.4.3, does not support the branching of control flows. For exactly this purpose, we introduce the abstract concept of *Split* that is a point in the body of a *Plan*, where a single thread of control splits into multiple threads of control. The abstract syntax is defined as follows:

Definition 4.7.6 (Split in PIM4AGENTS)

A Split is given by a 10-tuple S = (name, steps, flows, condition, localKnowledge, inFlow, outFlow, messageScope, traces, synchronizationMode), where:

- *name: specifies the name of the Split*

StructuredActivity

Activity

$steps : \mathbb{P}_1 \downarrow Activity©, condition : \mathbb{P}\,\mathbb{B}$	[Variables]
Δ	[Semantic Variables]
$active : \mathbb{B}, startActivities, endActivities : \mathbb{P} \downarrow Activity$	
$steps^+ : \mathbb{P}_1 \downarrow Activity;\ activeControlFlows : \mathbb{P}\ ControlFlow$	
$startActivities == \{s : steps \mid s \in Begin\}$	[I1]
$endActivities == \{s : steps \mid s \in End\}$	[I2]
$steps^+ == steps \cup \{s : \{sa : steps \mid sa \in StructuredActivity\}, a : Activity \mid a \in s.steps^+ \bullet a\}$	[I3]
$\forall s_1, s_2 : steps \bullet s_1.name = s_2.name \Rightarrow s_1 = s_2$	[I4]
$\forall f_1, f_2 : controlFlow \bullet f_1.name = f_2.name \Rightarrow f_1 = f_2$	[I5]
$\#startActivities = \#endActivities = 1$	[I6]
$\#condition \leq 1$	[I7]
$\forall cf : controlFlow \bullet cf.sink \in steps \wedge cf.source \in steps$	[I8]
$\forall s : steps \bullet s.inFlow \subseteq controlFlow \wedge s.outFlow \subseteq controlFlow$	[I9]
$active \Rightarrow \exists s : steps \bullet s.active$	[I10]
$activeControlFlows == \{acf : \bigcup \{cf : startActivities \bullet cf.outFlow\} \mid$ $\bigwedge head\, acf.condition = \text{true} \vee af.condition = \varnothing\}$	[I11]
$completed \Leftrightarrow \forall e : endActivities \bullet e.completed$	[I12]
$\#steps \geq 3$	[I13]

$OverallExit \mathrel{\widehat=} exit$
$InnerEntry \mathrel{\widehat=} \bigwedge s : startActivities \bullet s.enter$

Schema 4.7.5: Class schema of *StructuredActivity*

- *traces: defines the number of traces that must be synchronized*
- *synchronizationMode: defines the manner in which the threads of control are synchronized*

The variables steps, flows, condition, localKnowledge, inFlow, outFlow, and messageScope are used in the same manner as specified by Definition 4.7.5.

Split

StructuredActivity

$synchronizationMode : SynchronizationMode;\ traces : \mathbb{N}$	[Variables]
$\#\bigcup \{s : startActivities \bullet s.outFlow\} \geq 2$	[I1]
$\forall cf : (controlFlow \setminus activeControlFlows) \bullet cf.condition = \varnothing$	[I2]
$synchronizationMode = NofM \Rightarrow traces \leq \#activeControlFlows$	[I3]
$synchronizationMode \neq NofM \Rightarrow traces \geq 1$	[]
$synchronizationMode = XOR \Rightarrow traces == 1$	[]

Schema 4.7.6: Class schema of *Split*

The semantics of a *Split* is as follows: Any concept of kind *Split* must have more than one outgoing *ControlFlow* representing the split of execution control (cf. Invariant I1). Apart from the *active-ConotrolFlows*, Invariant I2 states that any *Split's ControlFlow* must have no further restrictions on its guard. Moreover, if the synchronization mode of a *Split* is *NoM*, the number of traces must be

smaller than the number of active *ControlFlows* (cf. Invariant I3). Two specializations of a *Split* are distinguished, i.e., the *Parallel* and *Decision* concept.

4.7.8 Parallel

Apart from sequential execution, in some situations, it might be necessary to execute *Activities* in parallel. The different execution branches are started and joined at some point in the body. Even further, parallel processing *Activities* could depend on each other.

A *Parallel* in PIM4AGENTS is a point in a *Plan*, where a single thread of control splits into multiple threads of control, which are executed in parallel at the same time, thus allowing Activities to be executed simultaneously or in any order. The abstract syntax of *Parallel* is as follows:

Definition 4.7.7 (Parallel in PIM4AGENTS)

A Parallel is given by a 10-tuple P = (name, steps, flows, condition, localKnowledge, inFlow, outFlow, messageScope, traces, synchronizationMode). The abstract syntax of these variables is given in Definition 4.7.6.

The only restriction of a *Parallel* is that the *condition* of any *startActivity's outFlow* must either be satisfied or empty. This is expressed through Invariant I1 expressing that any outgoing *ControlFlow* is active. The Object-Z specification of *Parallel* is given in Schema A.4.2.

4.7.9 Decision

Beside a *Parallel*, the concept of *Decision* also allows splitting of *ControlFlow*, where a *Decision* is a specialization of *StructuredActivity*, which defines a location in a *Plan* where the *ControlFlow* is split into two or more alternative branches. Based on a *condition*, at least one *Activity* of a number of branching *Activities* must be chosen. Whether the *Decision* is exclusive–meaning that only one of the alternative paths may be chosen or not—is expressed through the variable of the enumeration *DecisionMode*.

Definition 4.7.8 (Decision in PIM4AGENTS)

A Decision is defined by a 11-tuple D = (name, steps, controlFlows condition, localKnowledge, inFlow, outFlow, messageScope, traces, synchronizationMode, executionMode), where:

- *name: represents the name of the Decision*
- *executionMode: defines the execution semantics of the Decision*

The variables steps, controlFlows condition, localKnowledge, inFlow, outFlow, messageScope, traces, and synchronizationMode are defined in accordance to Definition 4.7.6.

When a *Decision* is reached, it results in the dynamic evaluation of the *conditions* of its *Begin's* outgoing *ControlFlows* to realize a dynamic conditional branch. That is, the *Decision* whose transition will be selected is chosen during the execution of the *Plan*. A static conditional branch specifies that the execution path is already determined prior to the firing of the first transition. This is contrary to a dynamic *Decision* where the *conditions* are only evaluated when the transition to this *Decision* is being taken. This implies that the selection of a transition from this Decision may depend on the *Activities* which have been executed up to this point.

If more than one of the *conditions* evaluates to true, a transition is selected non-deterministically. If none of the *conditions* evaluates to true, the model is considered ill-formed. A pre-defined else *condition* is available yielding a true value if all other transition *conditions* yield false. A non-deterministic *Decision* can be implemented if all *conditions* on the outgoing transitions have a true value.

Decision

Split

executionMode : *DecisionMode*	[Variables]
$executionMode = OR \Rightarrow \#activeControlFlows \geq 1$	[I1]
$executionMode = XOR \Rightarrow \#activeControlFlows = 1$	[I2]

Schema 4.7.7: Class schema of *Decision*

4.7.10 ParallelLoop

The *Parallel's* branches need to be defined at design time, however, in some situations, it is necessary to leave the number of branches open till run-time. Especially for these situations, the construct of a *ParallelLoop* has been brought in. Its abstract syntax is defined as follows:

Definition 4.7.9 (ParallelLoop in PIM4AGENTS)

A ParallelLoop is defined by a 11-tuple P = (name, steps, controlFlows, condition, localKnowledge, inFlow, outFlow, messageScope, traces, synchronizationMode, postCondition). These variables are defined in accordance to Definition 4.7.7 and Definition A.4.2.

The semantics of *ParallelLoop* inherits the semantics of *Loop* and *Parallel*. The only necessary refinement is that the number of *activeControlFlows* is restricted to 1. This is expressed in Schema A.4.5.

The metamodel of the behavioral viewpoint covering simple atomic tasks is depicted in Fig. 4.9. It includes the concepts *Task*, *SendMessage*, *ReceiveMessage*, *InternalTask*, *InitiateProtocol*, *Wait*, *Message*, *Protocol*, and *TimeOut* (the last two were part of the interaction aspect). Furthermore, it contains an enumeration called *MessageType* containing the literals *Asynchronous* and *Synchronous*.

4.7.11 ExecutionMode

To choose exactly one execution path from many alternatives is certainly the simplest form of decision. However, there is a more complex case that needs to be covered by the plan vocabulary. The enumeration *ExecutionMode* allows determining how the branches of a Decision in PIM4AGENTS are executed.

$ExecutionMode ::= OR \mid XOR$

OR mode allows choosing from one to all alternative paths at run time. Technically, it may allow zero paths chosen, but this could be considered as an invalid situation, where the ControlFlow stops unexpectedly.

XOR is defined as being a location in a Plan, where the flow is split into two or more exclusive alternative paths. The pattern is exclusive in the manner that only one of the alternative paths may be chosen for the Plan to continue.

A third alternative would be to execute the different branches in an "and" manner, where all paths are executed at the same time. However, in this case, the concept of a *Parallel* would be the best choice as it provides exactly the necessary operation semantics.

4.7.12 SynchronizationMode

The convergence of two or more threads of control into a single subsequent thread of control requires a mechanism to specify how to synchronize the branches. In which manner to synchronize the branches in PIM4AGENTS is expressed through the enumeration *SynchronizationMode*, which offers the following opportunities of synchronization:

SynchronizationMode ::= *OR* | *XOR* | *AND* | *NofM*

OR is defined as being a location in a *Split* where a set of alternative paths is joined into a single path.

XOR allows to synchronize exactly one path.

AND allows to synchronize all parallel paths. The challenge is that it will not be known ahead of time how many paths will actually arriving (i.e. are active). Thus, in the case of an *AND* it must be determined how many paths are activated, followed by synchronizing them without waiting for any other path.

NofM allows the modeler to define how many of the incoming alternatives are necessary to continue. All remaining paths are stopped and not further needed to continue the *ControlFlow*.

4.7.13 Task

In PIM4AGENTS, a *Task* defines an abstract superclass, its specializations are considered as one-shot behaviors in which a single action is performed. The formal definition of *Task* is as follows:

Definition 4.7.10 (Task in PIM4AGENTS)

A Task is defined by a 6-tuple T = (name, outFlow, inFlow, localKnowledge, messageScope, controlFlow), where these variables are defined in accordance to Definition 4.7.4.

In contrast to a *StructuredActivity*, a *Task* can be considered as an atomic *Activity*. Thus, it does not contain any *Activity* nor *ControlFlow* (the latter is expressed by Invariant I1 of Schema A.4.6).

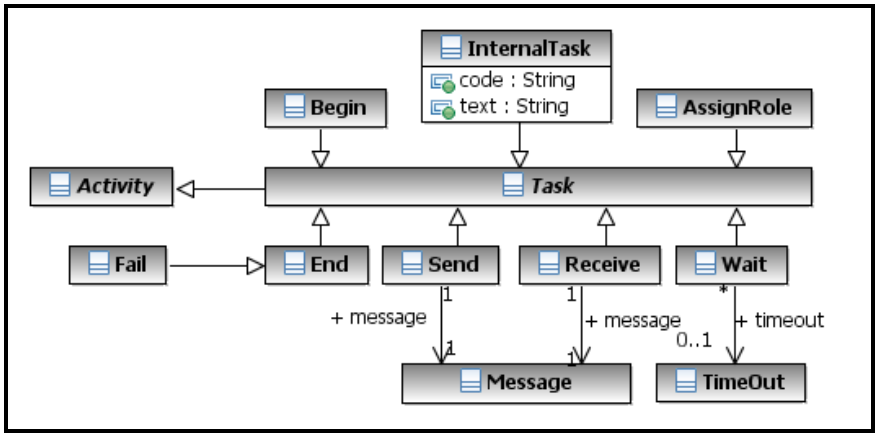


Fig. 4.9: The specializations of a Task in PIM4AGENTS (partial).

4.7.14 Send

The order in which *ACLMessages* are exchanged between actors is part of the interaction viewpoint discussed in Section 4.6. However, in order to complete the mechanism of exchanging messages, we need to define the actions necessary for sending and receiving the particular *Messages*. In this regard, the *Send* concept, as a specialization of *Task*, is a point within a *Plan* in which a *Message* is sent. The abstract syntax of *Send* is given in Definition 4.7.11.

Definition 4.7.11 (Send in PIM4AGENTS)

A Send is defined by a 7-tuple S = (name, inFlow, outFlow, flows, messageScope, localKnowledge, message, inInfoFlow, outInfoFlow), where:

- *name: defines the name of the Send*
- *message: depicts the particular Message that is sent within this Send*

The variables outFlow, inFlow, localKnowledge, messageScope, inInfoFlow, outInfoFlow, and controlFlow are defined in accordance to Definition 4.7.10.

As specialization of *Task*, a Send is an atomic *Activity* that is responsible for sending the attached *Message*. The formal semantic of *Send* is given in Schema 4.7.14. In addition to the variables already declared in Definition 4.7.11, we specify the semantic variable *messageIsSent* which can be considered as similar to *completed*.

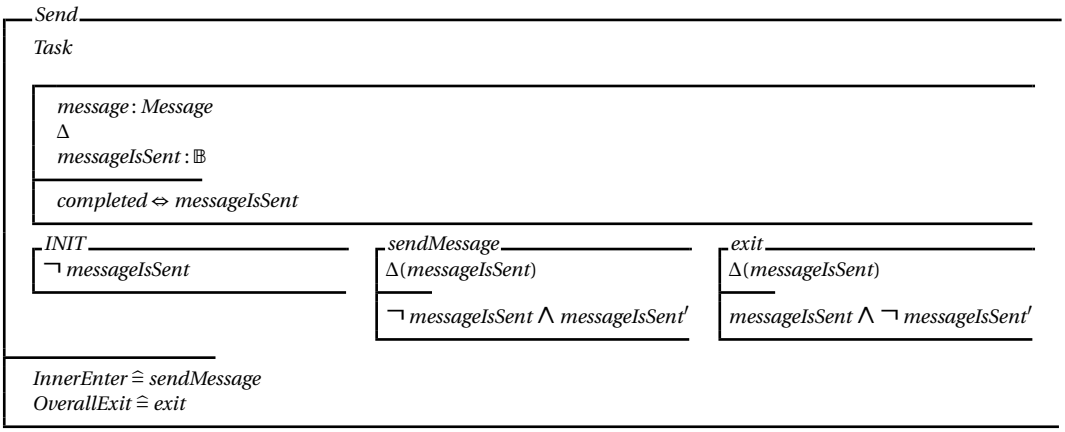
4.7.15 Receive

In correspondence to the *Send Activity*, *Receive* is a specialization of a *Task* within a *Plan* in which a *Message* is received.

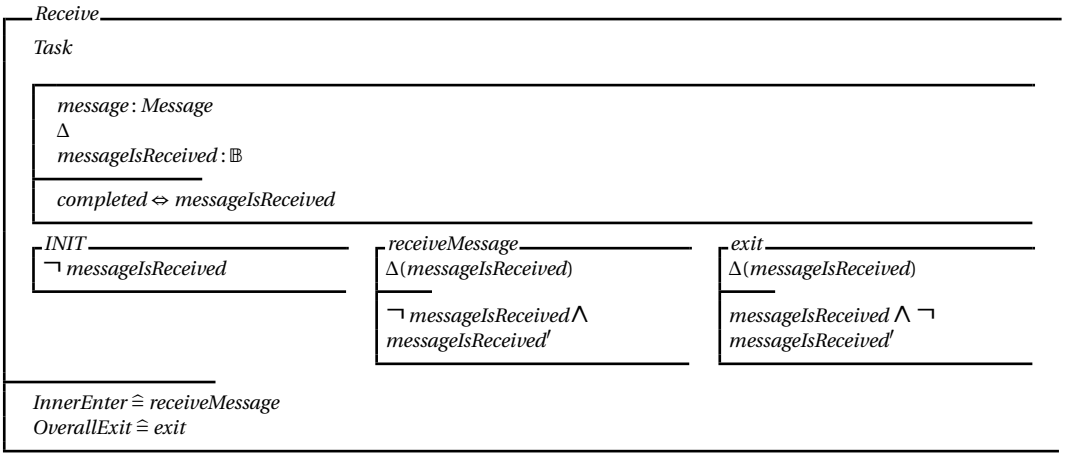
Definition 4.7.12 (Receive in PIM4AGENTS)

A Receive is defined by a 7-tuple R = (name, inFlow, outFlow, flows, messageScope, localKnowledge, message, inInfoFlow, outInfoFlow), where:

- *name: defines the name of the Receive*



Schema 4.7.8: Class schema of *Send*



Schema 4.7.9: Class schema of *Receive*

- *message: depicts the particular Message that is received within this Receive*

The variables outFlow, inFlow, localKnowledge, messageScope, inInfoFlow, outInfoFlow and controlFlow are defined in accordance to Definition 4.7.10.

The concrete semantics of *Receive* is depicted by Schema 4.7.9. Beside the variables introduces in Definition 4.7.12, moreover, we introduce the semantic variable *messageIsSent* that is true, iff the *Receive* task has been completed and the message successively sent.

Within a *Plan*, particular functionality needs to be expressed that cannot be specified in a graphical manner. For these kinds of situations, the designer may want to directly specify the necessary code. In PIM4AGENTS, this could be achieved through the concept of *InternalTask* that is formally defined in Section A.4.7. Other specializations of *Task* are, for instance, *Wait*, *Begin*, *Fail*, and *End*. Details on their abstract syntax and semantics are given in the Appendix at A.

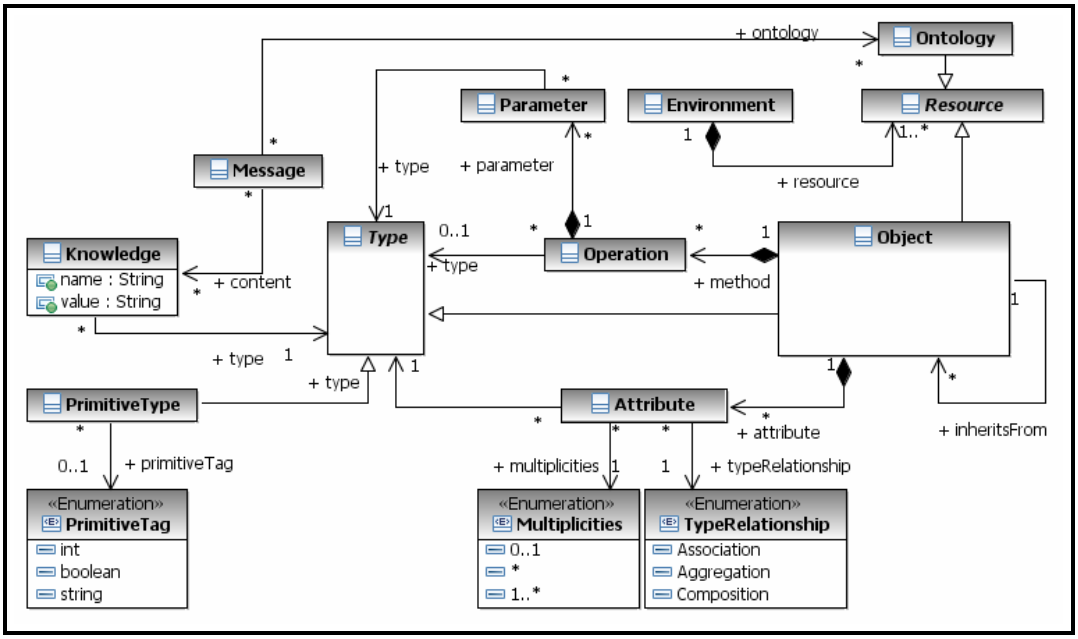


Fig. 4.10: The environment viewpoint of PIM4AGENTS.

4.8 Environment Viewpoint

To provide a general modeling abstraction and general modeling technique for agent-based environments is a very complex and difficult task. The main reason is that environments for different applications can be very different in nature as they are somehow related to the underlying technology (Zambonelli et al.; 2003). In order to conform to the definitions given in Section 2.1.7, we consider anything that is not directly part of an agent or organization as part of the environment. Hence, the environment may comprehend objects that can be either used by the agents, or information that can be perceived. To provide a general metamodel for a multiagent-based environment, we suggest to treat the environment in terms of resources like objects and services available to the agent society. These resources can be accessed and changed by any entity having access to them.

4.8.1 Environment

The *Environment* concept of PIM4AGENTS illustrates any kind of information or resource that can be accessed and used by agents. Even if the core *Environment* emphasizes on modeling objects and information, it can be easily extended to cover the various kinds of applicational settings. (Hahn et al.; 2008b), for instance, demonstrated one option of extending the *Environment* by semantic Web services. The abstract syntax of *Environment* is given in the following definition.

Definition 4.8.1 (Environment in PIM4AGENTS)

An Environment is given by a pair E = (name, resource), where name defines the name of the Environment and resource specifies the set of Resources available to the agents situated in the Environment.

The semantics of *Environment* is given in Schema A.5.1. The class schema inherits from the class schema of *NamedElement* and includes the variable *resource* (cf. Definition 4.8.1). Furthermore, the semantics of *Environment* is refined by Invariant I1 stating that any two *Resources* must be unique, otherwise they are considered as equal. A specialization of the abstract *Resource* is the concept of an *Object* that is treated in the next section.

4.8.2 Object

As stated before, agents do not exist in pure isolation. Instead, agents normally interact with objects or other agents to solve particular tasks and goals. *Objects* are part of the environment viewpoint and defined as follows:

Definition 4.8.2 (Object in PIM4AGENTS)

An Object is given by a 4-tuple O = (name, inheritsFrom, attribute, method) where

- *name: defines the name of the Object*
- *inheritsFrom: describes a taxonomic relationship between Objects*
- *attribute: depicts the set of Attributes the Object has available*
- *method: depicts the set of Operations the Object has available.*

Object

Type

$operation : \mathbb{P}\, Method©;\ attribute : \mathbb{P}\, Attribute©;\ inheritsFrom : \mathbb{P}\, Object$ [Variables]

Δ [Semantics Variables]

$inheritsFrom^{+} : \mathbb{P}\, Object$

$$inheritsFrom^{+} == inheritsFrom \cup \bigcup \{i : inheritsFrom \bullet i.inheritsFrom^{+}\} \quad [I1]$$

$$\forall a_1, a_2 : attribute \bullet a_1.name = a_2.name \Rightarrow a_1 = a_2 \quad [I2]$$

$$\forall o_1, o_2 : operation \bullet (o_1.name = o_2.name \wedge \#o_1.parameters = \#o_2.parameters \wedge \#o_1.type = o_2.type \wedge \forall i : 1..\#o_1.parameters \bullet o_1.parameters(i).name = o_2.parameters(i).name \wedge o_1.parameters(i).type = o_2.parameters(i).type) \Rightarrow o_1 = o_2 \quad [I3]$$

$$self \notin inheritsFrom^{+} \quad [I4]$$

Schema 4.8.1: Class schema of *Object*

An Object owns a name as well as *attributes* and *operations*. Schema 4.8.1 specifies the static semantic of *Object*. Beside the primary variables introduced by Definition 4.8.2, the semantic variable $inheritsFrom^{+}$ is introduced including the set of *Objects* this particular *Object* inherits from. This relationship is defined as the union of the *Object's* generalized *Objects* (expressed through the *inheritsFrom* variable) and the $inheritsFrom^{+}$ of these *Objects*. In addition, the Invariants I2 and I3 ensure that neither the *attributes* nor the *operations* of one and the same *Object* are ambiguous. Finally, an *Object* must not be a specialization of itself (see Invariant I4). The abstract syntax and semantic of the *Operation* and *Attribute* concepts are given in Section A.5.4 and Section A.5.3.

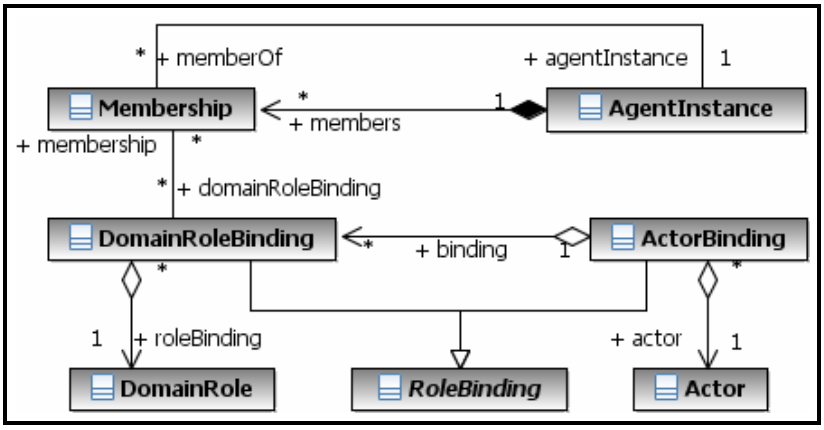


Fig. 4.11: The metamodel reflecting the deployment viewpoint of PIM4AGENTS.

4.9 Deployment Viewpoint

The views discussed so far mainly dealt with the type level. In terms of MDD and object-oriented languages like UML, instances are normally introduced on a lower level of the abstraction hierarchy compared to its type. However, in agent systems, for certain kinds of scenarios, it is certainly helpful to introduce the particular agent instances of an agent type already at design time. These scenarios (i.e. often referred to as closed MAS (Davidsson; 2001)) are normally characterized by a fixed number of run-time instances, which do not change during execution. In order to introduce run-time entities, we introduce the deployment viewpoint that defines (i) the run-time entities and (ii) how they are grouped into social structures defined in the organization viewpoint (cf. Section 4.4).

The metamodel of the deployment viewpoint is depicted in Fig. 4.11. It includes the concepts *AgentInstance*, *RoleBinding*, *DomainRoleBinding*, *ActorBinding*, *Membership*, as well as *DomainRole* and *Actor* from the role viewpoint (cf. Section 4.5).

4.9.1 AgentInstance

General purpose programming language like Java and more agent-based platforms like JACK and JADE have in common that when implementing systems, the programmer first defines the different kinds of types, followed by introducing the run-time instances and assigning the instances to their types. By using PIM4AGENTS, these three steps could already be carried on the design level.

For this purpose, the concept of an *AgentInstance* is introduced that specifies the autonomous, interactive entity in the running system. The main advantages of introducing run-time instances during design-time is to facilitate the binding between instances and roles. Hence, *AgentInstances* are directly assigned to either *DomainRoles* or *Actors* as role fillers. This assignment is done through the concept of a Membership that directly refers to a certain *DomainRoleBinding* to express that the particular *AgentInstance* currently plays the *DomainRole* referred by the *DomainRoleBinding*. For specifying the binding between *Actor* and *AgentInstance*, the concept of *ActorBinding* (see Section 4.9.4) is utilized. The abstract syntax of an *AgentInstance* is given in Definition 4.9.1.

AgentInstance

NamedElement

$agentType : \downarrow Agent;\ memberOf : \mathbb{P}\ Membership;\ members : \mathbb{P}\ Membership©$	[Variable]
$\forall m_1, m_2 : members \bullet m_1.name = m_2.name \Rightarrow m_1 = m_2$	[I1]
$\forall m_1, m_2 : members \bullet m_1.agentInstance = m_2.agentInstance \Rightarrow m_1 = m_2$	[I2]
$\forall a : agentType \mid a \notin Organization \bullet members = \varnothing$	[I3]
$\forall a : agentType \mid a \in Organization \bullet$	
$\quad \forall m : members \bullet m.domainRoleBinding.roleBinding \in a.requiredRole$	
$\quad \forall m : members \bullet self \in m.memberOf \bigwedge$	
$\quad \{mem : \bigcup \{drb : \bigcup \{co : a.organizationUse \bullet c.binding\} \bullet drb.membership\}$	
$\quad\quad \bullet\ mem.agentInstance\} \subseteq members$	[I4]

Schema 4.9.1: Class schema of *AgentInstance*

Definition 4.9.1 (AgentInstance in PIM4AGENTS)

An AgentInstance is given by a 4-tuple A = (name, agentType, memberOf, members), where:

- *name: defines the name of the AgentInstance*
- *agentType: depicts the Agent type represented by the AgentInstance in the running system*
- *memberOf: refers to the AgentInstances of type Organization to which this AgentInstance is bound to through DomainRoleBindings*
- *members: refers to the AgentInstances part of this particular AgentInstance of agentType Organization.*

As an *Organization* is a specialization of *Agent*, an *AgentInstance* may also refer to an *Organization* as its type. In this case, for each of its members (i.e. *AgentInstances*), the particular *AgentInstance* owns a Membership instance. The members are again either of type *Agent* or *Organization*. The Membership concept is introduced in Section A.6.1.

If an *AgentInstance's agentType* is of the kind *Organization*, it may specify several bindings—one for each *DomainRole* its *Organization* requires. These *DomainRoleBindings* define which *AgentInstances* are bound to which *DomainRoles*. The relationship between *Membership* and *DomainRole* can either be determined at design time by explicitly assigning *AgentInstances* to *DomainRoles* through the *Membership* concept or during run-time. For the latter case, particular properties are provided by PIM4AGENTS (see Section 4.9.2).

The semantics of an *AgentInstance* is given in Schema 4.9.1. It includes—as specified by Definition 4.9.1—the three primary variables *agentType*, *members*, and *memberOf*. The first two invariants ensure that the members of an *AgentInstance* are unique with respect to their *names* (Invariant I1) and *agentInstances* included (Invariant I2). In the third invariant, the *members* set is restricted to the empty set if that *AgentInstance* is not of type *Organization*. In contrast, if an *AgentInstance* is of type *Organization*, Invariant I4 ensures that (i) the *Members* refer to a *DomainRole*, which is required by the *Organization*, (ii) any *Member* is part of the *AgentInstance's members* and finally, (iii) the set of *Members* bound through one of the *Organization's Collaborations* is a subset of all *Members* referring to this *AgentInstance* through the *memberOf* attribute.

4.9.2 RoleBinding

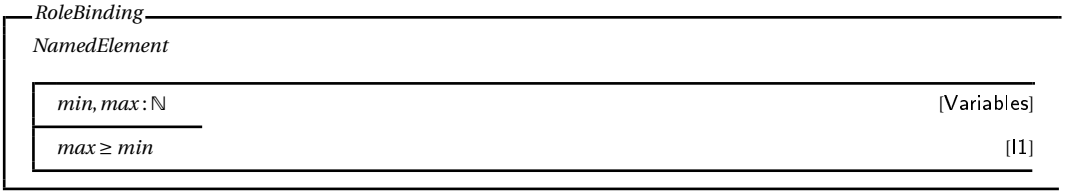
As mentioned earlier, the principle idea behind role assignment is to relate agent instances to roles in any form of agent grouping. In PIM4AGENTS, any kind of role assignment is done through the abstract concept of *RoleBinding* that binds, roughly spoken, *AgentInstances* to *DomainRoles* and *Actors*. The abstract syntax of the concept *RoleBinding* is given in Definition 4.9.2:

Definition 4.9.2 (RoleBinding in PIM4AGENTS)

A RoleBinding is given by a triple R = (name, min, max), where:

- *name: defines the name of the RoleBinding*
- *min: specifies the number of role fillers (i.e. AgentInstances) that need to be bound at least*
- *max: specifies the number of role fillers that need to be bound at most.*

A *RoleBinding* can be employed to directly assign *AgentInstances* to *Roles*. However, in some cases, often referred to as open MAS, the agent instances are primary introduced at the run-time level, which makes the direct assignment of them during design time impossible. However, even in this case, restricting the number of role fillers during design time might be necessary. For this purpose, we introduce the variables *min* and *max*, which prescribe the lower and upper bound of role fillers that can be assigned during run-time. The class schema of *RoleBinding* is given in Schema 4.9.2



Schema 4.9.2: Class schema of *Binding*

Both variables *min* and *max* are natural numbers, where *max* is greater or equal to *min* (cf. Invariant I1). Based on the concept of *RoleBinding*, we further distinguish between the two specializations, i.e. *DomainRoleBindings* binding *AgentInstances* to *DomainRoles* and *ActorBindings* binding *Actors* to *DomainRoles*.

4.9.3 DomainRoleBinding

The concept of *DomainRoleBinding* as specialization of *RoleBinding* is used to assign *AgentInstances* to *DomainRoles*. As a specialization of *RoleBinding*, it specifies which *Members*—representing the certain *AgentInstances*—are bound to which *DomainRole*. An informal definition of *DomainRoleBinding* is specified as follows:

Definition 4.9.3 (DomainRoleBinding in PIM4AGENTS)

A DomainRoleBinding is given by a 5-tuple D = (name, membership, roleBinding, min, max), where:

- *name: defines the name of DomainRoleBinding*
- *membership: illustrates the set of Memberships (i.e. AgentInstances) bound to the DomainRole referred*

- *roleBinding: refers to the DomainRole the AgentInstances are bound to*

The variables min and max are used in the context of a DomainRoleBinding *as specified in Definition 4.9.2.*

The semantics of *DomainRoleBinding* is given by Schema 4.9.3. It inherits from the class schema of *RoleBinding* and includes the variables *membership, roleBinding*, as well as *min* and *max* both of the type Integer. Critically, for all *Members* of *memberships*, it is necessary that the *Agent* referred to by the *AgentInstance* (i.e. through the *agentType* association) is also able to perform the particular *DomainRoleBinding's DomainRole*. This is expressed by Invariant I1 of Schema 4.9.3.

DomainRoleBinding

RoleBinding

$membership : \mathbb{P}\, Membership;\ roleBinding : DomainRole\textcircled{S}$

$\forall\, m : membership \bullet roleBinding \in m.agentInstance.agentType.performedRole$ [I1]

$max \neq 0 \Rightarrow max \geq \{m : membership \bullet m.agentInstance\} \geq min$ [I2]

Schema 4.9.3: Class schema of *DomainRoleBinding*

The number of *AgentInstances* bound to a certain *DomainRole* is furthermore restricted through the *min* and *max* variables. For instance, if *max* is greater than 0, Invariant I2 states that the number of *AgentInstances* is smaller or equal to *max* and greater or equal to *min*. However, if *max* is equal to 0, the number of *AgentInstances* should be considered as arbitrary number that must be fixed at design time.

In the same manner as *AgentInstances* are bound to *DomainRoles* through the concept of *DomainRoleBinding*, the concept of *ActorBinding* binds *DomainRoles* to *Actors*.

4.9.4 ActorBinding

For the purpose of assigning *AgentInstances* to *Actors*, we introduce the concept of *ActorBinding* that binds *Actors* to *DomainRoles*, which are again bound to *AgentInstances* through *DomainRoleBindings* (cf. Section 4.9.3). This consequently means that *AgentInstances* bound to a *DomainRole* through the *DomainRoleBinding* concept will finally send and receive the *ACLMessages* the particular *Actors* exchange in their active *MessageFlows*. The abstract syntax of *ActorBinding* is given in the following definition.

Definition 4.9.4 (ActorBinding in PIM4AGENTS)

An ActorBinding is given by a 5-tuple A = (name, actor, binding, min, max), where:

- *name: defines the name of the ActorBinding*
- *actor: represents the Actor to which AgentInstances are bound*
- *binding: identifies the DomainRoleBindings used in order to extract the AgentInstances bound to the particular DomainRole.*

The variables min and max are used in the context of an ActorBinding *as specified in Definition 4.9.2.*

The formal semantic of *ActorBinding* is given in Schema 4.9.4. The corresponding class schema inherits from the class schema of *RoleBinding* and includes in its declarative part the two variables *binding* and *actor*.

ActorBinding

RoleBinding

$binding : \mathbb{P}\, DomainRoleBinding\textcircled{S};\ actor : Actor\textcircled{S}$ [Variables]

Δ [Semantic Variables]

$instancesBound : \mathbb{P}\, AgentInstance$

$instancesBound == \{me : \bigcup \{drb : b.binding \bullet drb.membership\} \bullet me.agentInstance\}$ [I1]

$max > 0 \Rightarrow max \geq \#instancesBound \geq min$ [I2]

$\forall\, m : actor.messageSent \bullet$
$\quad \forall\, i : instancesBound \bullet$
$\quad\quad \exists\, p : i.agentType.potentialBehaviors, t : Send \mid t \in p.steps^{+} \bullet$
$\quad\quad\quad m = t.message.aclMessage \vee$
$\quad \exists\, b : actor.providesBehavior^{+}, t : Send \mid t \in b.steps^{+} \bullet$
$\quad\quad m = t.message.aclMessage$ [I3, $providesBehavior^{+}$ bereits teil der potentionalBaheivors]

$\forall\, m : actor.messageReceived \bullet$
$\quad \forall\, i : instancesBound \bullet$
$\quad\quad \exists\, p : i.agentType.potentialBehaviors, t : Receive \mid t \in p.steps^{+} \bullet$
$\quad\quad\quad m = t.message.aclMessage \vee$
$\quad \exists\, b : actor.providesBehavior^{+}, t : Receive \mid t \in b.steps^{+} \bullet$
$\quad\quad m = t.message.aclMessage$ [I4]

$actor.conflictsWith \cap \{b : binding \bullet d.roleBinding\} = \varnothing$ [I5]

$actor \notin \bigcup \{b : binding \bullet d.roleBinding.conflictsWith\}$ [I6]

Schema 4.9.4: Class schema of *ActorBinding*

Moreover, the secondary variable called *instancesBound* is introduced that unions all *AgentInstances* bound to the particular *Actor* through the *DomainRoleBinding* (see Invariant I1). Additionally, if *max* is greater than 0, Invariant I2 states that the number of *instancesBound* is smaller or equal to *max* and greater or equal to *min*. However, if *max* is equal to 0, the number of *instancesBound* is an arbitrary number that must not be fixed at design time. Invariants I3 and I4 ensure that any *AgentInstance* bound to an *Actor* through the *ActorBinding* concept has in fact the ability to send and receive the particular *Messages* of the *Interaction's Actors*, respectively. Finally, Invariants I5 and I6 ensure that any *ActorBinding* does not violate the *conficts With* relationship between *Actors* and the *DomainRoles* bound.

4.10 Bottom Line

In this chapter, we discussed the abstract syntax and the formal semantics of DSML4MAS. For defining the abstract syntax, we applied the principles of metamodeling and defined a platform independent metamodel for MASs called PIM4AGENTS. In this respect, we separated PIM4AGENTS into several viewpoints, each focusing on a particular aspect when designing a MAS. These aspects include viewpoints for modeling agent, organizations, roles, interactions, behaviors, environments, and deployment of agents. The resulting PIM4AGENTS metamodel can be considered as the cornerstone to the development of (i) model transformations between the PIM level represented

by PIM4AGENTS and the agent-based platforms situated on the PSM level and (ii) the model-driven integration of SOAs as detailed in Chapter 8.

The formal semantics is defined by using the formal specification language Object-Z. The resulting specification includes the abstract syntax as well as the denotational (static) and operational (dynamic) semantics. In particular, the denotational semantics is defined by introducing additional secondary (i.e. semantic) variables and invariants, the operational semantics is defined in terms of operations and invariants using the timed refinement calculus. The main advantage offered by this approach is that the syntax as well as the static and dynamic semantics can be integrated into one Object-Z schema class. This can hardly be achieved by the combination of UML and OCL as the latter only allows to define the static semantics. By defining the formal semantic model of PIM4AGENTS, existing Object-Z tools can be used for checking, validating and verifying PIM4AGENTS models. This is certainly an important language feature to minimize or even exclude errors on the abstract modeling level.

In addition, this formal specification is further utilized to provide the application developers a clear and detailed understanding on how to use DSML4MAS correctly. Therefore, the formal specifications defined with Object-Z is taken and manually transformed into a corresponding OCL specification. This specification is in a second step integrated into the graphical IDE of DSML4MAS. How this is done in practice is illustrated in the next chapter. By integrating the OCL statements within the graphical editor, we could provide more information to the application developers and assure that the generated models conform to the static semantics defined with Object-Z. Hence, the formal specification of PIM4AGENTS is further used to improve the modeling quality of DSML4MAS.

5. Methodology of DSML4MAS

One of the main problems that prevent AOSE from a broad application in main stream software development is the lack of adequate methodologies and suitable tool support. Both are necessary to support the unexperienced user to design the system in accordance to the vocabulary and semantics given by the language. In the software engineering domain, a methodology defines in general a guided procedure for using the (modeling) language to build software systems in a systematic manner. The methodology's procedure guides all involved roles (e.g. business analysts, system architects, engineers, developers, etc.) in using the given technologies and modeling techniques, and allows them to obtain the maximal benefits for the engineering process that are inherently supported by tools. The guided procedures ideally cover the complete system engineering process from initial design to system development and maintenance along with detailed methods for the modeling and development of specific aspects. The procedures are normally obtained from experiences gained from already completed system engineering projects.

The purpose of this chapter is to present the DSML4MAS methodology aiming at providing guided procedures for supporting the system engineering process of DSML4MAS by using its modeling techniques, tools and code generators. The DSML4MAS methodology includes proper support for all phases of the overall MAS engineering process, i.e. from the analysis stage to the final implementation of the system.

Structure of this Chapter Section 5.1 summarizes the basic components of any methodology. Afterwards, Section 5.2 reports on the tool support provided by DSML4MAS. Section 5.3 then introduces the concrete syntax of DSML4MAS followed by Section 5.4 illustrating the (semi-) automatic model-driven methodology process. Finally, Section 5.5 concludes this chapter.

5.1 Basic Concepts of Methodologies

A methodology is, in accordance to (Ghezzi et al.; 2002), a collection of methods covering and connecting different stages in a process. The purpose of a methodology is to prescribe a certain coherent approach to solving a problem in the context of a software process by preselecting and putting in relation a number of methods. At this, the methodology should be understood as collection of methods covering and connecting different stages or phases in a process. A methodology has two important components, (i) one component that describes the process elements of the approach, and (ii) one component that focuses on the work products and their documentation. AOSE methodologies mainly try to suggest a clean and disciplined approach to analyze, design and develop MASs, using specific methods and techniques that are presented in more detail in Chapter 10, when discussing the state of the art on AOSE modeling approaches.

In this dissertation, the term methodology denotes a set or collection of methods and related artifacts needed to support the model driven engineering of MASs. Our view on the term

methodology as a collection of methods is aligned with e.g. (Blum; 1994): "Methodology: A body of methods, meant to support all software development phases." and with (Estefan; 2007) who defines a methodology in the area of Model Based Systems Engineering (MBSE) as: "A MBSE methodology can be characterized as the collection of related processes, methods, and tools used to support the discipline of systems engineering in a model-based or model-driven context".

The main difference between an agent modeling language and an agent-oriented methodology is that a methodology builds upon the modeling language, but provides in addition mechanisms that support the design, analysis, and development of MASs. This means in accordance to (Bordini et al.; 2007a) that a methodology includes the following aspects:

Concepts define the vocabulary used by the methodology. As already mentioned earlier, for describing and modeling agent systems, a single set of basic concepts is not yet universally accepted or known. So each methodology focuses on concepts considered as most important. The concepts used in DSML4MAS are clearly defined by the abstract syntax given by the PIM4AGENTS metamodel (cf. Chapter 4). The vocabulary along with the viewpoint information is the foundation for creating models and notations.

Models and Notation are the artifacts generated during the analysis and design. These models are depicted using some notation, often graphical. To define the notation and the different kinds of diagrams used to determine the models, the concrete syntax of DSML4MAS is given in Section 5.3, which is based on the concepts and viewpoints given by the PIM4AGENTS metamodel.

Techniques usually formulated as heuristics guide the use through the existing phases in order to refine the developed design. The techniques used in DSML4MAS are given by the model transformations of the DSML4MAS model transformation architecture. As heuristics are not part of this architecture, parts of the design have still to be done manually, even if the model transformation architecture supports the automatic code generation.

Tool support is an essential feature of a methodology in order to support the user during the various design phases. Tools can range from simple drawing packages, to more sophisticated design environments that provide various forms of consistency checking. To provide adequate tool support for DSML4MAS, we developed the DSML4MAS Development Environment (DDE) that includes the concrete syntax defining the graphical visualization, the static semantics to check the conformance of the design, as well as the code generators, which are given in Chapter 7. Section 5.2 is devoted to present DDE.

Process defines the order in which the different software development phases are applied. Any process comprises several phases like the analysis phase, design phase, or the implementation phase. The process of designing in accordance to DSML4MAS is mainly defined through the model transformation architecture. A detailed discussion on the DSML4MAS process model is given in Section 5.4.

For a successful development of any visual modeling language, three facets are needed in accordance to Quatrani (2000): A notation, a process, and a tool. Quatrani claimed that "You can learn a notation, but if you don't know how to use it (process), you will probably fail. You may have a great process, but if you can't communicate the process (notation), you will probably fail. And lastly, if you cannot document the artifacts of your work (tool), you will probably fail". In order to develop a powerful modeling language for MASs, DSML4MAS provides all of these facets (i.e. a notation, process and tool) that are examined in detail in the forthcoming sections of this chapter.

5.2 Tool Support: The DSML4MAS's Development Environment

Only little research has been done with respect to the development of adequate tools to support the design of agent-based systems, which certainly hampers the breakthrough of MASs in industry. In particular, integrated development environment support for developing MASs is rather weak, and existing agent tools do not offer the same level of usability as state-of-the art object-oriented IDEs (Luck et al.; 2006). Beside a graphical visualization, adequate tool support should also provide facilities to support the domain experts with respect to testing, evaluation, and execution of the generated artifacts. In the present section, related work in the area of agent-based modeling tools is capitulated before presenting the core features of the DSML4MAS Development Environment (DDE).

5.2.1 Related Work

Several tool-supported methodologies for developing MASs exist. Examples are PASSI with the PASSI Tool Kit (PTK) (Cossentino and Potts; 2002) and ROADMAP with REBEL (Juan et al.; 2002). These methodologies differ in their scope, tool support, and maturity. In this section, we want to further discuss three tool supported agent-based methodologies that we consider the closest to the DSML4MAS approach[1].

In accordance to (Bresciani et al.; 2004), Tropos is a software development methodology founded on the key concepts of agent-oriented software development by focusing on the requirements analysis, design, and on model checking. The Tropos methodology bases on the Tropos metamodel (see Section 10.2.7 for a detailed discussion) and covers the development phases of requirements analysis, design, and implementation. Tools for goal analysis (GR-Tool, see (Giorgini et al.; 2005)) and model checking (T-Tool, see (Fuxman et al.; 2001)) have been separately implemented. The eCAT tool is used for automated testing. The modeling tool of the Tropos methodology is called TAOM4e[2] and provides code generation for JADE and Jadex. TAOM4e is based on the Eclipse Modeling Framework (EMF[3]) and the Graphical Editing Framework (GEF[4]). One important benefit of Eclipse's Graphical Modeling Framework (GMF) (that is used for creating DDE) over the combination of GEF and EMF is that its tooling component allows the model-driven creation of a graphical editor based on the underlying metamodel.

In accordance to (Padgham and Winikoff; 2002a), Prometheus is an agent-oriented software engineering methodology. Prometheus supports the whole agent-oriented software development process from analysis to implementation. The Prometheus Design Tool (PDT[5]) (Thangarajah et al.; 2005) offers diagrams for the high-level analysis of a system, the refinement with interaction diagrams with Agent UML (AUML, (Odell et al.; 2000)), and the specification of processes. PDT contains a cross checking tool that covers problems like inconsistency checking, identification of dangling model elements, type checking, etc. Moreover, PDT provides code generation for JACK. It seems that PDT was implemented as a usual Java Swing application. There also exists a plug-in for the Eclipse platform but the integration seems to be very weak.

[1] In Chapter10, these three methodologies are further evaluated in terms of their metamodel and interoperability support.

[2] http://sra.itc.it/tools/taom4e/

[3] http://www.eclipse.org/modeling/emf/

[4] http://www.eclipse.org/gef/

[5] http://www.cs.rmit.edu.au/agents/pdt/

According to (Pavón and Jorge; 2003), INGENIAS is a methodology for specifying MASs on a platform independent level. The INGENIAS metamodel covers aspects such as organizations of agents, agent interactions, and environments of MASs. The INGENIAS methodology is supported by the graphical modeling tool INGENIAS Development Kit (IDK) (Gomez-Sanz et al.; 2008a) providing code generation for JADE, where the INGENIAS Code Uploader extension supports refactoring of the generated code. INGENIAS belongs to the few approaches that also focus on code generation and implementation.

Instead of focusing purely on the analysis and design phases as most of existing related approaches do, the aim of the DSML4MAS methodology is to cover the whole design process from analysis to executable code. At this, PIM4AGENTS presents an expressive language that can be used to generate most parts of a MAS implementation. In our point of view, the automatic code generation and suitable tool support are critical for the practical application of agent-oriented software engineering. Our contribution consists of an expressive platform independent modeling language and adequate tool support that guides the user in designing and implementing MAS. The core features of DDE are as follows.

5.2.2 Features of the DSML4MAS Development Environment

DDE has been implemented using GMF and is based on a plug-in architecture. Hence, DDE inherits many useful features from GMF, such as unlimited undo and redo, auto-arrange, snap-to-grid, modeling assistance, a graphical outline, picture export to save the design in jpeg or png formats, and many more. The following section summarizes the core features of DDE that were not inherited by GMF, but base on the work presented in the upcoming chapters.

Model-driven approach DDE integrates a model-driven development process to close the gap between design and code. Hence, models in accordance to PIM4AGENTS are transformed to platform-specific agent models of JACK and JADE. Details of the DSML4MAS to JACK transformation are given in Section 7.3. In a second step, source code for JACK and JADE is generated using the model-to-text transformation engine of MOFScript. Finally, the generated source code can be edited, executed and tested.

Reduction of Complexity To reduce the complexity of the MAS design is one of the main objectives of the AOSE research area. For this purpose, DDE offers several views on a MAS. Each view (e.g. agent view, protocol view, deployment view, etc.) focuses on a certain aspect and abstracts from others. Changes that affect several views at the same time are automatically propagated to the others. Details on the DSML4MAS's specific views are given in Section 5.3.

Model validation Most of the design errors made when building MASs can already be captured at the model level. DDE integrates the static semantics of DSML4MAS (cf. Chapter 4) utilized to check the syntactic correctness of the created models. For this purpose, constraints based on OCL have been manually derived from the formal Object-Z specification of DSML4MAS to check the static semantics of the created models. These constraints are automatically evaluated during design time and support the developer to produce well-formed models. Details on the OCL specification are given in Section 5.3.2.

Integration of Service-Oriented Architectures MASs do not exist in pure isolation. Instead, they need to be integrated and combined with other related technologies like for instance Service-Oriented Architectures (SOAs). Therefore, in (Hahn et al.; 2008b), we demonstrated how to

integrate Semantic Web services into DDE. The service description can be defined at design-time and invoked by the run-time agents. Beside Semantic Web technologies, moreover, we provide a model-driven integration of SOAs into MAS defined in accordance to DSML4MAS. Chapter 8 is devoted to this integration.

Reusable Components DDE allows the user to reuse components (like plans, protocols, organizational structures, etc.) across several projects. This reduces development time and cost, and increases the quality of the components.

Refinement To support developers in specifying behaviors that conform to a certain protocol, we provide refinement functionalities that transform protocols into behaviors. This kind of refinement is part of the overall process guiding the design with DSML4MAS and is presented in Chapter 6.

Extensibility DDE is seamlessly integrated into the Eclipse workbench. This implies that the community can easily develop own extensions for DDE (e.g. transformations, views, model validation, etc.) and plug them into the Eclipse workbench. Furthermore, the DDE directly benefits from new developments around the very active Eclipse modeling project[6] and other Eclipse tools.

Open source DDE is launched as open source project. The source code is published under LGPL and can be downloaded[7] for free.

5.3 Models and Notation: The Concrete Syntax of DSML4MAS'S

The previous chapter dealt with the abstract syntax defined by the PIM4AGENTS metamodel and the semantics of DSML4MAS. The next step toward a graphical editor for DSML4MAS is the specification of the concrete syntax. This section explains how the concrete syntax of DSML4MAS is specified. After choosing the graphical notation for the concepts and relations, GMF by Eclipse is utilized to tie the domain concepts and their notation together.

5.3.1 Role of Notation

A common misunderstanding in the modeling world is the equality between a diagram and a model. Important to note is that a diagram is not a model. Instead a diagram is considered as a representation of some aspect or view of the model, using visual representation like lines, boxes, etc. According to the OMG way of thinking, the model is the whole machine-readable description of the system. Following this way of thinking, a modeling tool should not be only about the manipulation of a diagram, but generate a model that conforms to its metamodel. Apart from the model and metamodel, a diagram is from a user perspective the most important artifact (Booch; 1995).

Definition 5.3.1 (Notation, according to Booch (1995)

A notation serves as the language for communicating decisions that are not obvious or cannot be inferred from the code itself, provides rich enough semantics sufficient to capture all important strategic and tactical decisions and offers a concrete form for humans to reason about decisions.

[6] http://www.eclipse.org/modeling/
[7] https://sourceforge.net/projects/dsml4mas/

In order to determine a reasonable notation for MASs, we decided to adopt existing notations domain experts already use wherever possible, instead of re-inventing the wheel. However, as DSML4MAS emphasizes on the complete development process from requirement specification to implementation, none of the existing proposals of suitable agent-based notations fits to 100 %. Hence, we extended existing notations from the agent world and mixed it with basic notations from UML aiming at the development of a descriptive and distinguishable representation of the DSML4MAS elements.

5.3.2 Nine Diagrams to Design Multiagent Systems

GMF provides the possibility of creating multiple diagrams for one graphical editor. In order to reduce complexity when modeling with DSML4MAS, we split the design into various diagrams, i.e. for each viewpoint of PIM4AGENTS at least one diagram is created. At this, the main intention is to make the design the most intuitive for the user by reducing the complexity of the designing process itself and thus allowing for separation of concerns. Even if the design is split into various diagrams, all different diagrams within one project share the same model, which is an instance of the PIM4AGENTS metamodel. This feature has two main advantages as it allows (i) cross-checking among the diagrams and (ii) applying model transformations on one model. To illustrate how to model with DSML4MAS in a graphical manner, we studied a very intuitive example, i.e. the conference management system.

5.3.2.1 Illustrative Example: Conference Management System using DSML4MAS

In order to demonstrate the notation of DSML4MAS, throughout this thesis, we want to use the development of an agent-based conference management system (CMS, (Padgham and Luck; 2007; Zambonelli et al.; 2001)) as example that has already been used at the AOSE workshop in 2007 to provide a comparison between different AOSE tools and methodologies like O-MaSE (DeLoach; 2007), Tropos (Morandini et al.; 2007) and Prometheus (Padgham et al.; 2007b). Beside the good foundation we could base on, this case study is sufficiently familiar to most scientists, who already submitted a scientific paper to a conference or workshop.

For our purposes, the committee of a conference consists of the program committee chair, the program committee members, the reviewers, and the authors. These entities are now engaged in the following activities of the conference management:

- Submission phase: The program chair sends a call for papers to researchers that might be interested to contribute to the conference. If a researcher is interested in writing a paper, he/she has to submit his/her contributions before the submission deadline elapses. The author then receives an acknowledgment of his/her submission together with a paper identification number.
- Reviewing phase: Once the submission deadline has passed, the program chair partitions the set of papers received and assigns them to the program committee members in accordance to their particular interests (possibly expressed through a bidding on papers the PC have a particular interest on). The members of the program committee review the papers by either contacting referees and asking them to review a number of the papers, or by reviewing the papers themselves.
- Paper selection and author notification: The result of the reviews is collected by the program committee members and is sent back to the program committee chair. Based on the reviewers' recommendations, the chair is then responsible for making a decision, which

Fig. 5.1: The notation of the agent diagram. From left to right, the notations of agent, plan, capability, and domain role are depicted.

paper to accept or reject. Once the decision has been made, the corresponding authors of the papers are informed accordingly.

- Paper revising, final submission, and printing of proceedings: Authors who got an acceptance are now in charge of preparing a camera ready version, which again has to be submitted before the final deadline elapses. Once all camera ready versions have been received, the chairs produce a preface and finally prepare the draft proceedings, which are sent to the publisher for printing.

The remainder of this section is devoted to discuss the notation (i.e. concrete syntax and graphical editor) of the different diagrams by means of the CMS example. Therefore, we give an overview on the different diagrams followed by a detailed presentation using the CMS as example.

5.3.2.2 Agent Diagram

Modeling Constructs of the Agent Diagram The agent diagram is especially useful for modeling single agents types (which can get quite complex), but also for getting an overview of agent types available, which is particular useful for complex use cases. It allows modeling the different kinds of agent types, their knowledge, plans, capabilities as well as the domain roles that are performed by the agents. Hence, it integrates constructs from different views like the role and behavior diagram. The notation of the agent diagram is depicted in Fig. 5.1.

Apart from the domain roles directly introduced on this diagram, it furthermore, includes any domain role that is either instantiated in the role diagram (cf. Section 5.3.2.5), multiagent system diagram (cf. Section 5.3.2.11) or organization diagram (cf. Section 5.3.2.3). The same holds for plans (i.e. behavior diagram (cf. Section 5.3.2.7)), which can be opened by double-clicking on the plan icon.

Agent Diagram for CMS In the CMS use case, we distinguish two agent types, namely *Researcher* and *Senior Researcher*. The main difference between both is that a *Researcher*, in contrast to *Senior Researcher*, cannot be the chair of a conference. This is expressed by the fact that the *Researcher* agent cannot perform the *PC Chair* domain role.

Fig. 5.2 illustrates the graphical representation of the described scenario. The different roles of the CMS are modeled as domain roles that are performed by the agent types. For example, the *Researcher* agent is permitted to the *PC Member*, and *Author* domain roles, whereas the *Senior Researcher* can additionally perform the *PC Chair* domain role.

The behavior that is required by an agent to perform a domain role is specified by plans. For example, the *Evaluate Papers* plan specifies what an agent has to do to when acting as *PC Chair* in order to evaluate the received papers. In contrast, the *Write Paper* plan defines how the *Author* domain role acts when writing a paper for a conference. Beside the *WritePaper* plan, the *Researcher*

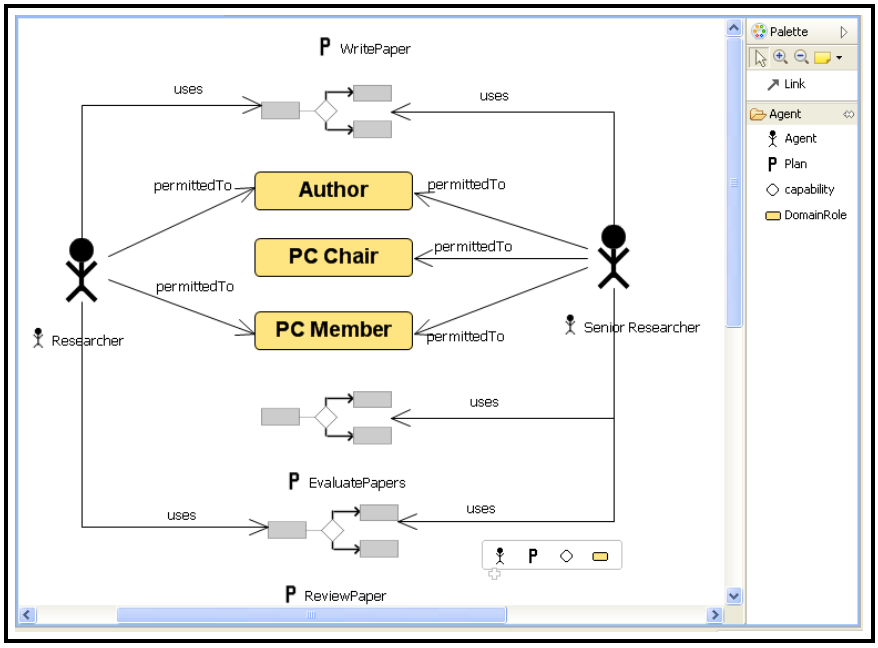


Fig. 5.2: The agent diagram of the CMS scenario.

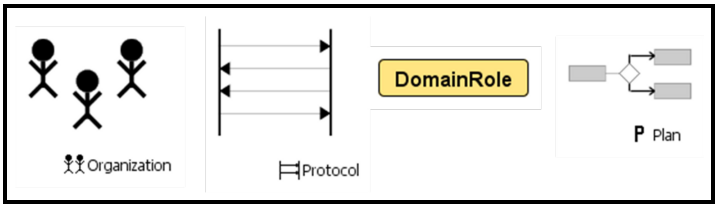


Fig. 5.3: The notation of the organization diagram. From left to right, the notations of organization, protocol, domain role, and plan are depicted.

agent can additionally use the *ReviewPaper* plan. How the body of a plan looks like is discussed in detail in Section 5.3.2.7.

5.3.2.3 Organization Diagram

Modeling Constructs of the Organization Diagram The organization diagram allows modeling the different kinds of organization types, their knowledge, plans, capabilities as well as domain roles either required or performed by the organizations. Moreover, the domain experts can additionally introduce agent interaction protocols to indicate how an organization and its members interact. Like in the case of the agent diagram, the domain expert can also make us of any domain role that is modeled outside the organization diagram. The notation of the organization diagram is presented in Fig. 5.3.

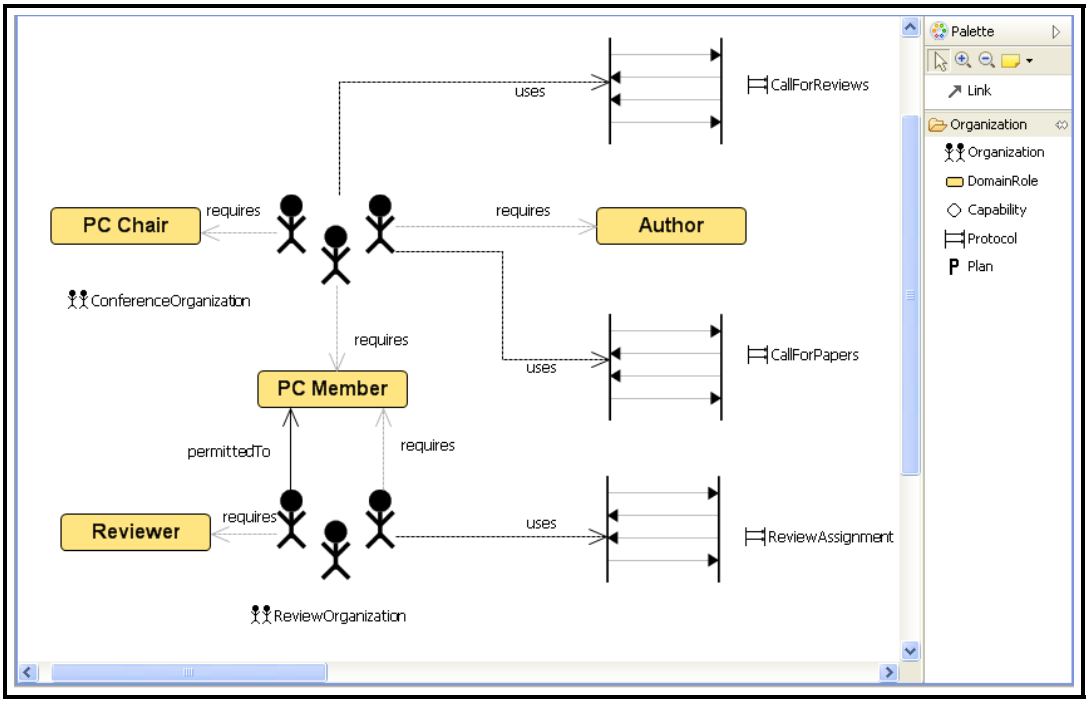


Fig. 5.4: The organization diagram of the CMS scenario.

Organization Diagram for CMS Fig. 5.4 depicts the *ConferenceOrganization* which is a generic organization type for conferences or workshops. It requires the domain roles representing the program committee chair (*PC Chair*), program committee members (*PC Member*), and *Author*. The two protocols *CallForPapers* and *CallForReview* specify how the organizational members separated into domain roles interact. The binding between domain roles of organizations and actors of protocols is specified as part of the collaboration diagram (see Section 5.3.2.4). Beside the *ConferenceOrganization* organization, we furthermore introduce the *ReviewOrganization* organization that is responsible for managing the preparation of reviews in the case that the *PC Member* outsources the review to external *Reviewers*. Hence, like the *Researcher* agent, the *ReviewOrganization* performs the domain role of the *PC Member*, but additionally requires the domain roles of *Reviewer* and *PC Member*. So, the *PC Member* domain role is performed but also required by the *ReviewOrganization*. The *ReviewAssignment* protocol is used to initiate the review process by outsourcing reviews to external reviewers.

5.3.2.4 Collaboration Diagram

Modeling Constructs of the Collaboration Diagram The collaboration diagram allows modeling the different kinds of collaborations inside an organization. Any collaboration is defined in accordance to the interaction they utilize and the bindings between the interaction's actors and the organization's domain roles. This is done in the collaboration diagram by firstly selecting the interactions this collaboration utilizes and secondly, defining the bindings (i.e. actor bindings (cf. Section 4.9.4)) between the organization's domain roles this collaboration makes use of and

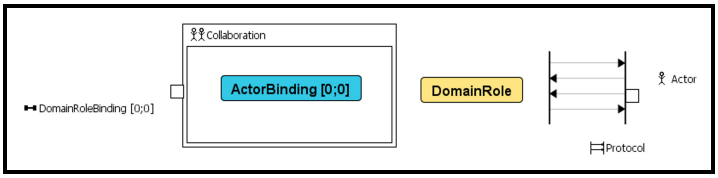


Fig. 5.5: The notation of the collaboration diagram. From left to right, the notations of domain role binding (illustrated as port), collaborations including actor bindings, domain roles, and protocols are depicted.

the actors of the selected interactions. The notation of the collaboration diagram is depicted in Fig. 5.5.

Collaboration Diagram for CMS Fig. 5.6 depicts the two collaborations *ReviewCollaboration* and *SubmissionCollaboration* of the *ConferenceOrganization*. The *SubmissionCollaboration* specifies the bindings *AuthorActor* and *ChairActor* of the *CallForPapers* protocol and the domain roles *Author* and *PC Chair* of *ConferenceOrganization*. The bindings are expressed through utilizing the actor bindings *AuthorActorAB* and *ChairActorAB* and the corresponding domain role bindings. Similarly, the *ReviewCollaboration* defines the bindings between the actors *Member* and *Chair* of the protocol *CallForReviews* and the domain roles *PC Member* and *PC Chair* of *ConferenceOrganization*. The numbers in squared brackets of the actor bindings express constraints for the number of role fillers that can be assigned at design time and filled during run-time. For example, the actor binding *AuthorActorAB* defines that at least one author must be bound. The maximum number is open expressed through the 0. For detailed information on these constraints, we refer to Section 4.9.3 and Section 4.9.4. As in the case of the domain role *PC Chair*, one domain role can be bound to several actors of different protocols even within the same collaboration. For example, actor bindings can be utilized to specify that one domain role is bound to an actor in the first protocol and to another actor in a second protocol. This implies whoever performs the domain role of that organization has to play the according actors in these bound protocols.

5.3.2.5 Role Diagram

Modeling Constructs of the Role Diagram The role diagram allows modeling of the different kinds of roles, either domain roles or actors defined within the interaction diagram, and how they relate to each other. The relationship between roles could be either of the form generalization, aggregation, or conflict. Moreover, the capabilities and knowledge a domain role has available can be designed within this diagram. Again, any capability, domain role, or plan that is modeled as part of this diagram is also represented in any other diagram that makes use of these concepts.

Role Diagram for CMS Fig. 5.7 shows the domain roles of the CMS scenario. The role diagram distinguishes between the domain roles *Author*, *PC Chair*, *Reviewer*, and *PC Member*, where *PC Member* is a specialization of *Reviewer*. The specialization relationship has the semantics (cf. Section 4.5.1) that any agent performing the *PC Member* role could also act as a *Reviewer*. Hence, the knowledge as well as capabilities are inherited. Apart from the generalization relationship, the aggregation relationship has the semantics that the domain role is composed of the aggregated domain roles. In the case of an organization, this means that the organization performs the domain role to the outside, but internally requires the aggregated roles.

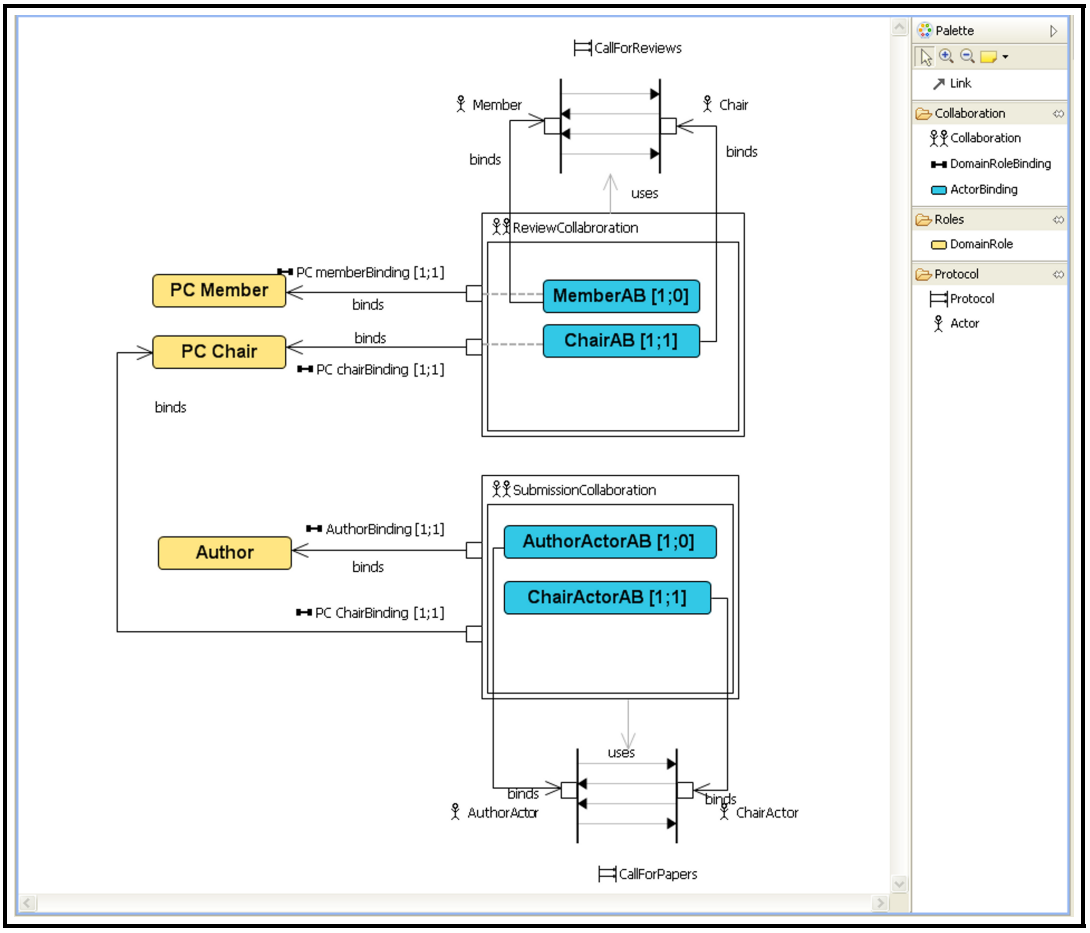


Fig. 5.6: The collaboration diagram of the CMS scenario.

Beside the relationships between the domain roles, as part of the role diagram, the domain expert can define which knowledge and capabilities are required and provided by the domain roles. In the CMS scenario, the *Author* domain role provides a *Submit Paper* capability including the *Submit Paper* plan, the *PC Chair* provides the *ManageConference* capability which includes the *SendCFPAction*. Finally, the *Reviewer* domain role provides the *Write Review* plan through the capability *Prepare Review*.

5.3.2.6 Interaction Diagram

Modeling Constructs of the Interaction Diagram The interaction diagram allows modeling of the actors part of an interaction and the ACL messages they exchange. In case of a protocol, the system designer can, moreover, define in which order these messages are exchanged. This is done by modeling the message flows and its message scopes along with their exchange modes. The notation of the interaction diagram is depicted in Fig. 5.8.

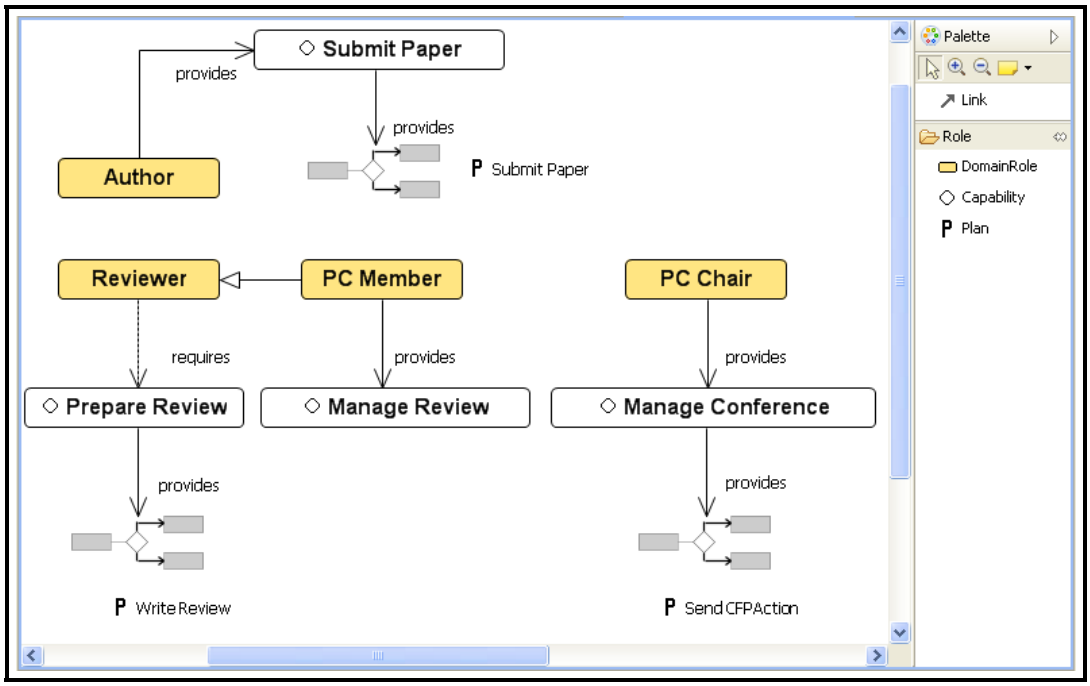


Fig. 5.7: The role diagram of the CMS scenario.

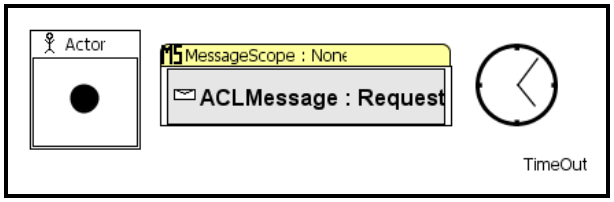


Fig. 5.8: The notation of the interaction diagram. From left to right, the notations of actor, including a message flow, message scope including an ACL message, and time out are depicted.

Interaction Diagram for CMS Fig. 5.9 shows the *CallForPapers* protocol of the CMS example. It describes the interaction between the *ChairActor* representing the program committee chair and the *AuthorActor* representing potential authors. The protocol is initiated by the *ChairActor* actor by sending the *CallForPaper* ACL message (performative *CFP*) to the *AuthorActor*. This means that the *CallForPaper* is sent to all candidates that are hidden (i.e. bound) behind this actor. There exists candidates that send a submission (represented by the *Submitter* actor) and candidates that do not submit (represented by the *Denier* actor). The protocol terminates for all role fillers of the *Denier* actor.

The *PaperDeadline* is the timeout between sending the *CallForPaper* message and receiving the submissions expressed by the *SubmitPaper* message. If the timer elapses, the *ChairActor* sends a *RejectPaper* message to all submitters that were rejected (represented by the *Rejected* actor) and an *AcceptPaper* message to all role fillers that were accepted (represented by the *Accepted* actor).

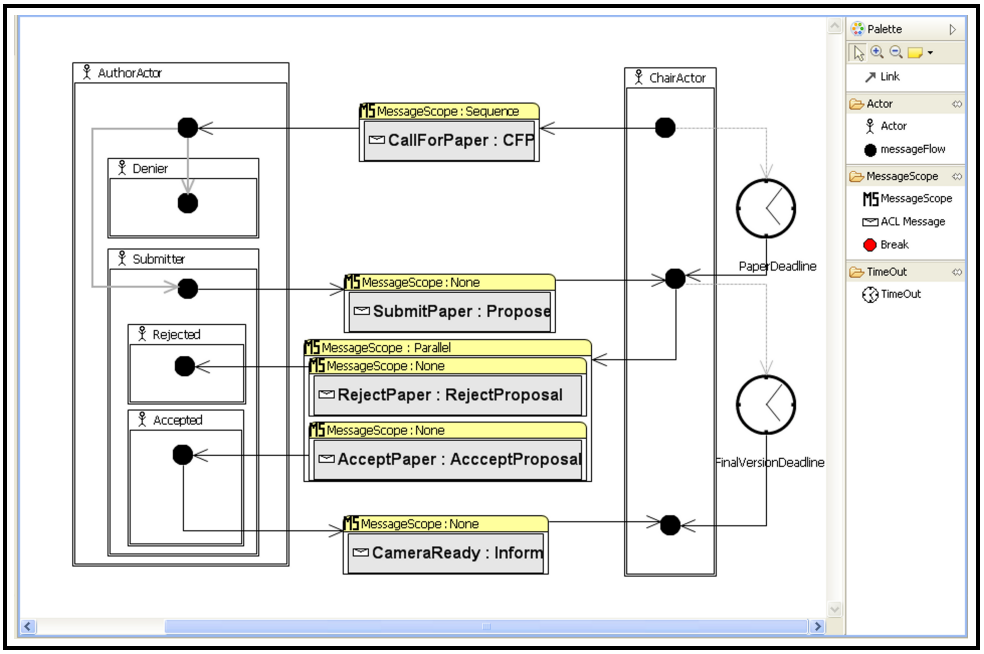


Fig. 5.9: The CallForPapers protocols of the CMS scenario.

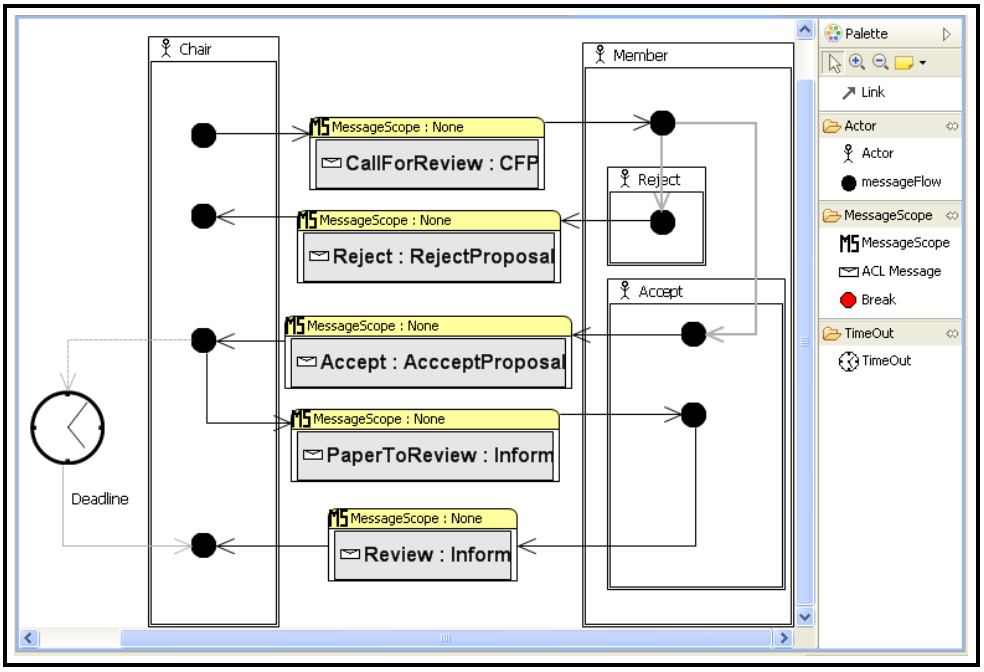


Fig. 5.10: The CallForReviews protocols of the CMS scenario.

All role fillers that receive an acceptance notification have to submit the camera ready version before the *FinalVersionDeadline* elapses.

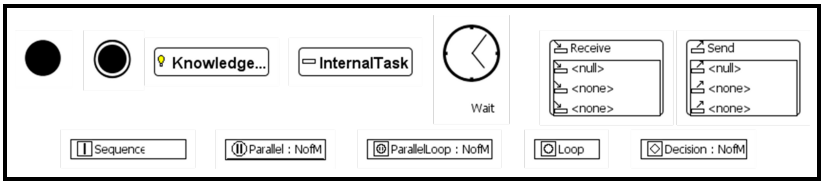


Fig. 5.11: The notation of the behavior diagram. From left to right, the upper row includes the notations of begin, end, knowledge, internal task, wait, receive, and send. The lower low presents the notations of sequence, parallel, parallel loop, loop, and decision.

Fig.5.10 illustrates the *CallForReviews* protocol of the CMS scenario. The purpose of this protocol is to assign submitted papers to reviewers. Therefore, it includes the actors *Chair* representing the program committee chair and *Member* representing the program committee members. The protocol starts with a *CallForReview* ACL message, which is either rejected by the *Reject* actor or accepted by the *Accept* actor which are both subactors of the *Member* actor. In the case of the latter, the *Chair* assign a paper to the *Accept* actor, who has to send the review back within the duration of the *Deadline* time out.

5.3.2.7 Behavior Diagram

Modeling Constructs of the Behavior Diagram The behavior diagram allows modeling of plans in a workflow-like manner, i.e. the different types of activities can be combined using control as well as information flows. To reduce the complexity of the plan's body, the system designer may introduce sub-plans that are used to hide information. Beside the process of a plan, moreover, the system designer can define the pre- and post-conditions of a plan in the properties view of the behavior diagram. The notation of the behavior diagram is depicted in Fig. 5.11.

Behavior Diagram for CMS Fig. 5.12 depicts the *SubmitPaper* plan, which is provided by the *Author* domain role. It implements the behavior of the *AuthorActor* actor and its sub-actors of the *CallForPapers* protocol from Fig. 5.9. In the first phase, the plan collects all *CallForPaper* messages in parallel. This is expressed through the parallel loop called *ReceiveRequest*. The task used to receive the message is called *ReceiveCFPAction* and part of *ReceiveRequest*. DSML4MAS distinguishes between ACL messages that are used to specify the message sequences of a protocol and the actual messages that are sent and received by plans. As protocols are reusable components, ACL messages do not specify the resources that are transmitted by them. If a message should be sent by a plan (e.g. the *CFPMessage* from Fig. 5.12), we have to introduce a new message that refers to an ACL message of a protocol (here the *CallForPaper*) and assign some application specific resources to it. In the CMS example, the program committee chair sends information about the conference, the deadlines, etc.

After receiving all *CallForPaper* messages, the agent has to decide whether to submit a paper or not. This is done in the *SubmitPaper?* decision. If the agent decides to submit a paper, at first, the called behavior is invoked that refers to the *WritePaper* plan. Otherwise, the plan terminates.

The internal task *WritePaper* is a kind of black box behavior that is not further refined at the model level. The behavior has to be implemented after generating the source code. For example, one could open a dialog box for the researcher to select the paper he/she wants to submit to the conference.

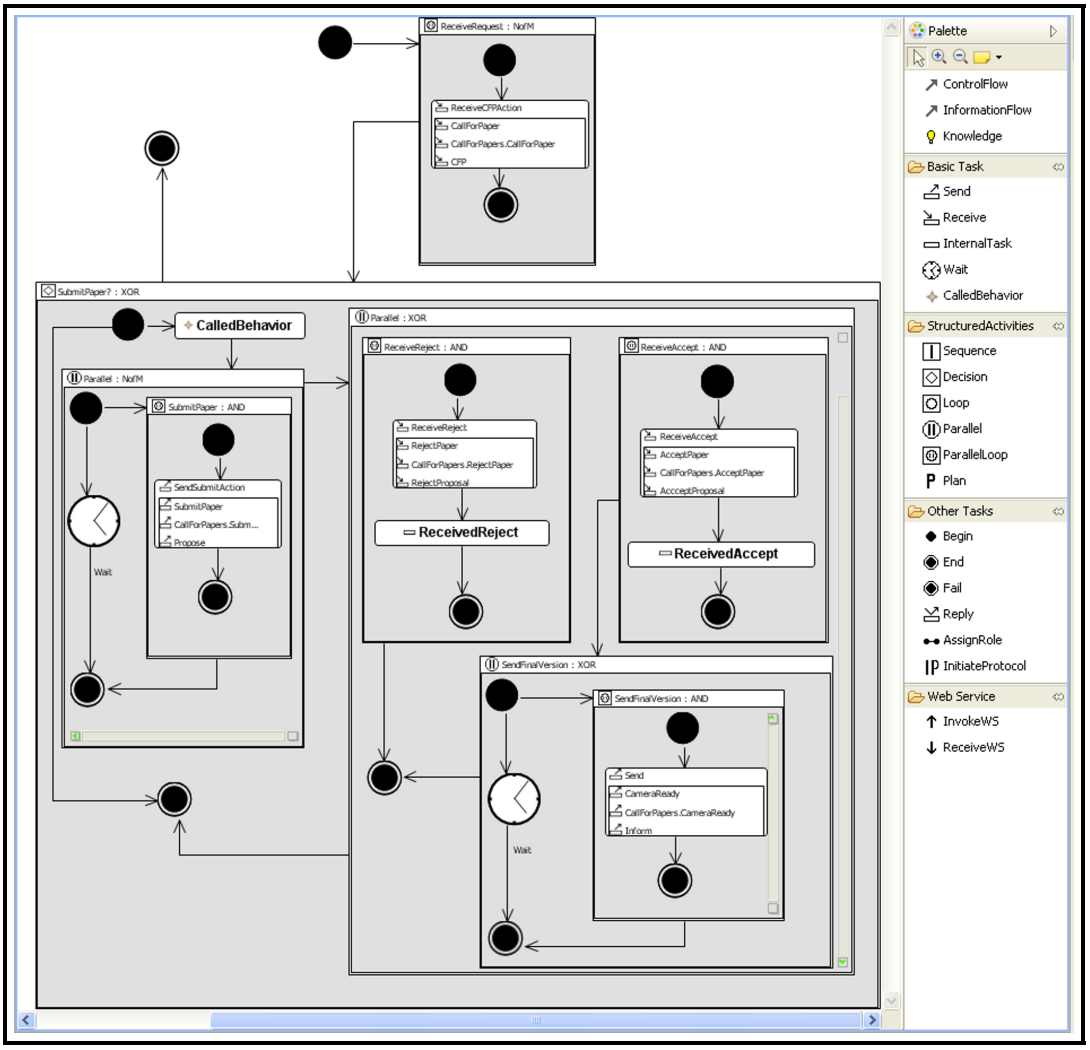


Fig. 5.12: Behavior diagram of the SubmitPaper behavior.

Either the author submits a paper or the plan terminates. If the author submits a paper, he/she has to wait for either a *RejectMessage* or an *AcceptMessage* from the program committee chair. The *ReceiveReject* and *ReceiveAccept* tasks are executed in parallel with XOR semantics, which can be set in the properties view of the parallel task. If the author receives an *AcceptMessage*, he/she has to finalize the paper (*FinalizePaper* task) and send it back to the program committee chair (*SendFinalVersion* task).

5.3.2.8 Environment Diagram

Modeling Constructs of the Environment Diagram The environment diagram allows modeling of any kind of either inside or outside the MAS existing resources that an agent may have access to

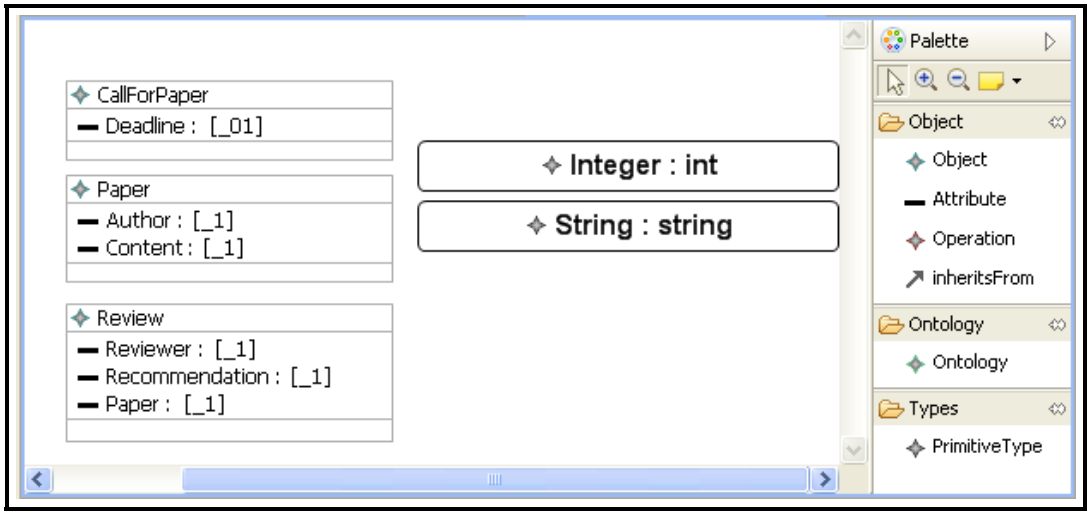


Fig. 5.13: The environment diagram of the CMS sceanrio.

achieve a certain task. The resources are modeled in an entity-relationship manner, where each resource is modeled as entity that can have relations to other entities.

Environment Diagram for CMS Fig. 5.13 depicts the environment diagram of the CMS scenario. It includes three objects, i.e. *CallForPaper*, *Paper* and *Review* used to store information. The *CallForPaper* object, for instance, is sent within the *CallForPaper* message of the *SubmitPaper* plan. The objects may have relations to other objects or to the primitive types *Integer* and *String*.

5.3.2.9 Deployment Diagram

The deployment diagram allows defining implementation specific information by offering means for modeling (i) the different agent instances that represent the run-time instances when executing the design and (ii) the way they are bound to organizations through the domain role binding and the membership concept. The assignment to organizations is done by linking the particular agent instance to the agent instance implementing the corresponding organization. By drawing this link, the designer is asked, which domain roles the member agent instance should perform in this organization instance context.

5.3.2.10 Deployment Diagram

Modeling Constructs of the Deployment Diagram The deployment view and its concepts are optional, i.e. concepts like *AgentInstance* do not need to be defined for executing the model transformations. However, there might be situations where the number of run-time instance is already known at design time in which case these instances can directly be defined using DSML4MAS. Details on how to use the deployment diagram are given be the process model of DSML4MAS in Section 5.4.2.2.

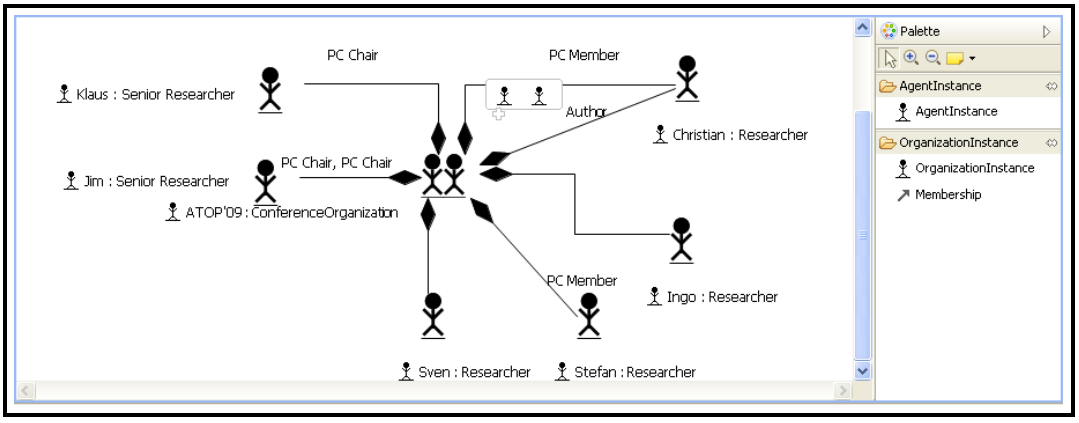


Fig. 5.14: The deployment diagram of the CMS scenario.

Deployment Diagram for CMS Fig. 5.14 shows the deployment diagram of the CMS example. We modeled an instance of the ConferenceOrganization, called *ATOP'09*. Furthermore, there are several agent instances, i.e., *Klaus* and *Jim* both of type *Senior Researcher* and *Christian* of type *Researcher*. The domain role an agent instance performs in an organization instance is specified by the membership concept which is visualized as a link.

5.3.2.11 Multiagent System Diagram

Modeling Constructs of the Multiagent System Diagram The multiagent system diagram gives an abstract overview of the whole MAS system by allowing to model (i) the different types of agents and organization part of the systems, (ii) the domain roles they perform or require, (iii) the messages that can be exchanged by the entities involved as well as (iv) the environments that can be accessed by the agents, organizations or roles. The abstract design made within the multiagent system diagram is also available in the more concrete agent diagram (cf. Section 5.3.2.2), organization diagram (cf. Section 5.3.2.3) and role diagram (cf. Section 5.3.2.5). The details on the different environments can be accessed on double-clicking the particular environments icons. The notation of the MAS diagram is depicted in Fig. 5.15.

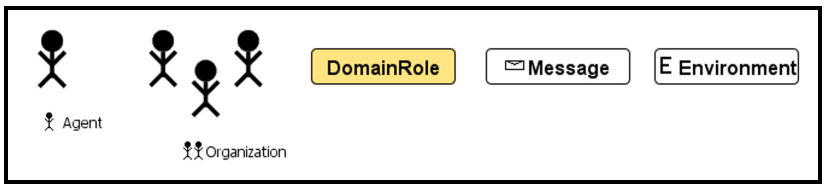


Fig. 5.15: The notation of the MAS diagram. From left to right, the notations of agent, organization, domain role, message, and environment are depicted.

Multiagent System Diagram for CMS Fig. 5.16 depicts the MAS diagram of CMS. It contains an overview on the present (i) organizations *ConferenceOrganization* and *ReviewOrganization*, (ii) the agents *Researcher* and *SeniorResearcher* and (iii) domain roles *Authors*, *PC Member*, *PC*

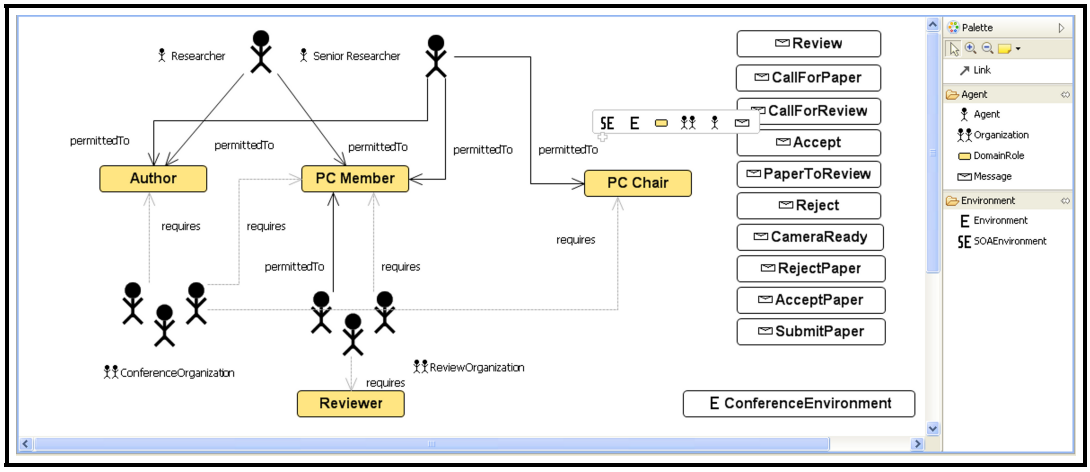


Fig. 5.16: The MAS diagram of the CMS scenario.

Chair, and *Reviewer* that are either performed or required by the agents and organizations, respectively. Moreover, it includes the environment *ConferenceEnvironment* and the messages (i.e. *Review*, *CallForPaper*, *CallForReview*, *Accept*, *PaperToReview*, *Reject*, *CameraReady*, *RejectPaper*, *AcceptPaper*, and *SubmitPaper*) that are exchanged between the entities involved.

5.3.3 Model Validation at Design-Time

Even if the graphical editor and the concrete syntax of DSML4MAS give the user an intuitive impression on how to use the language correctly, the generated artifacts are rarely complete with respect to the requirements that are needed for full code generation and the automatic execution. As argued before, a formal semantics increase the domain experts' understanding on how to model correctly in terms of ensuring that all requirements are met for full automatic code generation. Even when agent-based code is generated, mechanisms for full testing and validation are usually required, which consume a significant chunk of development effort. This effort can be decreased if the validation and testing facilities base on a formal semantics that can already be used at design time to test and validate the generated design.

In order to give the notation a clear semantics, which is one of Booch's key issues when developing an assistant notation, we make use of the formal semantics defined with Object-Z (cf. Chapter 4) in the graphical development environment of DDE to support validating and testing at design time. For this purpose, we manually transform the static semantics defined by the Object-Z specification into OCL statements that can be validated by the GMF environment at design time. How this manual transformation is done is described in (Roe et al.; 2003). This allows us to provide tool support for testing, evaluation, and execution of the designed artifacts. Furthermore, GMF provides different error levels and validation modes that further improve the mechanisms for testing and validating the design artifacts.

Error Levels For every rule in GMF, it is possible to specify an error message and an error level. There are three levels: `ERROR`, `WARNING`, and `INFO`. The according messages are displayed in the *Problems View* of Eclipse. Moreover, icons are displayed in the editor to highlight the elements that violate a constraint.

```
context PIM4Agents::Agent::Agent inv:
   self.behavior
      -> union(self.capability
         -> collect(c | c.behavior))
      -> union(self.performedRole
         -> collect(r | r.providesCapability
            -> collect(c | c.behavior)))
      -> size() > 0
```

Listing 5.1: Partial semantics of the agent view

Validation Mode Two validation modes exist for constraints. Live validation means that the constraints are always evaluated if something changes in the model. Manual validation, in contrast, means that the developer has to manually invoke the validation procedure.

5.3.3.1 Object Constraint Language

The Object Constraint Language (OCL) is a semi-formal constraint language that allows specifying constraints, which cannot be directly expressed within UML or metamodels. OCL gained much interest in the research community and industry due to the fact that OCL is one light-weighted formal method for object-oriented systems. Others are, for instance, Object-Z, Alloy, the Java Modeling Language (JML, (Gary T. Leavens et al.; 1999)). Tool support exists for both run-time checking of OCL specifications (e.g. (Demuth et al.; 2005)) as well as for static checking of metamodels against an OCL specification (e.g. USE (UML-based Specification Environment, (Gogolla et al.; 2007), Dresden OCL Toolkit, (Demuth; 2004)). OCL was advertised with the slogan *Mathematical Foundation, But No Mathematical Symbols* (Warmer and Kleppe; 2003) and is written using a concrete syntax that is inspired by object-oriented programming languages.

OCL can be used to constraint a UML model or metamodel in five different manners: A *constraint* defines a restriction on parts of the metamodel its models must conform to. An *invariant* gives a constraint that must always be met by all instances of a class. A *pre-condition* defines a constraint that must be true before the execution of an operation. Analogously, a *post-condition* defines a constraint that must be true after the execution of an operation. Finally, a *guard condition* specifies a constraint that must be true before a transition in a process model fires.

In addition to the capabilities of OCL to constraint and validate the design, it is also a building block of model transformation languages like ATL or QVT. There, OCL is used to explore the target and source models and to define the model mappings.

5.3.3.2 Partial OCL Specification of PIM4AGENTS

As illustrative example for demonstrating how to make use of the Object-Z specification, we select parts of the static OCL semantics of the agent view. Invariants I1 and I2 of the Object-Z specification of an agent require each agent to possess at least one behavior in order to act in an autonomous manner or to react to its environment. Listing 5.1 depicts the corresponding OCL constraint.

5.4 Process: DSML4MAS's (Semi-) Automatic Model-Driven Methodology

Several definitions of the term software process exist. In this dissertation, we mainly focus on software engineering processes. The probably most cited definition was given in (Humphrey; 1989). Humphrey (1989) defines a software engineering process "as the total set of software engineering activities needed to transform user's requirements into software". Hence, the process of a methodology normally includes steps of actions that are undertaken during the development, potentially by different users playing different roles in the complete software development process. Any software engineering process aims at (i) facilitating and supporting the development of high-quality software more quickly and at lower cost and (ii) analyzing, configuring, reusing, and executing the developed software. In the remainder of this section, we start by examining basic process models. Afterwards, built upon these categorizes, we present the model-driven software process of DSML4MAS.

5.4.1 Basic Process Models

To give the interested reader a basic understanding of process models, in the remainder of this section, basic approaches are explored that have been developed and applied in the (Agent-oriented) Software Engineering community in recent years.

5.4.1.1 Waterfall Model

The Waterfall model (Royce; 1987) is the probably simplest form of a process model, as the process itself is defined in a top-down manner, where the output of one step serves as the input of the next step. The Waterfall model distinguishes between five phases that are executed during the software development process: the analysis of the problem domain, designing the software system that solves the problem, coding the design into a particular programming language, testing the code with respect to the requirements, and finally the maintenance of the system.

Prominent AOSE methodologies based on the waterfall model are, for instance, Gaia (cf. Section 10.2.4), Prometheus (cf. Section 10.2.10), or AOR (cf. Section 10.2.2).

5.4.1.2 Spiral Model

The Spiral model (Boehm; 1986) defines an iterative development cycle in which inner cycles denote early system analysis and prototyping, and outer cycles represent the classic software life cycle. Each cycle consists of four phases: In the first phase the objectives are determined, followed by the second phase, which is responsible for evaluating the risks of the objectives. The third phase consists of the steps of developing and verifying and, finally, the fourth phase is used to evaluate and review the previous steps.

A prominent AOSE methodology based on the spiral model is MAS-CommonKADS (Peyravi and Taghyareh; 2007). Other well-known process models are, for example, the evolutionary and incremental process models of Tropos (cf. Section 10.2.7) or Ingenias (cf. Section 10.2.9).

Another process model category is the transformation model in which the software development process is considered as a sequence of steps that are (automatically) transferred into the next

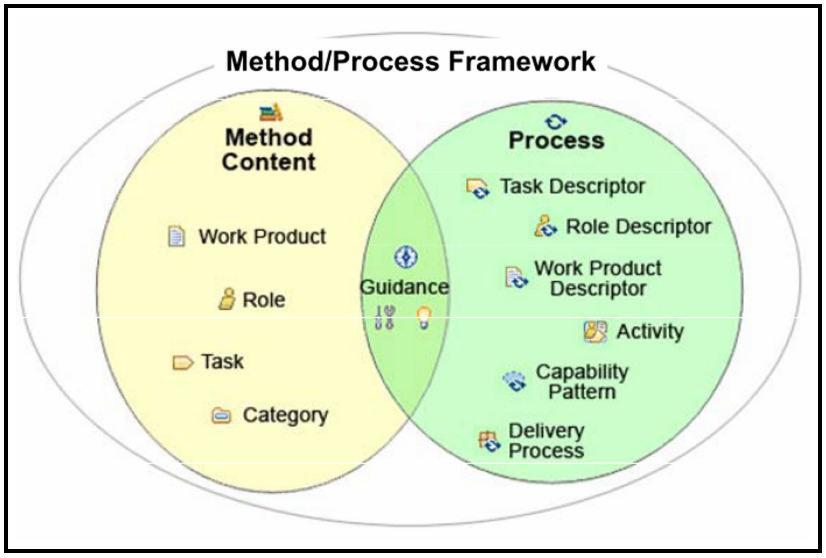


Fig. 5.17: The overall framework of EPF adopted from the EPF web-site.

sequent steps. An interactive version of this process model is used to formally describe the process model of DSML4MAS. To formally define the DSML4MAS methodology process, the Eclipse Process Framework is exploited.

5.4.2 Model-Driven Process Model for DSML4MAS

The main difference between a methodology and a pure modeling language is the methodology's process that defines in which manner the model should be designed. DSML4MAS can be used in both manners, either as pure modeling language or by applying the process provided by the methodology. In the former case, the application developers can freely choose in which order the different diagrams are completed. In the latter case, the semi-automatic process guides the design by automatically generating parts of the design.

5.4.2.1 Eclipse Process Framework

The Eclipse Process Framework (EPF[8]) is an open method engineering platform under the umbrella of Eclipse that provides an environment for process modeling. It implements existing OMG standards for MDD and is continuously developed with respect to newly emerging standards. In its current version (v. 1.5), EPF builds upon the Unified Method Architecture (UMA, (Ashbacher; 2008)), which is a UML metamodel that defines the well-established constructs related to software engineering and methodologies. EPF mainly follows two objectives:

- To provide an extensible framework and exemplary tools for software process engineering that include method and process authoring, library management, configuring and publishing of processes.

[8] http://www.eclipse.org/epf/

- To provide exemplary and extensible process content for a range of software development and management processes supporting iterative, agile, and incremental development, and applicable to a broad set of development platforms and applications.

As illustrated in Fig. 5.17, the conceptual structure of the EPF framework consists of the parts *Method Content* that contains the description of the static elements of engineering methodologies, and *Process* that defines the elements for assembling method content elements into smaller and larger methodologies that are defined in terms of processes. EPF defines four constructs for defining the Method Content:

- *Work Products* describe the object that results from conducting a guided procedure for methodological development; this can be an artifact or any other type of resulting object
- *Roles* describe the persons or positions that are involved in a software engineering process with respect to their capabilities and duties
- *Tasks* are the basic element for defining methodologies, providing a detailed guided procedure for creating a work product; a task defines the necessary development steps in form of natural language, and identifies the resulting work product as well as the involved roles
- *Categories* are a means to categorize the method content for specific application.

In EPF Processes, the Method Content is organized into processes that define the procedures for a software engineering process. Using descriptors for the method content, EPF distinguishes:

- *Capability Patterns* define partial processes that are reused in several methodologies for actual software engineering projects. They consists of tasks and defines a process for them.
- *Delivery Process* describes the actual methodologies for individual software engineering projects, consisting of tasks and capability patterns along with an overall process definition.

EPF implements the Software Process Engineering Metamodel (SPEM 2.0, (Object Management Group; 2008c)), which typical defines concepts of a process (process, phase, role, model, etc) that can be used to construct models that describe software engineering process in general and model-driven methodology processes like in the case of DSML4MAS in particular. The SPEM 2.0 metamodel has been approved in April 2008 as a formal specification of the OMG.

5.4.2.2 DSML4MAS Process

In DSML4MAS, we distinguish between several artifacts that are either automatically or manually produced during the phases of the semi-automatic process. These work products include the PIM4AGENTS, JackMM, and JadeMM models that are used as input and output models for the various model transformations along with the different diagram types of PIM4AGENTS (i.e. agent diagram, organization diagram, collaboration diagram, role diagram, interaction diagram, behavior diagram, environment diagram, and deployment diagram) presented in Section 5.3.2.1. These diagrams types are instantiated during the DSML4MAS process phases, which are all modeled as capability patterns.

The (semi-) automatic process proposed for DSML4MAS is illustrated in Fig. 5.18. The process starts with the *analysis phase*, which results in the artifacts interaction diagram and environment diagram. The interaction diagram generated by the (business) analyst is used as input for the endogenous model transformation that produces initial role and behavior diagrams, further refined in the *architectural specification phase*. However, this only happens, if an interaction has been designed, otherwise, the *architectural specification phase* is directly performed, without executing the endogenous model transformation. After refining the roles, the multiagent system, the agent

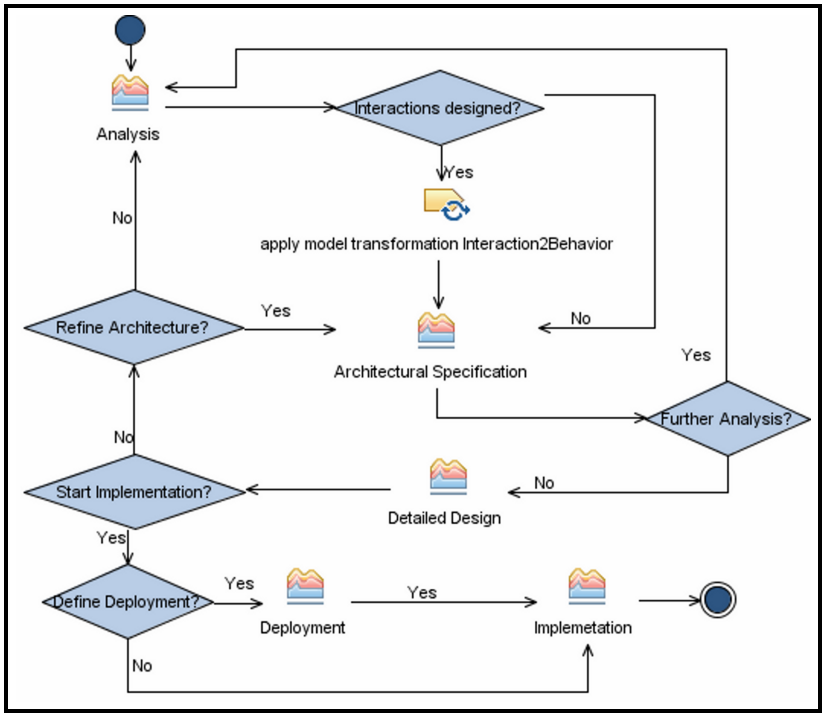
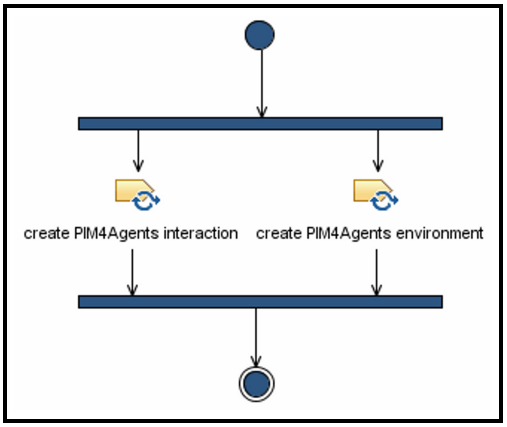


Fig. 5.18: The (semi-) automatic process of DSML4MAS.

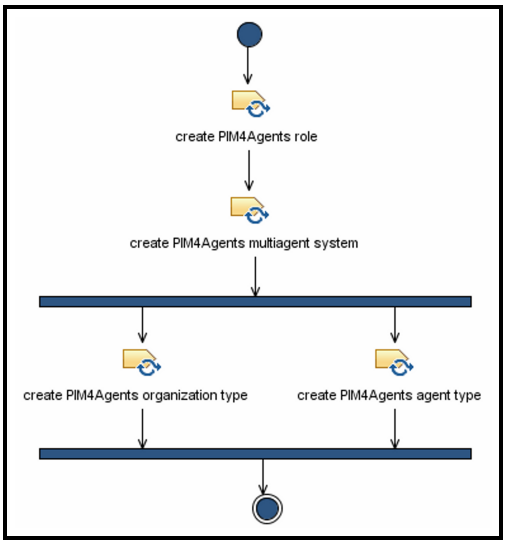
and organization diagrams, the DSML4MAS application developer has the option either to perform further analysis or to continue with the detailed design. In the latter case, the collaboration diagram is instantiated and the behavior diagram further refined based on the skeleton generated by the endogenous model transformation.

After the *detailed design phase*, the application developer could either refine the steps done in the *analysis phase* and *architectural specification phase* or start with the implementation. This could be done by the role of the programmer in two ways: The first option is to define a deployment of DSML4MAS in the *deployment phase*, the second option is to directly go to the *implementation phase* and do the deployment on one of the execution platforms. After deciding to start with the implementation, there is no option to go back to one of the previous phases. If the architecture or detailed design would change, this would mean that the designer has to jump into the particular phase again, with the option to skip phases, if the design requirements of a certain diagram have not changed.

Analysis phase The main objective of the analysis phase is to identify the actors involved in the software system and the interaction between them to manifest the relationships among them. This is done through the interaction diagram. Moreover, the analyst already details the kind of environment in terms of resources the actors (i.e. agents) may apply to meet their objectives. This is achieved by using the environment diagram. Based on the interaction defined, parts of the remaining phases can be generated in a semi-automatic manner by the endogenous transformation (see Chapter 6). This transformation works on the interaction and environment viewpoints and



(a) The details of the analysis phase



(b) The details of the architectural specification phase

Fig. 5.19: The EPF process of the *analysis phase* and *architectural specification phase*.

generates the messages part of the MAS, the domain roles of the architectural design phase, as well as parts of the internal behaviors of the detailed design phase.

The EPP process of the analysis phase is depicted in Fig. 5.19(a). The process includes the tasks `create PIM4Agents interaction` and `create PIM4Agents environment` that can be performed in parallel as they are independently in terms of input and output. An interaction (protocol) should be understood in this context as mechanism to define the abstract use case that will be refined in the further steps. The task `create PIM4Agents environment` is responsible for defining the resources (e.g. objects and services) available outside the MASs. How these resources are used by the agents and organizations will be indicated in the *architectural specification phase*.

Architectural specification phase The DSML4MAS *architectural specification phase* involves completing the MAS by modeling the agents and organizations. This is mainly achieved through the agent and organization diagram. Parts of the domain roles are automatically introduced in this stage by the endogenous transformation. Further domain roles can also be brought into the design and linked to the already existing ones through the *specializationOf* relationship (cf. Section 4.5.1). Also, any kind of resource a domain role may require and provide can be added in the role diagram. Domain roles can also be introduced on the organization and agent diagram.

The EPF process of the *architectural specification phase* is depicted in Fig. 5.19(b). This process includes the task `create PIM4Agents role`, `create PIM4Agents multiagent system`, `create PIM4Agents organization type`, and `create PIM4Agents agent type`. The organization and agent types are introduced in the multiagent system view and later on refined in parallel in the agent and organization diagram through the tasks `create PIM4Agents agent type` and `create PIM4Agents organization type`, respectively.

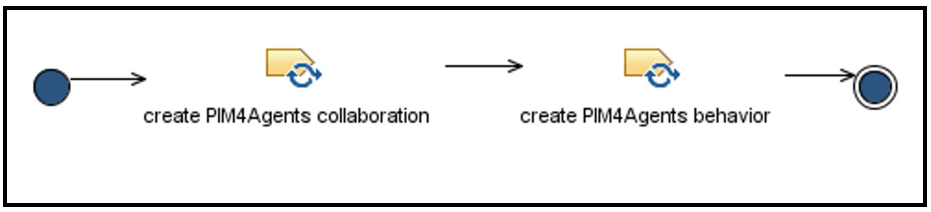


Fig. 5.20: The EPF process of the *detailed design phase*.

Detailed design phase Having finished the *architectural specification phase*, the developers can now move to the next step of DSML4MAS, the *detailed design phase*. At this stage, as part of the behavior diagram, the developers need to extend the generated plans with private information, but also need to specify new plans characterizing the internal behavior, which do not depend on any interaction. Apart from detailing the internal behaviors, the designer may specify which actors of the interaction defined in the analysis phase is played by which domain role in a certain organizational context. This is done through the collaboration diagram.

The EPF process of the *detailed design phase* is depicted in Fig. 5.20. This process includes the tasks `create PIM4Agents collaboration` and `create PIM4Agents behavior` that are realized in a sequential manner.

Deployment phase After the detailed design has been completed, the designer (i.e. programmer) may instantiate the agent instances that are involved in running system along with the bindings between them and the domain roles defined in the *architectural specification phase*. For this purpose, the developer establishes in the deployment diagram (i) the membership between agent instances and (organizational) agent instances and (ii) the domain role that is played in this organizational context.

Implementation phase When the design made with DSML4MAS is complete, as final step, the programmer may execute the model transformations integrated into DDE to JACK and JADE. This generates code, which can finally be refined within the underlying agent programming language. The details of this phase are discussed in Section 7.4.

5.5 Bottom Line

One of the main problems that prevent AOSE from a broad application in main stream software development is the lack of methodologies and suitable tool support. In this chapter, we demonstrated the methodology of DSML4MAS guiding the application developers through the different phases of the design, from requirements to implementation. For this purpose, we focused in this chapter on (i) the notation (i.e. concrete syntax) used to design with DSML4MAS in a graphical manner and the provided tool support and (ii) the model-driven process giving procedures for guiding the application developers from analysis to implementation.

To reduce the design complexity, the notation of DSML4MAS is split into different diagrams based on the viewpoints of DSML4MAS. The modeling constructs of each diagram were illustrated by using the well-known conference management system scenario. These diagrams are part

of different phases of the methodology process that includes the phases analysis, architectural specification, detailed design, deployment, and implementation. The process integrates three model transformations, i.e. the endogenous transformation supports the mapping between different views and diagrams, the vertical transformation allows transferring models between different abstraction levels, from platform-independent to more platform-specific models. The methodology process has been formalized using the Eclipse Process Framework.

Part III

Code Generation and Integration

6. Endogenous Transformation: From Interaction to Behaviors

Modern information systems are considered as collection of independent units that interact with each other through the exchange of messages. For coordination purposes, the interaction among agents is of particular importance in MASs. Agent interaction protocols (AIPs), as debated in Section 2.1.6, are one important mechanism to define agent-based interactions and hence play also a major role within DSML4MAS. In this chapter, we illustrate how to design AIPs with DSML4MAS and discuss a model-driven approach to use the protocol description as input to automatically generate corresponding agent behaviors implementing the global interaction. This endogenous model transformation is part of the model-driven process of the DSML4MAS methodology to automatically bridge the gap between the analysis and the detailed design phase of DSML4MAS.

Scope of this Chapter MASs define a powerful distributed computing model, enabling agents to cooperate with each other. Hence, the interaction between agents is considered as basic building block of MASs (see (Jennings; 2001)). An interaction is thereby considered as mechanism to express the dependence between agents (see Definition 2.1.6), where AIPs as a special case of interactions describe how messages are exchanged and thus focus on the global perspective of interactions between two or even more entities.

The importance of interactions in MAS is underlined by the fact that existing methods for designing MASs like Tropos (Susi et al.; 2005), Prometheus (Padgham et al.; 2007a), Gaia (Cernuzzi and Zambonelli; 2004), or INGENIAS (Pavón and Jorge; 2003) already include mechanisms to express AIPs. In particular, all of them use some sort of Agent UML diagrams (AUML) (Bauer and Odell; 2002). However, AUML in its current version has some drawbacks[1] that are intended to be resolved by the DSML4MAS approach.

Even if the design of the agents' interactions from a global perspectives seems to be the natural way, many popular agent programming languages and platforms do not even support modeling of this global perspective. Instead, for modeling the interaction between agents, they normally provide mechanisms to specify the interaction from the perspective of each entity involved. These *internal behaviors* normally contain information on which messages are exchanged, in which order these are exchanged and who is the receiver of them. Consequently, the information on the interaction is hard-coded in the internal behaviors and hence, cannot be used as global pattern. Even if the modeling of the global perspective is not supported by the most agent-based programming languages and platforms, we strongly believe that a combination of both approaches is necessary and suitable mechanisms are needed for combining them. Apart from skipping the global perspective and directly starting with the internal behaviors, DSML4MAS offers two options to combine both approaches when starting with a global perspective:

[1] We refer the interested reader to Section 10.2.1 to get further insights on AUML's pros and cons.

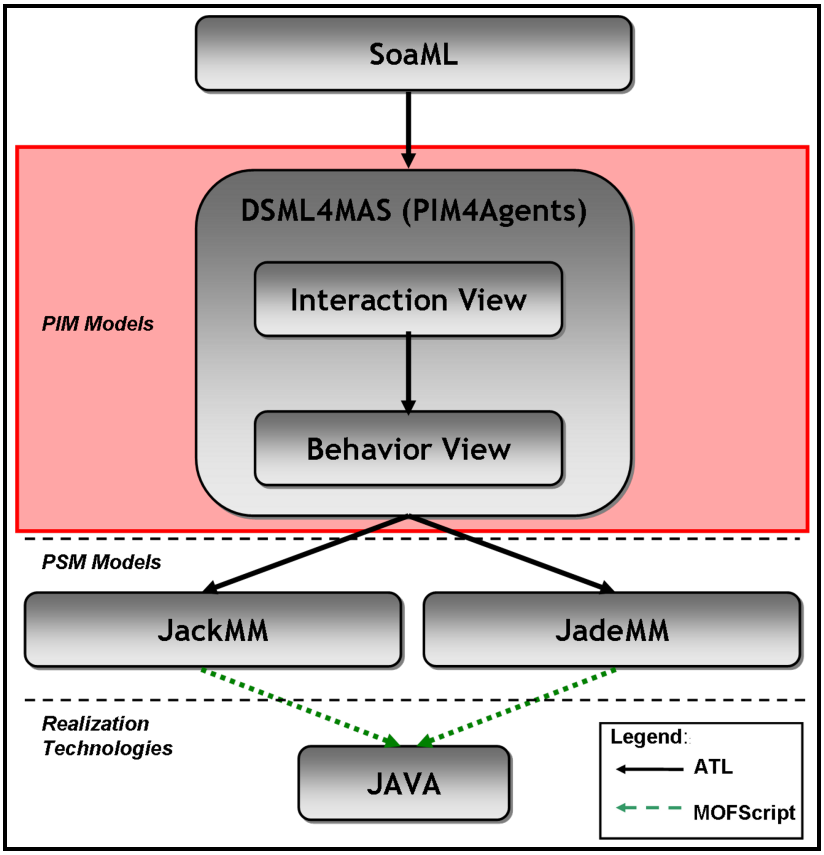


Fig. 6.1: Scope of this chapter: Endogenous model transformation within DSML4MAS.

- The first option is to model the particular internal behaviors by hand. In order to produce executable code, it must be ensured that the internal agents' behaviors conform to the AIPs of the involved actors. As previously introduced in Chapter 5, to support the user in keeping the design conform, constraints are specified to ensure that, for instance, the message order described by an AIP is implemented by the internal behaviors of the corresponding agents performing the particular protocol's actor. These constraints are expressed by Object-Z constraints, translated to OCL and integrated into the graphical editor to support the designer in checking the well-formedness of the design.
- The second option is to automatically generate partial internal behaviors on the input of the corresponding global interaction of the AIP. This can be achieved by defining an endogenous transformation between the interaction and behavior view of DSML4MAS.

Undoubtedly, the first option is more error-prone regarding harmonizing the interaction and behavior models. The second approach assures that the behavior conforms to the interaction model if the interaction metamodel can be transformed in an unambiguous manner. The critical and private information of an agent are then added to the generated plans.

However, especially in a more business-oriented context, developer often first define the manner in which the agents interact followed by defining the behaviors that actual implement the agreed interactions. Hence, in this chapter, we focus on the second option which is more

appropriate from a methodology point of view (cf. Fig. 6.1). Instead of checking whether the agent's internal behavior implements the protocol description, we focus on a protocol-driven approach that takes a protocol description as input and generates a corresponding behavior description that automatically conforms to the agreed AIP. In a second step, the system designer refines the behavior description by adding, for instance, private and critical information. Finally, in a last step, the generated design including the manually refined behaviors and the other views and diagrams is transformed to JACK code. This generated implementation can in combination with manually written code (if necessary) be executed and, hence, implements the interaction defined within the AIPs. Chapter 7 discusses the generation of JACK code.

Structure of this Chapter Section 6.1 illustrates how to design AIPs using DSML4MAS, followed by Section 6.2 giving a comparison between the DSML4MAS approach and the sate of art of general purpose languages for specifying global interactions. Section 6.3 presents the model transformation between the interaction view and the behavioral view of PIM4AGENTS. Finally, Section 6.4 summarizes the main achievements of this chapter.

6.1 Modeling Service Interaction Patterns using DSML4MAS

A lot of effort has been undertaken to identify the most common interaction scenarios from a business perspective, which have been published by Barros et al. (2005b) as Service Interaction Patterns. Design patterns, in general, capture the static and dynamic structures of solution that occur repeatedly when producing applications in a particular context (Schmidt; 1995). In order to demonstrate the strengths of the DSML4MAS approach, we take these patterns as a base and illustrate how to use the interaction view of PIM4AGENTS to fulfill the proposed requirements. For illustration purposes, DDE (cf. Section 5.2) is used for producing the AIPs. As these patterns were taken as benchmark for many existing special purpose languages for modeling interactions, they offer the nice opportunity to relate our framework of modeling AIPs with others.

To indicate the Service Interaction Patterns, Barros et al. consolidate recurrent scenarios and abstract them in a way that provides reusable knowledge. They distinguishes between four groups of patterns, i.e. *single-transmission bilateral patterns*, *single-transmission multilateral patterns*, *multi-transmission patterns*, and *routing patterns*. These four groups are precisely discussed in the following by using the DSML4MAS concrete syntax.

6.1.1 Single-transmission bilateral interaction patterns

The *single-transmission bilateral interaction patterns* category corresponds to elementary interactions, where a party sends (receives) a message, and as a result expects a reply (sends a reply). This group covers one-way (*send* and *receive*) and round-trip bilateral interactions (*send/receive*). As both, *send* and *receive*, are part of the *send/receive* pattern, which is again part of the *one-to-many send/receive* pattern, we focus on the latter which is discussed in Section 6.1.2.2.

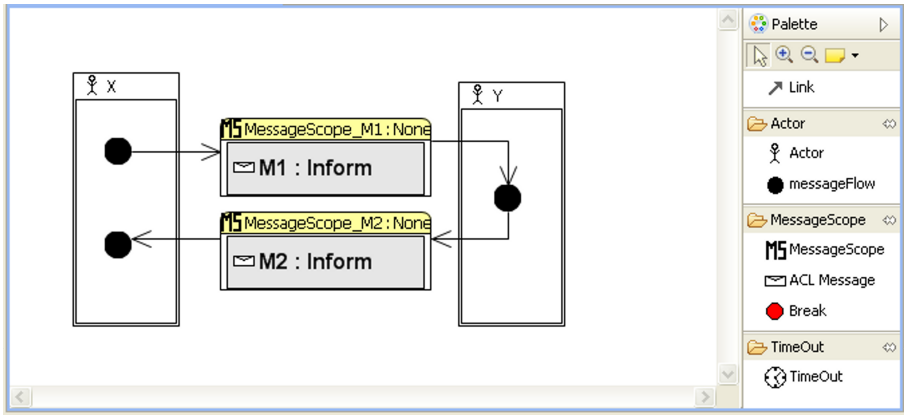


Fig. 6.2: Pattern 3: Send and Receive.

6.1.1.1 Pattern 3: Send/receive

Description A party X engages in two causally related interactions: in the first interaction party X sends a message to another party Y (the request), while in the second one party X receives a message from Y (the response) (Barros et al.; 2005b).

Realization Fig. 6.2 depicts this pattern using DSML4MAS. For this purpose, two actors (i.e. X and Y) are defined, where actor X sends a message M1 to actor Y that replies by sending message M2. As this patterns requires the interaction between exactly two entities, the both actors X and Y bind exactly one agent instance through the actor binding concept.

6.1.2 Single-transmission multilateral interaction patterns

The *single-transmission multilateral interaction patterns* category stays in the scope of non-routed patterns, but deals with multilateral interactions. This means that a party may send or receive multiple messages, but as part of different interaction threads dedicated to different parties. The patterns *one-to-many send* (Pattern 5) and *one-from-many receive* (Pattern 6) are not explicitly discussed in this evaluation, as both are part of the *one-to-many send/receive* pattern.

6.1.2.1 Pattern 4: Racing incoming messages

Description Party X expects to receive one message among a set of messages. These messages may be structurally different (i.e. different types) and may come from different categories of partners. The way a message is processed depends on its type and/or the category of partner from which it comes (Barros et al.; 2005b).

Realization Fig. 6.3 depicts the *racing incoming messages* pattern using DSML4MAS. Actor Y is divided into three subactors (i.e. Y1, Y2, and Y3). Each entity bound to one of the subactors either sends M1, M2, or M3. Even if the actors send different messages, all of them are sent by the particular agent instances in parallel. This is expressed through the highest message scope (i.e.

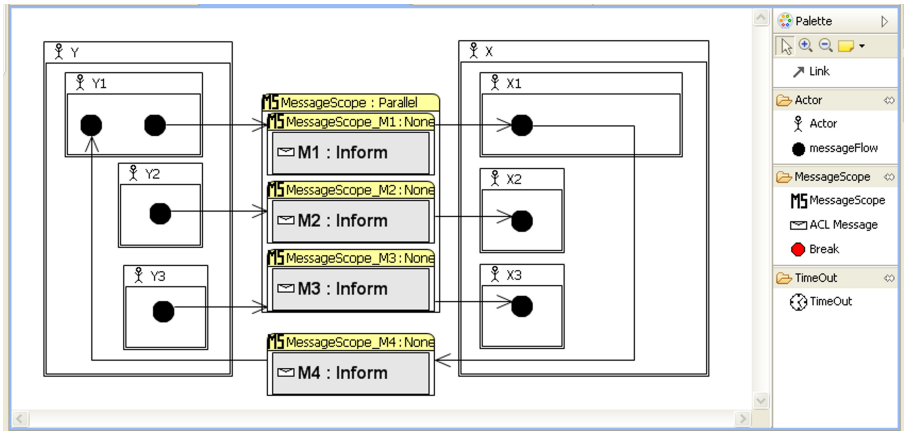


Fig. 6.3: Pattern 4: Racing incoming messages.

MessageScope:Parallel). Depending on a certain message or category of actor, a certain answer (i.e. M4) is sent back to the particular subactor or to all of them.

6.1.2.2 Pattern 7: One-to-many send/receive

Description Party X sends a request message to several other parties Y1,...,Yn, which may all be identical or logically related. Responses are expected within a given timeframe. However, some responses may not arrive within the timeframe. The interaction may complete successfully or not depending on the set of responses gathered (Barros et al.; 2005b).

Realization Fig. 6.4 depicts the one-to-many send/receive pattern using DSML4MAS. The parties are again modeled as actors, where this time, several agent instances Y1,...,Yn are bound to actor Y. Sending a message to an actor means that the particular message is sent to each entity bound to the target actor. This means that the ACL message M1 is sent to each of the Y1,...,Yn agent instances in parallel. When receiving M1, each of them sends the corresponding answer ACL message M2 to actor X. A timeout called TimeOut ensures that the interaction does not end up in a deadlock. If this timeout is raised, the interaction continues with the message flow specified by the timeout's *messageFlow* reference. However, the default message flow—if the *messageFlow* reference is empty—is the next message flow in the row. The *messageFlow* reference is illustrated in detail in Section 6.1.3.2.

6.1.3 Multi-transmission interaction patterns

This *multi-transmission interaction patterns* category corresponds to interactions where a party sends (receives) more than one message to (from) the same party.

6.1.3.1 Pattern 8: Multi-responses

Description A party X sends a request to another party Y. Subsequently, X receives any number of responses from Y until no further responses are required. The trigger of no further responses

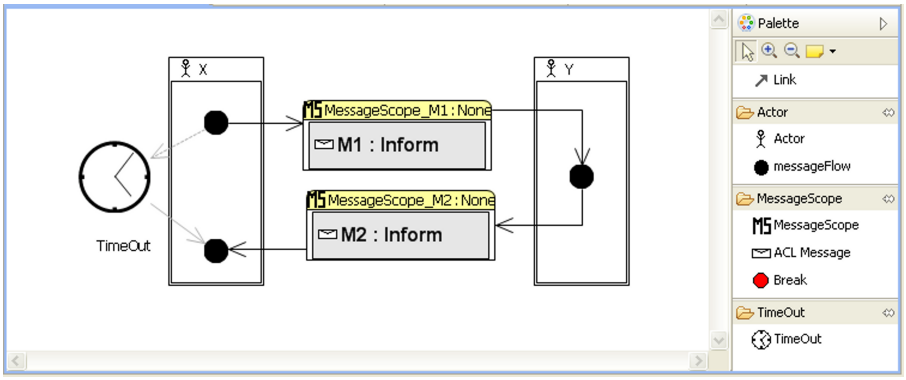


Fig. 6.4: Pattern 7: One-to-many send/receive.

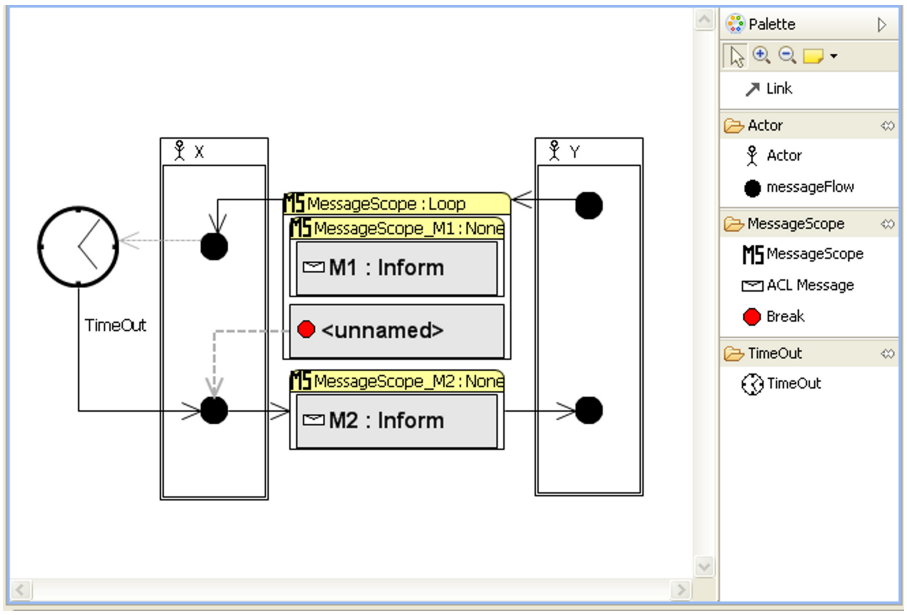


Fig. 6.5: Pattern 8: Multi-responses.

can arise from a temporal condition or message content, and can arise from either X or YŠs side. Responses are no longer expected from Y after one or a combination of the following events: (i) X sends a notification to stop, (ii) a relative or absolute deadline indicated by X, (iii) an interval of inactivity during which X does not receive any response from Y, or (iv) a message from Y indicating to X that no further responses will follow. From this point on, no further messages from Y will be accepted by X (Barros et al.; 2005b).

Realization A general visualization of this pattern using DSML4MAS is given in Fig. 6.5. The depicted AIP starts by Y sending a couple of requests to X, which is expressed through the literal Loop as exchange mode. The message M1 is sent to actor X until either (i) the TimeOut is raised resulting in continuing the interaction where X sends message M2 to Y, (ii) X sends a notification

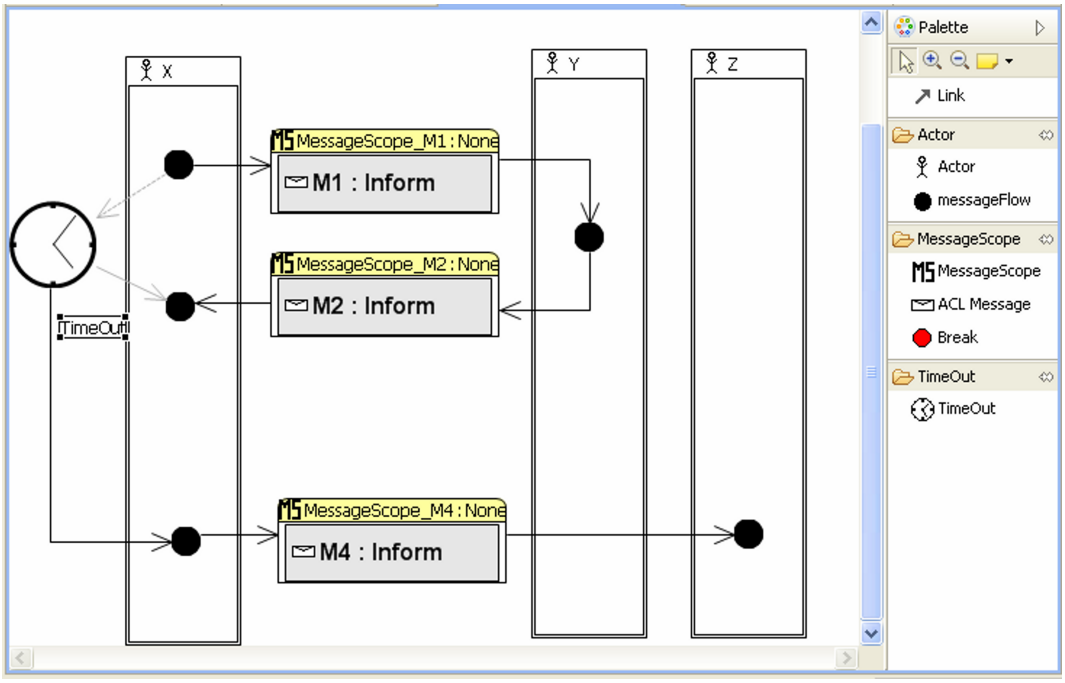


Fig. 6.6: Pattern 9: Contingent requests.

message M2 to stop this conversation, or (iii) Y sends a notification message M1 that breaks the Loop to stop this conversation.

6.1.3.2 Pattern 9: Contingent requests

Description A party X makes a request to another party Y. If X does not receive a response within a certain timeframe, X alternatively sends a request to another party Z, and so on (Barros et al.; 2005b).

Realization Fig. 6.6 illustrates the *contingent requests* pattern designed by DSML4MAS. Overall, three actors (i.e. X, Y, and Z) are defined, where X sends a request message M1 to Y. If Y does not respond in a timely manner by sending message M2, the TimeOut is raised and actor X sends a request message M4 to actor Y. When modeling this pattern in this manner, each actor consists of exactly one agent instance, i.e. the min and max values of the actor binding is set to 1.

The just presented design of Pattern 9 is rather static as each request to another third party (e.g. Z) needs to be explicitly modeled. In the following, an alternative representation of Pattern 9 is discussed, which is depicted in Fig. 6.7. Therefore, actor Y contains a set of entities that could be, in principle, requested, although, in each iteration only one entity is finally asked. To represent this, we introduce a subactor Y_Selected (where Y is the superactor containing all agent instances) that needs to fill one entity at minimum and maximum. Additionally, we introduce an actor Y_NotSelected that contains all entities that initially have not been selected. This Y_NotSelected is further divided into three actors, i.e. Y_Selected, Y_NotSelected, and Y_Refused, where the last

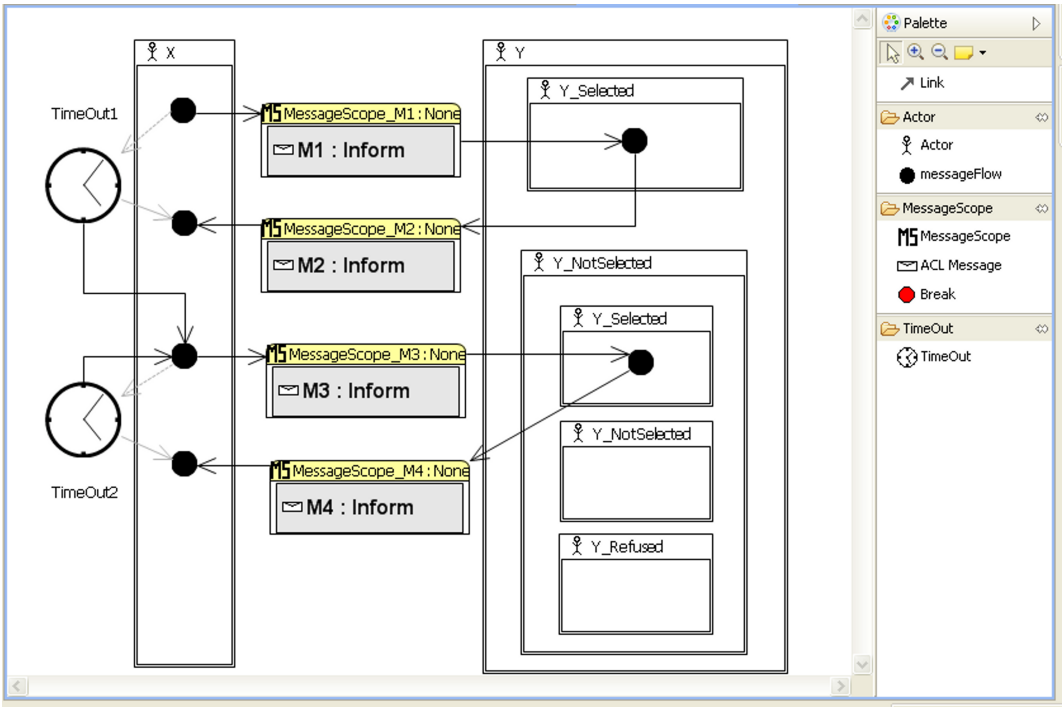


Fig. 6.7: Pattern 9: Alternative contingent requests.

one contains any agent instance that has not responded in a timely manner before the TimeOut2 has been raised. In this case, the interaction restarts at the third message flow, where a new agent instance is selected (i.e. removed from Y_notSelected and put into Y_selected). However, the assignment of agent instances needs to be defined in X's plan and cannot be defined in the AIP itself. TimeOut2 is defined as a relative value depending on the time when M3 is sent. For instance, we define the TimeOut2 as the actual time plus two minutes. Hence, each agent instance of Y_Selected has exactly two minutes for answering the request before TimeOut2 occurs.

6.1.3.3 Pattern 10: Atomic multicast notification

Description A party sends notifications to several parties such that a certain number of parties are required to accept the notification within a certain timeframe. For example, all parties or just one party are required to accept the notification. In general, the constraint for successful notification applies over a range between a minimum and maximum number (Barros et al.; 2005b).

Realization Fig. 6.8 depicts the *atomic multicast notification* pattern using DSML4MAS. Actor X sends a notification message Notification to actor Y that may consist of several agent instances. Some of these entities may either accept or reject this notification, which is expressed through the subactors Y_accept and Y_reject, both filled at run-time. To specify that a certain number of agent instances should accept the notification, the *min* and *max* attributes of the actor binding concept for Y_accepted can be set accordingly.

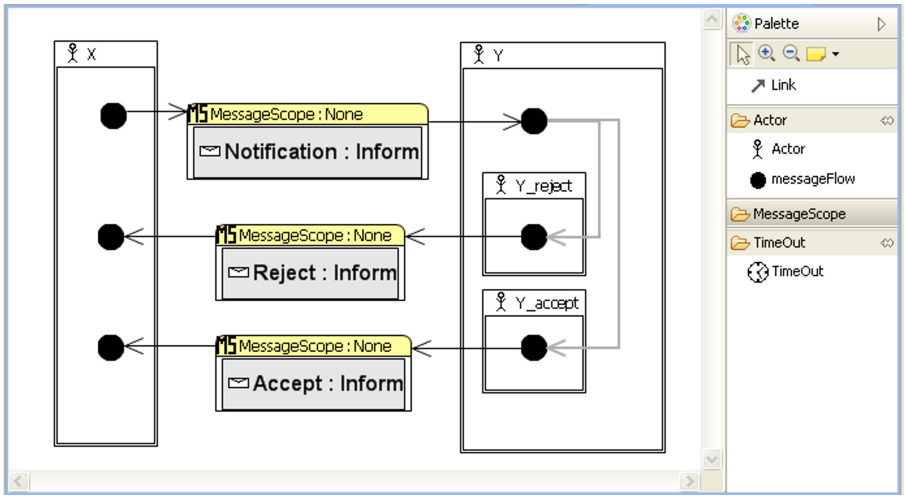


Fig. 6.8: Pattern 10: Atomic multicast notification.

6.1.4 Routing patterns

Routing patterns require a way to specify the intended receiver of a message, i.e. the receiver of a response is not necessarily the sender of the request.

6.1.4.1 Pattern 11: Request with referral

Description Party X sends a request to party Y indicating that any follow-up response should be sent to a number of other parties (Z1,...,Zn) depending on the evaluation of certain conditions. While faults are sent by default to these parties, they could alternatively be sent to another nominated party (which may be party X) (Barros et al.; 2005b).

Realization Fig. 6.9 depicts this pattern. Actor X sends a request message M1 to actor Y. If a fault occurs at the side of actor Y, it sends a error message M3 to X, otherwise Y requests actor Z. Afterward, actor Z informs actor X by sending the ACL message M4.

6.1.4.2 Pattern 12: Relayed request

Description Party X makes a request to party Y, which delegates the request to other parties Z (consisting of Z1,...,Zn) that then continue interactions with party X while party Y observes a view of the interactions including faults. The interacting parties are aware of this view (as part of the condition to interact) (Barros et al.; 2005b).

Realization Fig. 6.10 depicts the this pattern modeled with DDE. Actor X sends a request message M1 to actor Y. This actor then forwards this request message M2 to actor Z. This actor then sends message M3 to X, additional, a copy is sent to Y in parallel.

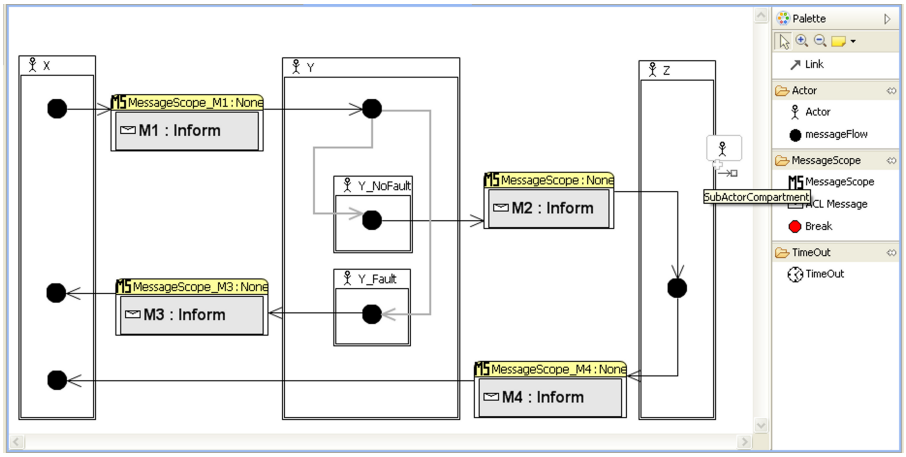


Fig. 6.9: Pattern 11: Request with referral.

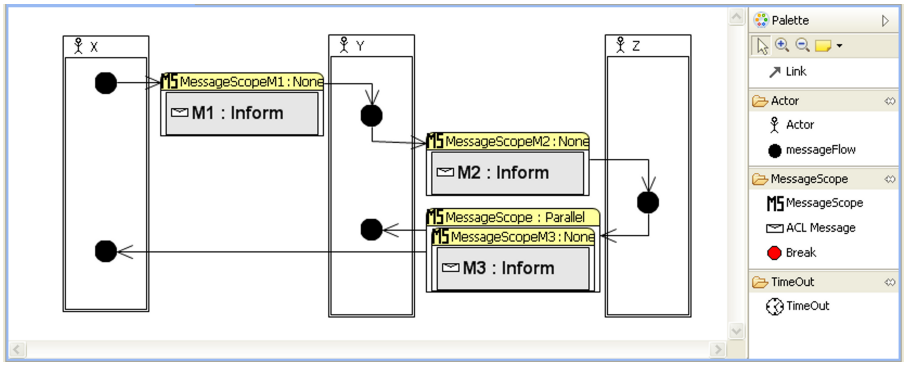


Fig. 6.10: Pattern 12: Relayed request.

6.1.4.3 Pattern 13: Dynamic routing

Description A request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request. When the parties that were issued the request have completed, the next set of parties are passed the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the intermediate steps (Barros et al.; 2005b).

Realization Fig. 6.11 depicts the dynamic routing pattern modeled with DSML4MAS. In this model, an actor X sends a request message M1 to actor Y and Z in parallel. Based on message M1, actor Y sends a M2 message to Z. Based on some predefining or dynamic conditions, Y either sends the M3 message to Y or the M4 message to actor X. To represent this choice in the model, we introduced two subactors of Z, i.e. Z_Y and Z_X. The agent instances bound to these subactors then send the corresponding message.

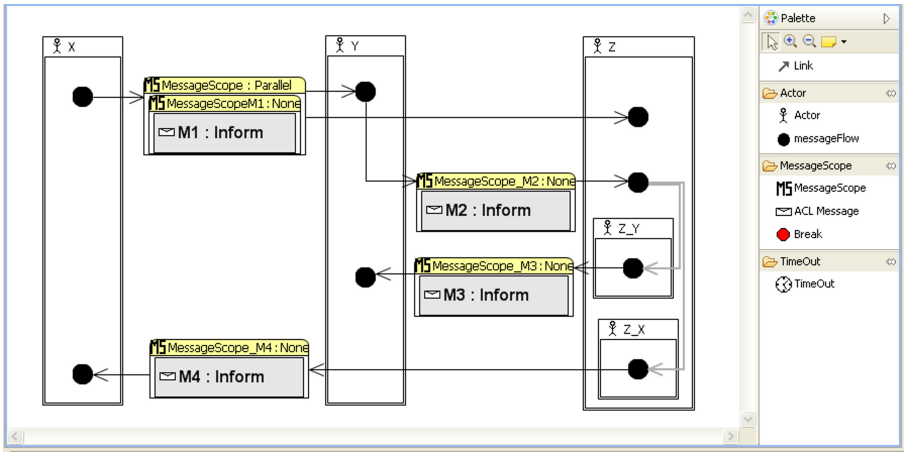


Fig. 6.11: Pattern 13: Dynamic Routing.

6.2 Comparison with the State of the Art

Special purpose languages for describing interactions from a global perspective have been subject of extensive research (e.g. Web Service Choreography Description Language (WS-CDL) (Kavantzas et al.; 2005)). In the following, some of them are discussed and related to the DSML4MAS approach. This evaluation is again based upon the Service Interaction Patterns proposed by Barros et al..

Extended Business Process Modeling Notation (BPMN) (Decker and Puhlmann; 2007) proposes several extensions to overcome the limitations of BPMN regarding its suitability for choreography modeling. Extended BPMN supports most of the patterns, however, it does not support *atomic multicast notification* and only partly supports the *contingent requests*.

WS-CDL (Kavantzas et al.; 2005) is an XML-based language focusing on interactions and their relationships. Interactions are bi-lateral and involve either one message (request-only or response-only) or two messages (request-response). WS-CDL supports most of the service interaction patters, however, patterns where the participating entities are only known at run-time (i.e. *one-to-many send*, *one-to-many send/receive*, and *contingent request*) are not directly supported. However, the main criticism of WS-CDL is that the integration with BPEL4WS (Louridas; 2008)—the corresponding language to define interactions from a local perspective—can only hardly be achieved (see (Barros et al.; 2005a) for more details).

Let's Dance (Zaha et al.; 2006a) is a visual choreography language that is not tied to any particular execution technology. Let's Dance supports mainly all service interaction patterns, however, only one timeout can be specified, which makes the support of the *contingent requests* pattern difficult. In (Zaha et al.; 2006b), an approach to generate local models for each actor that is participating in a choreography is given. Currently, a model transformation between the local models and BPEL4WS is developed to automatically generate BPEL4WS code templates.

BPEL4Chor (Decker et al.; 2007) is an extension of BPEL4WS to shift it from the orchestration (i.e. local) to the choreography (global) layer. It directly supports the service interaction patterns except the *atomic multicast notification* pattern. Furthermore, as BPEL4Chor mainly bases on BPEL4WS an integration with the latter is possible.

Pattern	WS-CDL	ext. BPMN	Let's Dance	BPEL4Chor	Dsml4Mas
1	+	+	+	+	+
2	+	+	+	+	+
3	+	+	+	+	+
4	+	+	+	+	+
5	+/-	+	+	+	+
6	+	+	+	+	+
7	+/-	+	+	+	+
8	+	+	+	+	+
9	+/-	+/-	+/-	+	+/-
10	-	-	-	-	+
11	+	+	+	+	+
12	+	+	+	+	+
13	?	?	?	+	+

Tab. 6.1: Service interaction pattern support in WS-CDL, extended BPMN, Let's Dance, and DSML4MAS. Dynamic routing is not considered in the assessment of WS-CDL, extended BPMN, and Let's Dance.

In the MAS community, Agent UML (AUML) is the most prominent modeling language for specifying AIPs. AUML is an extension of UML to overcome the limitations of UML with respect to MAS development. AUML results from the cooperation between the OMG and FIPA, aiming to increase acceptance of agent technology in industry. In particular, AUML specifies AIPs by providing mechanisms to define agent roles, agent lifelines (interaction threads, which can split into several lifelines and merge at some subsequent points using connectors like AND, OR or XOR), nested and interleaved protocols (patterns of interaction that can be reused with guards and constraints), and extended semantics for UML messages (for instance, to indicate the associated communicative act, and whether messages are synchronous or not). However, AUML does not allow to express more specialized subactors. For this purpose, in (Haugen; 2008), Haugen suggested two improvements to the UML 2.0 sequence diagram notation in order to define multicast messages and combined-fragment iterators over subsets. This approach share several commonalities with our approach in DSML4MAS. However, the suggested improvements are not part of the recent version of AUML.

Even if AUML can be considered as de facto standard for modeling AIPs, tool supported is very limited. In (Winikoff; 2005), a textual notation and graphical tool have been presented. In (Ehrler and Cranefield; 2004), an approach is presented that automatically interprets AUML AIPs. However, the resulting tool called Paul (Plug-in for Agent UML Linking) only supports parts of AUML as only the alternative operator is implemented. Furthermore, the code generation is limited to two agent lifelines and it is pretty unclear if multicast messages are supported. However, AUML does not allow to specify the sending of multiple messages as needed in the case of Pattern 7 *one-to-many send/receive*.

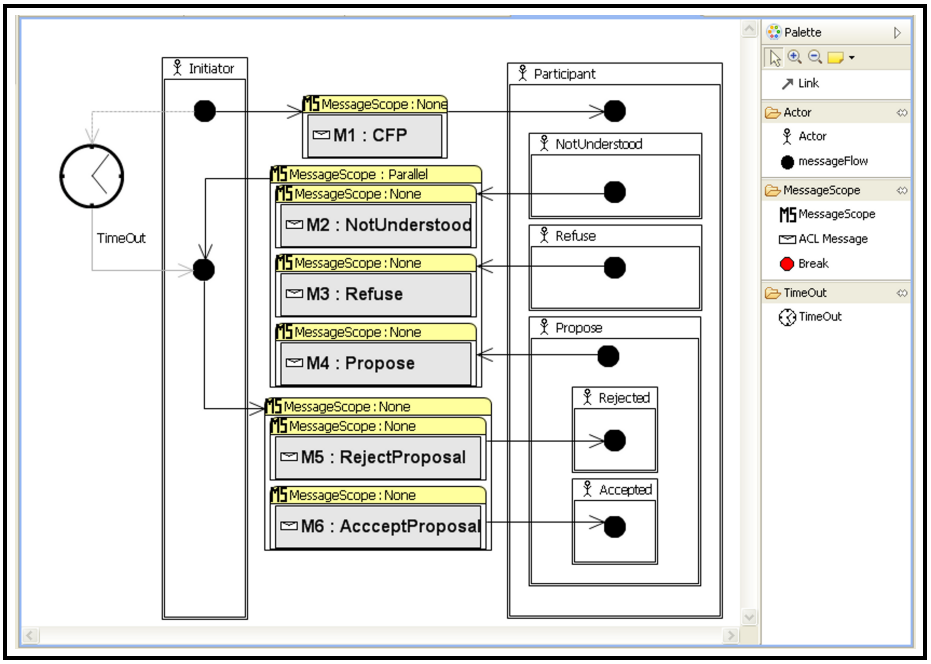


Fig. 6.12: The Contract Net Protocol designed using DSML4MAS.

6.3 From Agent Interaction Protocols to Behavior Descriptions

After discussing how to use DSML4MAS for designing AIPs, we now demonstrate as part of this section how to transform AIPs to internal behaviors. Therefore, we use the contract net protocol (CNP, (Smith; 1988)) as example and present how the model transformation (cf. Section 6.3.2) transfers the global description of CNP to internal behaviors.

6.3.1 Illustrative Example: Contract Net Protocol

CNP belongs to the family of cooperation protocols and is the most prominent protocol in DAI as it provides a solution for the connection problem, i.e., to find an appropriate agent to work on a given task. It bases on the contracting mechanism used by business to govern the exchange of goods and services and though uses a minimum of messages which makes it very efficient for task assignment. Although CNP is one of the most used agent interaction protocols, there are some limitations: A task may be awarded to a participant with limited capability if a better qualified participant is busy right now. So there exists only a suboptimal solution. There might also be reasons, why the initiator does not receive any bids. In (Knabe et al.; 2002), an extended version of CNP is presented to overcome these limitations.

CNP defines how an initiator sends out a number of calls for proposal to a set of participants. Depending on their free capacities, some of these participants will refuse the call, while others may come up with a proposal. After the answers have been received from every participant or a deadline is reached, the initiator evaluates the proposals and awards to contract to these sending

the most adequate bid(s), the others were rejected. After termination of the task the awarded participant reports their results to the initiator.

For designing CNP using DSML4MAS, firstly, we introduce two actors called Initiator and Participant. The protocol starts with the first message flow of the Initiator that is responsible for sending the ACL message M1 of the performative type CFP. The M1 message specifies the task as well as the conditions that can be specified within the properties view of the graphical editor. When receiving M1, each agent instance performing the Participant decides on the base of free resources whether to propose by sending M4 of performative type Propose or to refuse by sending M3 of performative type Refuse. The third option is to send M2 of performative type NotUnderstood in case either the content or the ontology of M1 could not be understood.

How the decision on free resource is implemented cannot be expressed in the protocol description, as private information are later on manually added to the automatically generated behavior description. To distinguish between the alternatives, three additional actors (i.e. NotUndestood, Propose and Refuse) are defined that are subactors of the Participant (i.e. any agent instance performing the Participant should either perform the NotUndestood, Propose or Refuse actor). The transitions between the message flow of the Participant and the message flows of its subactors through the *messageFlow* reference underline the change of state for the agent instance performing the Participant actor. However, the transitions are only triggered if a certain criterion is met. In the CNP case, the criterion is that the Initiator got all replies, independent of its type (i.e. M2, M3 or M2). The *postConditions* of a message flow can be defined in the properties view of the graphical editor. The message flows within the NotUnderstood, Refuse and Propose actors are then responsible for sending the particular messages (i.e. M2, M3 and M4).

After the deadline expired (defined by the TimeOut) or all answers sent by the agent instances performing the Participant actor are received, the Initiator evaluates the proposals in accordance to a certain selection function, chooses the best bid(s) and finally assigns the actors Accepted and Rejected accordingly. Again, the selection function is not part of the protocol, but can be defined later on in the corresponding plan. Both, the Accepted and Rejected actors, are again subactors of the Propose actor. The Initiator sends the message M6 with the performative type AcceptProposal to the Accepted actor and a message M5 with the performative type RejectProposal message to the Rejected actor in parallel.

6.3.2 Model-to-Model Transformation: From Interactions to Behaviors

In the following, the most important mapping rules—visualized in Fig. 6.13—for transferring AIPs to internal behaviors in DSML4MAS are discussed. The general idea of this model transformation is to initiate one plan for each actor not contained by any other actor inside an AIP. The generated plan expresses how the exchange of messages is proceed from the perspective of the individual actor and hence for any agent instance bound through the concepts of actor bindings (cf. Section 4.9.4) and domain role bindings (cf. Section 4.9.3).

The first mapping rule, generates a plan for each actor of the input protocol. The message flows this actors and its sub-actors are active are transformed to actions in the plan responsible for sending and receiving ACL messages. The details of the mapping between actor and plan is depicted by Mapping Rule 6.1.

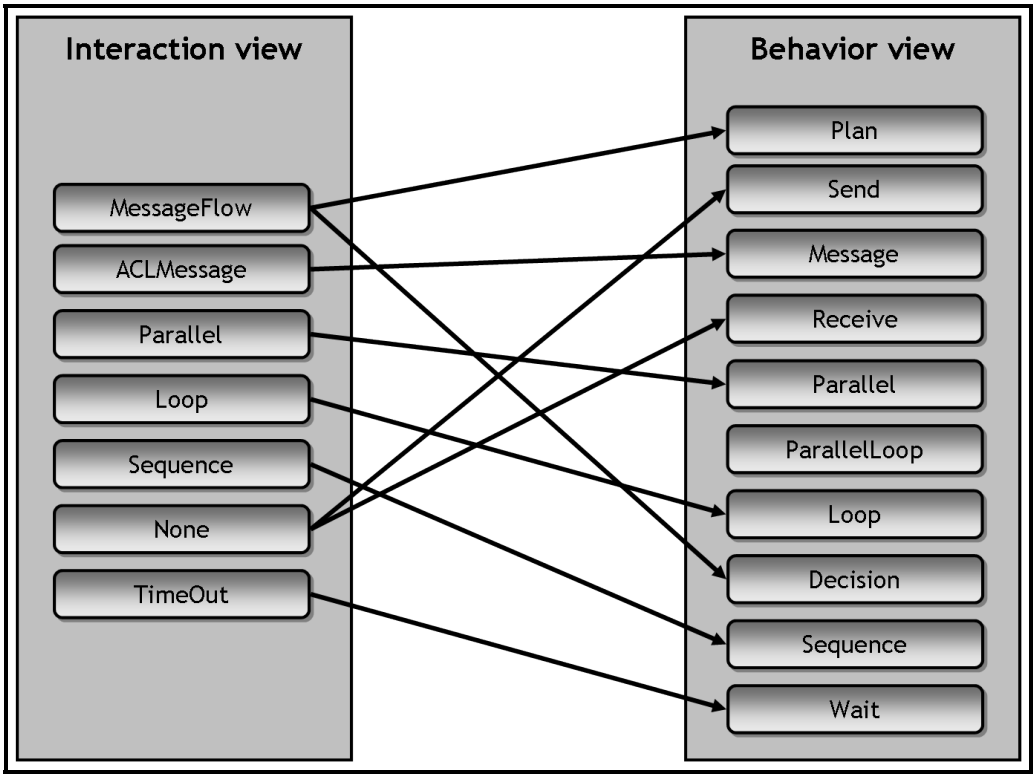


Fig. 6.13: Conceptual model transformations between the interaction view and behavioral view of DSML4MAS.

Mapping Rule 6.1: Actor → Plan

- **name**: name of the *Actor's Protocol* plus the name of the particular *Actor*
- **steps**: collection of *Send* and *Receive* tasks (cf. Mapping Rule 6.5 and 6.6) as well as the collection of *Parallel*, *Sequence*, and *Loop* activities (cf. Mapping Rules 6.2-6.4) that are generated on the base of the *Actor's MessageFlows*
- **controlFlow**: collection of *ControlFlows*, which are introduced for the purpose of combining the generated *Activities*

Mapping Rule 6.1 generates the plan itself, the body of the plan, in contrast, is generated by any subsequent active message flow and further relevant information with respect to (i) the kinds of message scopes used by its message flows and (ii) potential time outs are collected and transfered to the corresponding target concept of the behavior view. To generate the plan-specific structure, depending on the particular input element, the Mapping Rules 6.2 to 6.8 are invoked.

The generated concepts are collected by the *steps* variable. The generated control flows necessary to indicate the business logic are included in the *controlFlow* variable. Further variables of the plan like *informationFlow*, *localKnowlegde*, etc. are not instantiated as the AIP does not provide any adequate information on them.

The generated plan body hence consists of a number of send/receive actions expressed by the corresponding send and receive tasks generated by either invoking Mapping Rule 6.6 or Mapping Rule 6.5. Likewise, depending on exchange mode's type of an actor's message flow the activities *MessageScope*, *Parallel*, *Sequence*, and *Loop* are introduced. This means, that a *Parallel* operation is mapped to a *Parallel* activity, a *Sequence* operation is mapped to a *Sequence* activity, and finally a *Loop* operation is mapped to a *Loop* activity. This is formalized in more detail by the Mapping Rules 6.2 to 6.4. A message flow is also the source for generating *Decision* activities. However, the types of message flows coming into consideration is restricted to those that have a non-empty *messageflow* reference. The interpretation of this is that any agent instance bound to this actor could behave differently in the actual state (i.e. *MessageFlow*) and will hence be situated in different states in the course of the protocol.

Last but not least, the *Begin* and *End* tasks are automatically introduced. Apart from the parts of a plan that can be automatically generated, missing concepts like *InformationFlow* as well as *Knowledge*, which are necessary for executing the plan, have to be manually added by the system developer in the detailed design phase.

Any plan generated in this manner is encapsulated as actor's provided capabilities, which are part of the *potentialBehaviors* (see Section 4.3.1) an agent bound to the corresponding actor has access to.

Mapping Rule 6.2: ExchangeMode:Parallel → Parallel

- **name**: name of the *ExchangeMode's MessageScope* plus the extension *Parallel*
- **flows**: collection of *ControlFlows* necessary for linking the generated *Activities*
- **steps**: collection of *MessageScopes* part of the *MessageScope's messageSplit* reference. Depending on the *MessageScope's* type, one of the Mapping Rules 6.2 to 6.6 is invoked

For any kind of MessageScope contained by another MessageScope of type ExecutionMode:Parallel, a unique trace within the newly formed Parallel activity is reserved, where each trace may again consist of any other combination of complex (e.g. Sequence) or simple (e.g. Send, Receive) control structures. The structure of this trace certainly depends again on the *operations* of its children of type MessageScope. Any MessageScope of operation types ExchangeMode:Sequence is mapped in a nearly similar manner, which is expressed by Mapping Rule 6.3.

Mapping Rule 6.3: ExchangeMode:Sequence → Sequence

- **name**: name of the ExchangeMode's MessageScope plus the extension Sequence
- **flows**: collection of ControlFlows, which are instantiated when linking the generated Activities included by the *steps* variable
- **steps**: collection of MessageScopes part of the MessageScope's *messageSplit* variable. Depending on the MessageScope's type, one of the Mapping Rules 6.2 to 6.6 is invoked

In contrast to *ExchangeMode:Parallel*, in the case of an *ExchangeMode:Sequence*, a unique trace is defined. The order in which the contained activities are arranged is deduced from the order (from top to bottom) of the contained message scopes. The other parallel's variables like *synchronizationMode* are not automatically instantiated and need to be filled by hand. Like in the previous mapping rule, in the case of *ExchangeMode:Loop*, only a single trace is generated. However, this trace is executed in a loop manner. The details are given in Mapping Rule 6.4.

Mapping Rule 6.4: ExchangeMode:Loop → Loop

- **name**: name of the *ExchangeMode's MessageScope* plus the extension *Loop*
- **flows**: collection of *ControlFlows* that are introduced when linking the generated *Activities*
- **steps**: collection of *MessageScopes* part of the *MessageScope's messageSplit* reference. Depending on the *MessageScope's* type, one of the Mapping Rules 6.2 to 6.6 is invoked

The unique trace includes again a number of activities, where the order of them is deduced from the ordering of the contained message scopes expressed by the *messageSplit* reference. In contrast to the cases previously debated, a *ExchangeMode:None* refers to exactly one ACL message. For this case, a further splitting into message scopes through the variable *messageSplit* is forbidden. Hence, the ACL message referred to by the message scope is either sent or received, which means that the particular message scope is either mapped to a receive or send task. The type of the task introduced finally depends on whether the corresponding message flow sends (i.e. as part of the *forkOperator* variable) or receives (i.e. as part of the *joinOperator* variable) the particular ACL message. If the message scope is part of a message flow's *forkOperator*, Mapping Rule 6.5 will be applied.

Mapping Rule 6.5: ExchangeMode:None → Send

- **name**: name of the *ExchangeMode's MessageScope* plus the extension *Send*
- **message**: reference to the transformed *ACLMessage* (cf. Mapping Rule 6.7) that is referred by the *ExchangeMode's MessageScope*

Otherwise, if the message scope is part of any message flow's *joinOperator*, Mapping Rule 6.6 is invoked.

Mapping Rule 6.6: ExchangeMode:None → Receive

- **name**: name of the *ExchangeMode's MessageScope* plus the extension *Receive*
- **message**: reference to the transformed *ACLMessage* (cf. Mapping Rule 6.7) that is referred by the *ExchangeMode's MessageScope*

Both, the send and receive activities, generated by the Mapping Rules 6.5 and 6.6 refer to a message that is generated by applying Mapping Rule 6.7. At this, the ACL message referred to by the source message scope serves as input. Due to the fact that multiple agent instances might be bound to one actor, the send and receive tasks are both integrated into a parallel loop activity (cf. Section 4.7.10). This integration allows the iteration over the entire set of agent instances bound. This allows keeping a plan as generic as possible, as the information how many entities are finally playing the particular actors does not need to be known at design time.

The next mapping rule deals with the mapping between the two different kinds of messages. Within a protocol, ACL messages are used to illustrate how the exchange of information is proceeded. Whereas, "normal" messages that are defined as part of the environment are used within plans to represent the exchange of process-dependent information. One of the main difference between the two notions is that a message is globally accessible, i.e. any agent can use them inside its plans, whereas an ACL message is only visible inside a specific interaction. Mapping Rule 6.7 illustrates the details when mapping an ACL message to a message.

Mapping Rule 6.7: AclMessage → Message

- **name**: name of the *ACLMessage* plus the extension *Message*
- **aclMessage**: reference to the target *ACLMessage*

Mapping Rule 6.7 mainly defines the type of the message that is referred by the send and receive activities. However, as the protocol does not provide any information concerning the message's content that is exchanged, the content slot needs to be filled manually with process-dependent information. The same holds for the *sender* and *receiver* slot, as the information on the concrete agent instance sending and receiving the message is—at least in the most cases—not available during design-time, but most properly appointed at run time.

Finally, the last basic rule deals with the mapping of time outs. For this purpose, each time out is mapped in the manner that a wait activity is introduced and integrated into a parallel activity consisting of two paths. One path includes the wait activity, the other one includes the activities responsible for sending and receiving messages within the given time frame.

Mapping Rule 6.8: TimeOut → Wait

- **name**: name of the *TimeOut* plus the string extension *Wait*
- **timeout**: reference to the target *TimeOut*

The mapping between time out and wait activities is straight forward as all necessary information to instantiate the wait (e.g. time to wait) are already available within the time out. As an example, the mapping rules presented in this section are now applied to CNP.

6.3.3 Applying the Model Transformations: CNP's Behaviors

Section 6.3.1 presented the CNP as example of a complex agent-based interaction protocol. Now, the CNP is used as input for the endogenous model transformation bridging the analysis phase and detailed design phase of the DSML4MAS methodolgy process.

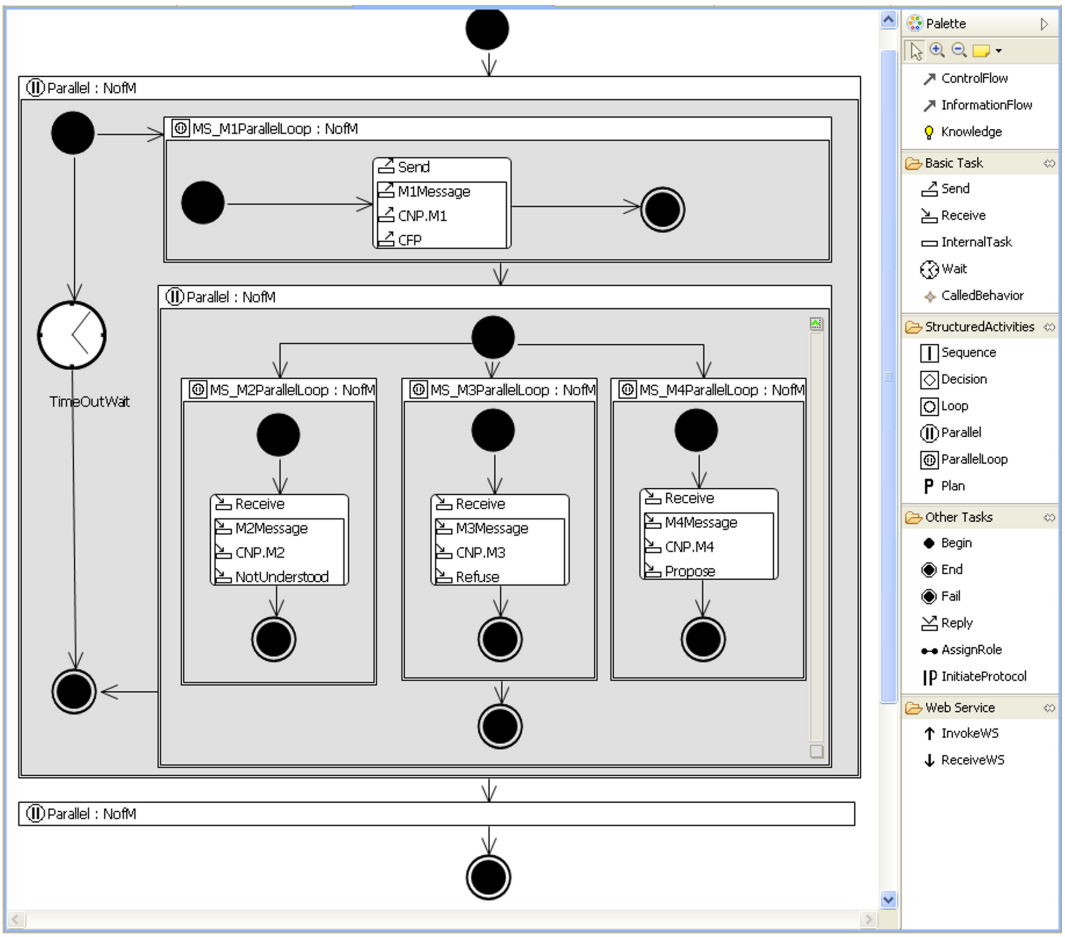


Fig. 6.14: The generated *SendAcceptReject* plan.

Fig. 6.14 depicts the generated Initiator's plan for collecting the initial responses from the Participant's side and sending of accept and reject messages. It bases on the message flows responsible for receiving the messages M2, M3 and M4 typed by the performatives NotUndestood, Refuse, and Propose, respectively. The plan mainly consists of two major parts, i.e. issuing of call for proposal messages and collecting the associated answers as well as awarding the contract to those that offer the bases conditions.

In the first phase, the CollectResponses parallel is triggered that has two traces, one responsible for waiting until the particular TimeOut is raised and one for issuing the call for proposals and collecting the responses. The execution mode of this parallel statement is XOR, meaning that the statement can be left after all messages were received or a certain time—which is defined by the wait statement TimeOutWait—has been waited. The execution mode of a parallel statement can be selected within the properties view of the graphical editor. This is also the case for messages referred to by the send and receive activities and time outs referred to by the wait task.

After issuing the call for proposals by sending the generated M1Message message to all agent instances bound to the Participant actor inside the parallel loop, the answer messages (either

M2Message, M3Message, or M4Message) are collected inside the parallel loop statement iterating over all agent instances the M1Message was sent. The execution mode of this parallel loop is AND meaning that for each agent instance bound to the Participant actor one trace is defined.

The messages are collected inside a ParallelLoop called CollectResponsesParallelLoop. For each entity bound to the Participant actor, either a plan ReceiveResponse or a plan ReceiveRefuse is executed. These plans are responsible for updating the Initiator's knowledge base which is later on utilized for selecting the most adequate bidder(s).

In the second phase, the SelectBestBidder plan is triggered which needs to be added manually as the information according to which criteria the best bidders are selected is not part of the protocol description. After an allocation has been evaluated, in the last phase, the agent instances are assigned to the corresponding actors BestBidder and RemainingBidder and informed accordingly. This is done in the SendAcceptReject parallel, where the Reject and Accept messages are sent to the RemainingBidder and BestBidder concurrently. Again, the send tasks are integrated into a parallel loop activity specifying that for each agent instance bound to one of the actors either the message Accept or Reject is sent.

6.4 Bottom Line

This chapter centered around agent-based interactions in DSML4MAS. For this purpose, we demonstrated how to use DSML4MAS to model the well-known Service Interaction Patterns proposed by Barros et al.. The main result of this first evaluation (summarized in Table 6.1) is that nearly each pattern—in contrast to other proposed standards—can nicely be described using the interaction view of PIM4AGENTS. This is an astonishing result, as special tailored interaction languages (e.g. WS-CDL, BPMN) lack this expressiveness. Moreover, we demonstrated that the Contract Net Protocol, as the most well-known AIP of the MAS community, can nicely be described using DSML4MAS. The main reason for this is that one-to-many interactions, as well as the differentiation between subactors of the same actor is naturally supported by the interaction view of PIM4AGENTS. This is a very interesting result as the most used modeling language for agents, i.e. AUML, lacks this kind of expressiveness.

Beside demonstrating how to model AIPs using DSML4MAS furthermore, we discussed how to transform AIPs to executable behaviors by applying principles of MDD. This model transformation is part of the DSML4MAS methodology and comprises the instantiation of messages and domain roles. For transforming the protocol description into the process-centric models of DSML4MAS, mappings between concepts of the interaction and behavioral views were defined. This was mainly done in an one-to-one manner. However, private information that are not part of the protocol description has to be manually integrated in the agents'internal behaviors to make each agent's behavior complete regarding execution. In a next and final step, the DSML4MAS model, which comprehends all necessary information including the generated behavioral model, is mapped to one of the supported agent-based programming language. This mapping finally allows to execute the protocol description. Chapter 7 of this dissertation is devoted to this PIM to PSM transformation.

For describing the interaction between agents in DSML4MAS, we mainly focus on AIPs. However, we also examined how to extend DSML4MAS to allow the modeling of flexible interactions. As a first step, we defined a library of AIPs that can be used to flexibly integrate any AIP into the design. Using this extension, the designer is able to specify pre- and postconditions that should hold in a particular AIP. The model transformations generate a corresponding execution model in which

the particular agent selects the most adequate AIP from the library satisfying these conditions at run-time. This extension leads to a more flexible protocol selection and thus improves the robustness of communication. We refer to (Leon-Soto et al.; 2009) for a detailed discussion on these extensions.

7. Vertical Transformation: From Design to Executable Code

As stated before, even if agent-based computing can be considered as promising approach to develop complex software systems, there are difficulties in implementing MAS due to the skills needed to move from analysis and design to code. To close the general gap between analysis and design, the MDD initiative offers suitable mechanisms that could also be utilized in the context of AOSE and DSML4MAS in particular.

In this chapter, we illustrate how to make use of the principles of MDD in the AOSE context by defining a model transformation between DSML4MAS and the agent-platform JACK. In particular, we define (i) a model-to-model transformation between the PIM4AGENTS metamodel and the metamodel of JACK called JackMM and (ii) a model-to-text transformation between JackMM and an XML-like structure to allow the import of the generated documents into the development environment of JACK.

Scope of this Chapter A strong distinction has traditionally been made between modeling languages and programming languages. One reason for this is that modeling languages have been traditionally viewed as having an informal and abstract semantics, whereas programming languages are significantly more concrete (in terms of providing an operational semantics) due to their need to be executable.

As noted by Oluyomi (2006), on the one hand, various cases of MAS development were documented ending with the analysis and design stages, on the other hand, others only focus on the implementation of MASs. But, there are not many documented examples of complete MAS development from requirements gathering to deployed systems. Consequently, the current state of the art in developing MASs is to design the agent systems basing on an AOSE methodology and take the resulting design artifact as a base to manually code the agent system using existing agent-oriented programming languages (AOPLs) or general purpose languages like Java. The resulting gap between design and code may tend to the divergence of design and implementation, which makes again the design less useful for further work in maintenance and comprehension of the system (Bordini et al.; 2007a).

In the DSML4MAS context, instead, we developed model transformations to two target AOPLs (i.e. JACK and JADE) and currently investigate a model transformation between PIM4AGENTS and Jadex. The main motivation for choosing the mentioned AOPLs is their different view on agent systems. JACK and Jadex, for instance, base on principles of the BDI architecture presented in Section 2.1.4, whereas JADE focuses on the compliance with the FIPA[1] specifications for interoperable intelligent MASs and thus concentrates on interaction aspects.

[1] http://www.fipa.org

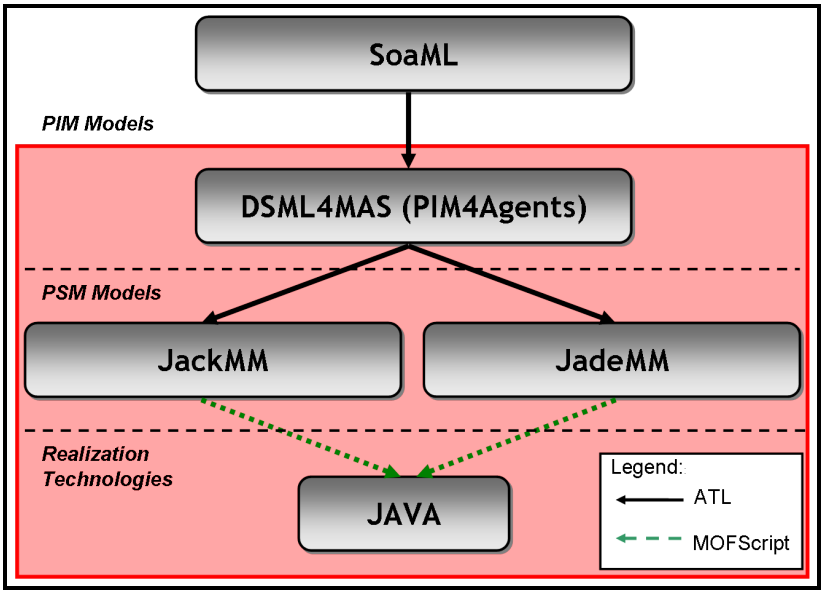


Fig. 7.1: Scope of this chapter: Model transformation between DSML4MAS and JACK and JADE.

A mapping from the PIM4AGENT'S concepts to the concepts of the different execution platform further demonstrates that the vocabulary of PIM4AGENTS can be considered as platform independent and detailed enough for code execution. In this chapter, we exemplarily focus on the DSML4MAS to JACK transformation. For detailed information regarding the DSML4MAS to JADE transformation, we refer to (Hahn et al.; 2009a) and (Gründel; 2009).

Structure of this Chapter The remainder of this chapter is organized as follows: Section 7.1 briefly presents the current state of the art on agent platforms and agent programming languages. Followed by Section 7.2 giving an overview of the agent platform JACK and introducing the metamodel of JACK. Section 7.3 then discusses the model transformation between DSML4MAS and JACK and Section 7.5 illustrates the implementation made by the model transformation applied on the conference management system introduced in Section 5.3.2.1. Finally, Section 7.5 concludes this chapter by summarizing the DSML4MAS model-driven approach to close the gap between design and implementation.

7.1 Agent Programming Languages and Platforms

In order allow the implementation of MASs and agent-based applications in general, nowadays, several MAS and agent platforms have been developed. The development of MASs is specially challenging, because it encompasses multiple and complex concerns such as distribution and concurrency. The main objective of AOPLs is to provide an infrastructure that supports the agent-based development by making these concerns transparent. Agent-oriented development frameworks provide combinations of agent-oriented programming languages and execution platforms (Bordini et al.; 2006). These languages support expressing application logic with agent-oriented concepts to provide a middleware layer between agent requirements and final realization.

AOPLs can be distinguished into three categorizations: Declarative, imperative and hybrid. In the remainder of this section, we briefly give an introduction into the area of agent-based programming by discussing the most sophisticated agent platforms and programming languages.

AgentSpeak AgentSpeak (Rao; 1996) is a BDI-based programming language, based on the Procedural Reasoning System (PRS) and the Distributed Multi-Agent Reasoning System (dMARS) (d'Inverno et al.; 2004), which determines the behavior of the agents it implements. Several extensions to AgentSpeak were provided: Jason[2] (Bordini et al.; 2007b), for instance, is an interpreter for an improved version of AgentSpeak, including speech-act based agent communication. In (Meneguzzi and Luck; 2008), AgentSpeak(PL) is proposed, which is an extended AgentSpeak(L) interpreter including a planning component to reason about declarative goals.

Java Agent DEvelopment Framework[3] Java Agent DEvelopment Framework (JADE) (Bellifemine and Rimassa; 2001) is a platform providing all necessary communication infrastructure to comply to the FIPA specification. It intentionally leaves open the internal agent architecture and necessary concepts. Instead, JADE focuses on communication performed through message passing, where each agent is equipped with an incoming message box. Standard interaction protocols specified by FIPA such as FIPA-request or FIPA-query can be used as standard templates to build an agent conversation. The protocols themselves, however, cannot be defined on a centralized view, only from the perspective of each agent involved.

Jadex[4] Jadex (Pokahr et al.; 2005a; Braubach and Pokahr; 2007; Pokahr et al.; 2005b)—as an extension of JADE—is a FIPA-compliant agent environment based on Java. It aims at modeling goal-oriented agents toward the principles of BDI. It thus combines BDI-style reasoning and FIPA-compliant communication. The Jadex engine executes the internal life cycle of an agent by keeping track of the agent's goals, while continuously selecting and executing plan steps, based on internal and external events.

Abstract Agent Programming Language[5] **(3APL)** 3APL (Dastani et al.; 2003; Hindriks et al.; 1999) is a cognitive agent programming language that includes reasoning features based on the principles of BDI. The 3APL IDE allows developers to edit 3APL programs that implement single agents, a Java-based interpreter then allows executing the 3APL programmed agents.

JACK Intelligent Agents[6] JACK (Java Agent Compiler & Kernel) is an agent-based development environment developed by the Agent-Oriented Software Group at Melbourne. Like other AOSE programming languages (e.g. Jadex or 3APL) or methodologies (Padgham and Winikoff; 2002a; Bresciani et al.; 2004; Cervenka et al.; 2004; Bauer et al.; 2001; Cheong and Winikoff; 2005a), JACK has been developed to foster the software-based development of BDI agent architectures. Thus, JACK provides programming constructs and concepts for developing complex agent-oriented applications. It bases on previous practical implementations of such systems (see Huber (1999)). In contrast to Jadex and JADE, JACK is commercially available, however, a demonstration license can be obtained at the official Web site[7].

The JACK Agent language was defined as an extension to Java to cover agent-based concepts. The JACK compiler allows debugging and compiling the JACK Agent Language to pure Java that can be executed on any Java platform. Last but not least, the JACK Agent Kernel provides the underlying run-time engine supporting the execution of the agent-based program.

[2] http://jason.sourceforge.net/JasonWebSite/Jason%20Home.php
[7] http://agent-software.com/

Beside language, compiler and kernel, JACK provides a UML-like graphical modeling environment—the so-called JACK Development Environment (JDE)—that allows modeling of MASs on a more abstract level. JDE provides graphical tools for defining agents, link them to (i) events that allow to send and handle messages and (ii) plans they are using to reach their goals. The internals of plans can be graphically defined in a workflow-like manner.

The deliberation process of a JACK agent normally consists of four steps. In the first step, the agent receives an event either internally triggered by an owned plan or from the outside (i.e. from another agent) by so-called message events. Based on the event type, the agent evaluates the set of relevant plans to handle the particular event instance. In the third step, the agent selects applicable plans from the set of relevant plans. Finally, in the fourth step, one of the applicable plan instances is selected by the agent and executed.

Beside the just mentioned platforms, further languages have been proposed, like for instance, Agent Development Kit[8] (Xu and Shatz; 2003), Living Systems Technology Suite (LS/TS[9]) (Rimassa et al.; 2006), Cougaar[10] (Gracanin et al.; 2005), Cybele[11], April Agent Platform[12], FIPA-OS [13] (Poslad et al.; 2000; Laukkanen et al.; 2002), Nuin[14] (Dickinson and Wooldridge; 2003), and ZEUS[15] (Hyacinth et al.; 1999).

To conclude, we can say that several different forms of AOPLs exist, however, as Leszczyna (2004) pointed out, only few, e.g. Agent Development Kit, JADE and JACK Intelligent Agents, are still maintained. Additional information on the implementation platforms can be found in (Bordini et al.; 2006) and (Bordini et al.; 2009). To demonstrate the usefulness of the DSML4MAS model-driven approach, we focus in the remainder of this chapter on the JACK Intelligent Agents platform.

7.2 Metamodel of Jack Intelligent Agents

In order to integrate JACK into our model-driven methodology, we specified a metamodel for JACK based on the documentation we found in (JACK Intelligent Agents; 2005) and other related approaches (e.g. (Papasimeon and Heinze; 2001)). The metamodel of JACK called JackMM—firstly presented in (Fischer et al.; 2006; Hahn et al.; 2006b)—conforms to the Ecore meta-metamodel. It is structured into three main views, i.e. the agent view, the team view, and the process view. To form the base for the model transformation between DSML4MAS and JACK, the definitions of these three viewpoints are given in the remainder of this section.

7.2.1 Agent View

The agent view specifies the structure of the autonomous entities formed to achieve a set of desired objectives. A subset of the metamodel for this view is presented in Fig. 7.2. The core concept is the

[8] http://edocs.beasys.com/manager/mgr20/pguide/overview.htm
[9] http://www.whitestein.com/
[10] http://www.cougaar.org/
[11] http://products.i-a-i.com/index.shtml
[12] http://sourceforge.net/projects/networkagent
[13] http://sourceforge.net/projects/fipa-os
[14] http://www.nuin.org/
[15] http://www.labs.bt.com/projects/agents/zeus/

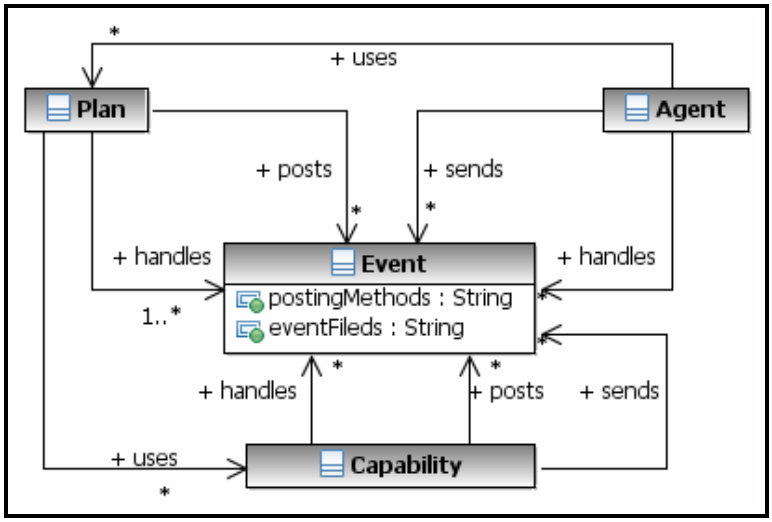


Fig. 7.2: The agent metamodel (simplified) reflecting the agent view of the JACK framework.

concept of an *Agent* that owns *Plans* to (i) handle and send *Events* and (ii) achieve goals, which are not explicitly represented as concept. A more detailed description on each single concept of the agent view is given in the following.

7.2.1.1 Agent

An *Agent* is a component that can exhibit reasoning behavior under both proactive (i.e. goal-directed) and reactive (event-driven) stimuli. When an *Agent* is instantiated, it waits until a certain goal is instantiated or it handles an *Event* that it must respond to by activating a *Plan*. Hence, when such a goal or *Event* arises, the *Agent* determines what course of action it will take. The abstract syntax of the *Agent* concept in JackMM is defined as follows:

Definition 7.2.1 (Agent in JACK)

An Agent is a 4-tuple Agent = (name, sends, handles, uses), where ***name*** *defines the name of the agent,* ***sends*** *specifies the events that the agent sends externally to other agents,* ***handles*** *defines the events that the agent will respond to if they arise by executing a plan, and* ***uses*** *defines the plans that the agent potentially executes in reaction to an event.*

The actions or activities an *Agent* performs are defined by so-called *Plans* that are specified to achieve certain goals. Any *Plan* must at least handle a single *Event* and may post/send several *Events*, which are either of the form message or goal event.

7.2.1.2 Plan

A *Plan* models procedural descriptions of what an *Agent* does to handle a given *Event*. Similar to plans in PIM4AGENTS, a *Plan* in JackMM can be considered as a sequence of activities an *Agent* executes in order to handle an *Event* or achieve a certain goal. Thus, a *Plan* describes a set of steps that are to be followed in order to accomplish a task. If a plan is applicable for certain situations is

defined through a so-called context condition. When an *Agent* receives an *Event*, the *Agent* may consider a set of relevant and applicable *Plans*. The relevance check is performed by considering the context of the *Event*, whereas the applicability check is achieved by considering the agent's current state. Thus, the beliefset constellation and the *Event* context can be considered to further filtering the set of feasible *Plans*. In the BDI architecture, this corresponds to the selection of different alternatives to achieve a goal. Apart from simple *Plans*, moreover, meta-plans can be defined that allow selecting the most useful applicable *Plan*. Informally, a *Plan* in JACK is defined as follows:

Definition 7.2.2 (Plan in JACK)

*A Plan in JACK is a 4-tuple Plan = (name, reasoningmethod, handles, posts), where **name** defines the name of the plan, **reasoningmethod** defines methods that an Agent may execute when it runs this Plan, **handles** specifies the Events triggering the execution of the Plan, and **posts** defines the Events that are posted by this Plan.*

Reasoning methods part of a Plan used for implementation issues are, in general, different from normal Java methods in that they execute as finite state machines, and may succeed or fail, depending on whether the *Agent* can complete each statement contained. The top-level reasoning method is called body, which contains a *Process* that concretely specifies how the *Agent* wants to achieve a certain goal. Details on the *Process* and its constructs are given in Section 7.2.3.1.

7.2.1.3 Event

An *Event* presets the type of stimuli *Agents*, *Teams*, *Roles*, or *TeamPlans* react to or send by taking the information carried by the event instance into account. *Events* are, in general, an import construct in agent-oriented systems as those are responsible to trigger any kind of activity within an agent. JACK distinguishes between (i) *internal stimuli* that are *Events* the *Agent* or *Team* posts to itself within plans, (ii) *external stimuli* that are messages from other *Agents*, and (iii) *motivations* such as goals the *Agent* may have. Hence, the concept of an *Event* can be used for both modeling intentions and modeling communication between *Agents*. Definition 7.2.3 states the abstract syntax of *Event*.

Definition 7.2.3 (Event in JACK)

*An Event in JACK is a triple Event = (name, eventFields, postingMehtods), where **name** defines the name of the Event, **eventFields** specifies the set of attributes that are defined within the Event and **postingMethods** all kinds of methods that can be invoked using the Event.*

The *eventFields* allow specifying the information that is sent either internally or externally as part of an *Event*. The *postingMethods* enables the designer to use different instances of an *Event*, i.e. by defining different posting methods, the designer can specify, which kind of information is exchanged in a certain situation by sending the same *Event* type.

7.2.1.4 Capability

A *Capability* in JACK can be understood to have a capability to reach a desired goals. In principle, a *Capability* allows grouping behaviors (i.e. *Plans*) that are necessary for providing certain functionalities. The concept *Capability* furthermore allows for code reuse, encapsulation of

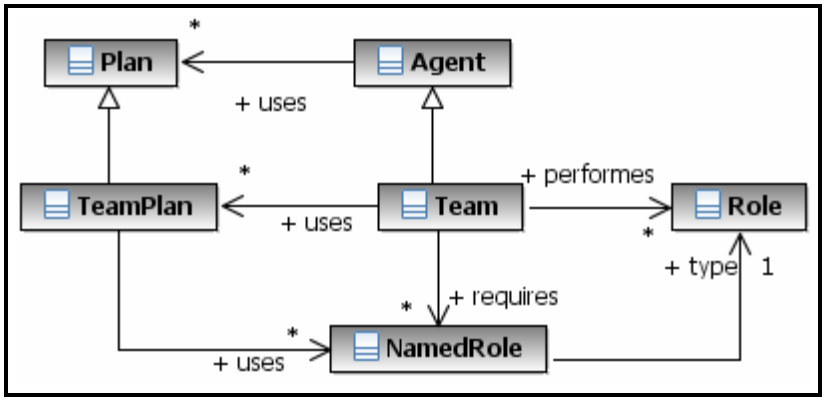


Fig. 7.3: The team metamodel (simplified) reflecting the team view in the JACK framework.

functionality, and simplification of the design process. By code reuse, we mean that different *Agents* may reuse the same capability or functionality, sharing *Capabilities* from a common library. Encapsulation means that all components that make up a *Capability* are grouped together inside the capability construct. The building of capability hierarchies is also supported by JACK.

Definition 7.2.4 (Capability in JACK)

A Capability in JACK is a 6-tuple Capability = (name, plans, handles, posts, sends, subCapabilities), where ***name*** *defines the name of the Capability,* ***plans*** *refers to the set of Plans grouped by the Capability,* ***handles*** *specifies the Event that triggers the Plan, whereas* ***posts*** *and* ***sends*** *indicate those Events that are either internally posted or externally sent. Finally, the attribute* ***subCapabiliies*** *refers to the Capabilities recursively grouped by this Capability to allow the building of hierarchies.*

A Capability can be considered as little agent as it similarly owns *Plans* that handle and send *Events* and allow achieving goals. They are normally used to structure certain *Plans* an *Agent* can make use of for certain situations. Further concepts of the agent view that are not used in the model transformation are, for instance, the concept of a *Beliefset* that represents any kind of logical knowledge about the world or the agent's own state.

7.2.2 Team View

Grouping agents into social structures like organizations and communities is an important concept to structure MASs (cf. Section 2.1.5). In order to allow the grouping of agents in JACK, the agent view has been extended. In the team view, JACK facilitates the collaboration among teams to achieve a common goal. The main concept is, therefore, the concept of *Team* consisting of other autonomous entities (i.e. *Teams*) that are coordinated with so-called *TeamPlans* (extending simple *Plans*). Similarly to *Plans*, a *TeamPlan* owns conditions and a body that again consists of a *Process* (cf. Section 7.2.3.1).

7.2.2.1 Team

As depicted in Fig. 7.2.3.1, a *Team* is a specialization of the *Agent* concept that is a distinct reasoning entity. It is characterized by the *Roles* it performs (i.e. to play *Roles* in other *Teams* as well) and/or

the *Roles* it requires other *Teams* to perform. The formation of a given *Team* is achieved by attaching sub-teams capable of performing the *Roles* required by the *Team* to achieve a common goal. By filling the required *Roles*, a *Team* can delegate tasks to its team members through *TeamPlans* using concepts like *TeamAchieveNode*. The abstract syntax of a *Team* is defined as follows:

Definition 7.2.5 (Team in JACK)

*A Team is a 6-tuple Team = (name, uses, sends, handles, performs, requires), where **name** defines the name of the Team, **uses** defines the TeamPlans that the Team can execute in reaction to an Event, **sends** specifies the Events the Team sends externally to other Teams, **handles** defines the Events that the Team will attempt to respond to if they arise by executing a TeamPlan, **performs** specifies the Role the Team performs itself to the outside, and **requires** defines the NamedRoles the Team requires in order to solve tasks requested by the outside.*

For establishing complex *Teams*, they can be nested (i.e. a *Team* can consists of other *Teams* etc.). The sort of nesting is, however, not done explicitly by defining *Teams* containing other *Teams*. Rather, a *Team* requires a set of *NamedRoles* that are performed through other *Teams*. Hence, any *Team* owning the certain *NamedRole* is not explicitly bound to a unique *Team*, but may perform one and the same role in different team contexts. This gives the *Team* more flexibility, as the *Team* performing the certain *NamedRole* can be flexible bound during run-time. For coordinating team members, special activities are offered by a *TeamPlan* extending ordinary *Plans* (cf. Fig. 7.3).

7.2.2.2 TeamPlan

A *TeamPlan* specifies the behavior of a *Team* in reaction to a specific *Event*. As a specialization of *Plan*, a *TeamPlan* also defines a set of activities specifying how a particular task is achieved by particular *Roles*. Thus a *TeamPlan* defines how a task is achieved in terms of *Roles*. In order to coordinate the *Team's* behavior, a *TeamPlan* provides additional constructs like the *SendNode* and *TeamAchieveNode*. Section 7.2.3 gives a detailed overview on the main constructs within a *TeamPlan's* body.

Definition 7.2.6 (TeamPlan in JACK)

*A TeamPlan in JACK is a 5-tuple TeamPlan = (name, reasoningmethod, handles, posts, uses), where **name** defines the name of the plan, **reasoningmethod** defines the methods a Team may execute when it runs this TeamPlan, **handles** specifies the Event triggering the execution of the TeamPlan, **posts** defines the Events posted by the TeamPlan, and **uses** defines the NamedRoles that are needed by the TeamPlan for the purpose of assigning tasks to the Teams filling this role.*

The main difference to a *Plan* is that a *TeamPlan* additionally requires a set of *NamedRoles* that represent interfaces to their members. Using these interfaces, the *Team* can assign adequate subgoals to its members through a *TeamPlan*, where adequate means that the sub-teams should have *TeamPlans* available that allow them to achieve the requested goals. Hence, a *TeamPlan* is used like a normal *Plan* to achieve goals, but it can, furthermore, be used to coordinate the *Team's* members to achieve complex goals that go beyond the functionalities and capabilities of a single *Agent*.

7.2.2.3 Role

A *Role* in JACK defines a relationship between a role tenderer (which could be a *Team*) and a role filler (sub-team), specifying the goals that both participants must achieve. *Roles* are a very

important concept to define *Teams* in JACK as those specify which messages—which are rather *Events*—the role fillers are able to react to and which messages they are likely to send. The abstract syntax of a *Role* in JACK is defined as follows:

Definition 7.2.7 (Role in JACK)

*A Role in JACK is a 4-tuple Role = (name, handles, posts, sends), where **name** defines the name of the Role, **handles** specifies the Events the Role should be able to handle, **posts** illustrates the Events the Role should be able to internally post and finally, **sends** depicts the Events the Role should be able to send to other Roles.*

Even if the concepts of a *Role* in JackMM and a *DomainRole* in PIM4AGENTS seem to be very similar, the main difference is that a *Role* in JackMM does not refer to *TeamPlans*, neither it provides nor requires any certain behavioral descriptions. In fact, as previously mentioned, a *Role* in JACK should be more considered as interface, providing certain communication channels through which any *Team* can interact with.

7.2.2.4 NamedRole

NamedRoles are used within JACK whenever a *Team* requires a certain functionality. This could either happen within the *Team* itself or as part of a *TeamPlan* when a task/goal is assigned. Definition 7.2.8 gives a more precise definition.

Definition 7.2.8 (NamedRole in JACK)

*A NamedRole in JACK is a tuple Role = (name, type), where **name** defines the name of the NamedRole and **type** names the Role that the NamedRole represents.*

Beside the *NamedRole* itself, within a *TeamPlan*, the system designer can define how many *NamedRoles* are necessary at least and most to achieve the certain functionality. Important to mention is that a *NamedRole* can be considered as placeholder for a *Role* in a *Team* context. For which role the placeholder is finally used is expressed by the *NamesRole's type*.

7.2.3 Process View

For BDI architectures, plans represent the behavioral elements of an agent and are composed of a head and a body part (cf. Section 2.1.4). The process view of JackMM focuses on the *Plan* (or *TeamPlan*) body in JackMM and its constructs to allow providing a predefined course of action, given in a procedural language. This course of actions or process is to be executed by the *Agent* (or *Team*), when the *Plan* (or *TeamPlan*) is selected for execution, and may contain actions like sending messages, manipulating beliefs, or creating subgoals. The *Process* is included by a *ReasoningMethod* that includes the sequence of activities executed by the *Agent* (or *Team*) and a set of *LocalVariables* that can be accessed and changed within a *Process*. The metamodel of the process view is depicted in Fig. 7.4.

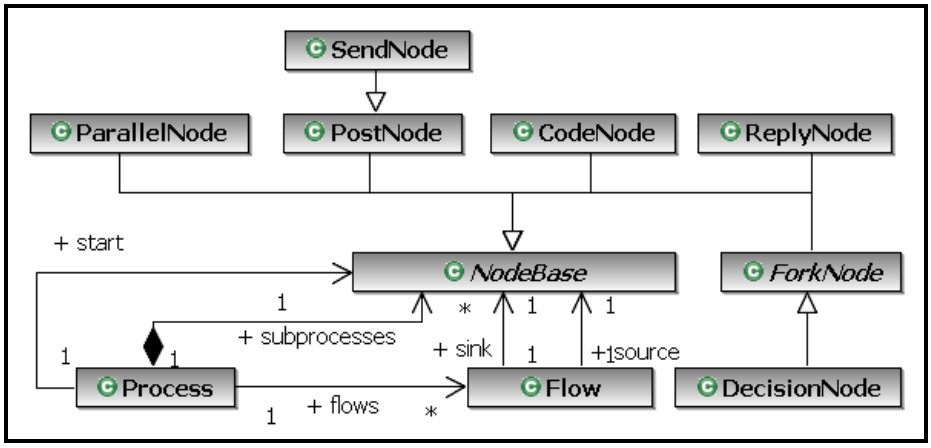


Fig. 7.4: The partial process metamodel reflecting the process view in the JACK framework.

7.2.3.1 Process

A Process in JackMM defines the set of activities—the so-called NodeBases—that are necessary for achieving a particular goal and a set of Flows that connect the determined NodeBases. The so-called LocalVariables allow defining how information flow between the NodeBases. The abstract syntax of NodeBase is as follows:

Definition 7.2.9 (Process in JACK)

A Process *in JACK is a triple Process = (start, subprocesses, flows), where* ***start*** *refers to the first NodeBase,* ***subprocesses*** *contains all NodeBases except the starting node that are necessary for achieving the overall goal, and* ***flows*** *defines the set of Flows that connect the contained NodeBases.*

7.2.3.2 NodeBase

A *NodeBase* is an abstract class that provides the common attributes for further specializations. This concept has several specializations that are discussed in the remainder of this section.

Definition 7.2.10 (NodeBase in JACK)

A NodeBase in JACK is a tuple NodeBase = (name), where ***name*** *defines the name of the NodeBase.*

As mentioned before, a *NodeBase* is an abstract class that is the specialization of several activities within a *Process*. In the following, we briefly introduce the main concepts necessary to adequate implement the model transformation between design and implementation.

- **ForkNode** is an abstract class that extends *NodeBase* for the support of alternative outputs, where *alternativeFlow* indicates an alternative following node in execution, with *defaultFlow* being the default one.
- **ParallelNode** represents the parallel statement node, where *parallelTasks* refers to a collection of tasks or processes that must be executed in parallel.
- **ParallelTask** represents a parallel process inside the *ParallelNode*, where *label* refers to an identifier for execution and exception handling within the *ParallelNode*.
- **PostNode** posts an *Event* to a *Role*, where the attribute *event* refers to the *Event* to be posted.

- **SendNode** sends an *Event* to a *Role*, where the attribute *targetAgent* refers to the name of the recipient *Agent* for the sent *Event*.
- **ReplyNode** replies to an *Event* received by the *Agent*, where *originalMessage* refers to the *Event* to which the reply responds.
- **CodeNode** executes Java code within the *Plan*, where the attribute *code* defines the Java code to be executed.
- **DecisionNode** represents an if-else decision, where the *condition* represents the boolean expression to be evaluated in the decision.
- **TeamAchieveNode** delegates a task or goal to a subteam of the given *Team*, where the variable *roleInstanceLocalReference* refers to the role container that should handle the goal and *eventInstance* defines the *Event* describing the goal that the subteam must try to achieve.
- **WaitForNode** causes the *Agent* to wait for a given *condition* to be true. This condition could either be a logical variable or integer defining a certain time value.

The presented concepts are only a selection of the most important ones for our purposes. The complete list of process activities is given in the JACK documentation (JACK Intelligent Agents; 2005). Similar to the behavior viewpoint in PIM4AGENTS, in order to connect these *NodeBases* and hence to define the process logic, the concept of *Flow* is utilized that is defined in the following.

7.2.3.3 Flow

A *Process* in JacKMM is as mentioned earlier a workflow-like sequence of activities (i.e. *NodeBases*). Similar to the *ControlFlow* in PIM4AGENTS, which connects two *Activities*, in order to define the execution order within a *Process* in JackMM, the concept of a *Flow* is introduced connecting two *NodeBases* that are executed in a sequential manner. Definition 7.2.11 gives the formal specification.

Definition 7.2.11 (Flow in JACK)

*A Flow in JACK is a tuple Flow = (name, source, sink), where **name** refers to the name of the Flow, **source** the source NodeBase and **sink** the sink NodeBase.*

Hence, like the concept of a *ControlFlow* in PIM4AGENTS, a *Flow* in JackMM is a directed link between two *NodeBases*, i.e., the sink and source. The abstract syntax of the JACK metamodel is, in the following section, used to depict the core mappings of model-driven methodology process between DSML4MAS on the PIM level and JACK on the PSM level.

7.3 From DSML4MAS to JACK

In this section, we illustrate our model-driven approach to connect the design and the implementation in DSML4MAS. Therefore, we apply the principles of MDD by defining a model transformation between DSML4MAS and JACK. In accordance to the DSML4MAS model transformation architecture (Section 3.2.2.3), this model transformation consists of (i) a model-to-model transformation between PIM4AGENTS and JackMM (cf. Section 7.3.1) and (ii) a model-to-text transformation between JackMM and GCode (cf. Section 7.3.2), which generates an XML-like document that can be imported into the JDE.

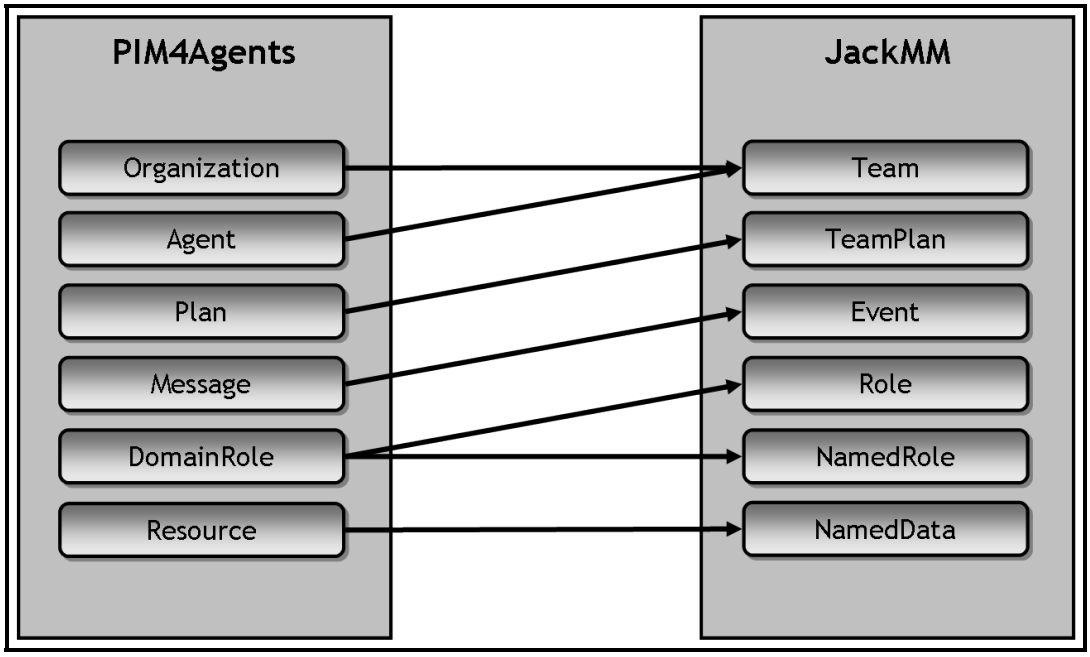


Fig. 7.5: An overview on the model mappings from PIM4AGENTS to JackMM.

7.3.1 Model to Model Transformation: From PIM4AGENTS to JackMM

In this section, we bring together the metamodels of PIM4AGENTS and JackMM. Therefore, several basic mapping rules were defined (cf. Fig. 7.5) that are listed in the remainder of this section. Again, these mapping rules consist of (i) a head that defines which concepts from the source metamodel PIM4AGENTS are mapped to which concepts of the target metamodel JackMM and (ii) a body that defines how the attribute's information of the target metamodel is derived.

The first mapping rule of the model-to-model transformation covers the mapping from the organization view of PIM4AGENTS to the team view of JackMM. The concrete mappings are expressed in Mapping Rule 7.1.

Mapping Rule 7.1: PIM4Agents:Organization → JackMM:Team

- **performs**: collection of *DomainRoles* performed by the *Organization* (cf. Mapping Rule 7.6)
- **requires**: collection of *DomainRoles* performed by the *Organization's* members (cf. Mapping Rule 7.5)
- **handles**: collection of *Messages* referred to by a *Receive* in any *Plan* the *Organization* has access to (Mapping Rule 7.4)
- **sends**: collection of *Messages* referred to by a *Send* in any *Plan* the *Organization* has access to (cf. Mapping Rule 7.4)
- **uses**: collection of *Plans* the *Organization* has access to (cf. Mapping Rule 7.3)

The source and target concepts of Mapping Rule 7.1 nicely corresponds to each other as both, the *Organization* and *Team*, (i) make use of a process that specifies how their members are coordinated or a certain functionality is provided and (ii) require and perform roles[16]. The main difference between both concepts is the manner in which interactions are defined. The interaction in PIM4AGENTS is mainly defined through AIPs, whereas JackMM favors an event-driven approach without explicitly specifying AIPs, but only the *Events* and the *Plans* handling and sending them. However, as the endogenous transformation depicted in Chapter 6.3 transforms any *Protocol* description into corresponding *Plans* of PIM4AGENTS, we do not need to handle *Protocols* in the vertical transformation at all and can mainly focus on the transformation of either generated or manually defined *Plans* in PIM4AGENTS. Hence, any *Plan* used by an *Organization* is mapped to a set of *TeamPlans* the created *Team* makes use of.

The set of *Plans* an *Organization* as a specialization of an *Agents* can adopt is the union of (i) the set of *Plans* the *Organization* refers to through its *behavior* attribute, (ii) the set of *Plans* contained by the *Capabilities* the *Organization* might apply and (iii) the kinds of *Plans* contained by any *Capability* the *Organization's DomainRole* provides. The complete list of *Plans* an *Organization* may use is defined through the secondary variable *potentialBehaviors* declared in the *Agent* Schema (cf. Schema 4.3.1). As any plan in JACK handles an *Event*, a *Plan* in PIM4AGENTS is split into sequences that have all in common that they start with a *Send* task. *Plans* that do not react on an incoming *Message* are directly transformed. How this is done is detailed by Mapping Rule 7.3. The *Events* a *Team* sends or handles are, moreover, extracted from the *Messages* referred to by a *Send* or *Receive* statement. Finally, the *Team* performs and requires *Roles* that are extracted from the *Organization's* provided and required *DomainRoles*.

The second transformation rule deals with the mapping from the agent view of PIM4AGENTS to the team view in JackMM. The corresponding mapping is expressed by Mapping Rule 7.2.

Mapping Rule 7.2: PIM4Agents:Agent → JackMM:Team

- **performs**: collection of *DomainRoles* performed by the particular *Agent* (cf. Mapping Rule 7.5)
- **handles**: collection of *Messages* received in any *Plan* the *Agent* has access to through the *Receive* activity (cf. Mapping Rule 7.4)
- **sends**: collection of *Messages* received in any *Plan* the *Agent* has access through the *Send* activity (cf. Mapping Rule 7.4)
- **uses**: collection of *Plans* either (i) contained in the *Agent's behaviors*, (ii) contained in the *Agent's capabilities* or (iii) provided by any *Role* the *Agent* is performing through the *providesCapability* variable (cf. Mapping Rule 7.3)

At a first glance, the concept *Agent* of JackMM seems to be the best match for an *Agent* in PIM4AGENTS, but as the latter performs *DomainRoles*, it is recommended to assign an *Agent* in PIM4AGENTS to a *Team* in JackMM as an *Agent* in JackMM does not refer to any *Roles* (see Fig. 7.2). The main difference between Mapping Rule 7.2 and Mapping Rule 7.1 is the fact that, when mapping an *Agent* to a *Team*, we instantiate an atomic *Team* meaning that the *Team* does not require any *NamedRole* (i.e. sub-team) to which tasks are assigned in *TeamPlans*. When mapping an *Organization*, the initiated *Team* requires a set of *NamedRoles* that are performed by its members (which are again of type Team).

[16] in case of PIM4AGENTS these are DomainRoles

The mapping of the remaining variables is done as follows: The *Plans* used by the *Agent* in PIM4AGENTS are mapped to a set of *TeamPlans* the corresponding *Team* makes us of. The *Messages* sent and received by the *Agent* are mapped to *Events* that are either sent or handled by the *Team*. Lastly, the *Team* performs the *Roles* that are generated based on the input of the *Agent's DomainRoles*.

After demonstrating how to map the autonomous entities (i.e. *Agent* and *Organization*), we now illustrate how to transform *Plans* in PIM4AGENTS facilitating autonomous entities to behave in an autonomous manner. Hence, the third mapping rule covers the mapping between the behavioral view of PIM4AGENTS and the process view of JackMM.

Mapping Rule 7.3: PIM4Agents:Plan → JackMM:TeamPlan

- **uses**: collection of *DomainRoles* required by the *Organization* using this *Plan*. If an *Agent* uses this *Plan*, no *NamedRoles* are instantiated (cf. Mapping Rule 7.5)
- **sent**: collection of *Messages* sent within the *Plan*, i.e. *Messages* referred to by the *Plan's Send* activities (cf. Mapping Rule 7.4)
- **handles**: collection of *Messages* received within the *Plan*, i.e. *Messages* referred to by the Plan's Receive activities (cf. Mapping Rule 7.4)
- **body**: direct mapping between the *Activities* of PIM4AGENTS and *NodeBases* of JackMM
- **relevanceCondition**: the *preCondition* of the Plan in PIM4AGENTS

As previously presented, a *TeamPlan* requires a set of *NamedRoles*, which are extracted from the *DomainRole* an *Organization* in PIM4AGENTS requires. The kind of *Events* either handled or sent can easily be deduced from the *Messages* received or sent, respectively.

As specified in Definition 4.7.1, a *Plan* in PIM4AGENTS consists of several *Activities*, which are connected through a *ControlFlow* and possibly *InformationFlow*. A *Plan* unions *StructuredActivities* that define more complex control structures and atomic *Tasks*. Any kind of *Activity* refers to exactly one incoming and outgoing *ControlFlow*. How to map the particular concepts of PIM4AGENTS is illustrated in Table 7.1.

The body of the corresponding *TeamPlan* is generated in an one-to-one manner at least for those *Activities* in PIM4AGENTS that were directly supported by JackMM. An example is the *Send* activity, which is transformed to a *SendNode* in JackMM. The *Message* a *Send* refers to is directly transformed to the corresponding *Event*.

As a *TeamPlan* automatically handles an *Event*, we do not need to directly transfer *Receive* activities to related concepts in JackMM. However, as *Plans* in PIM4AGENTS base on *MessageFlows*, which clearly describe which *ACLMessages* are received and sent, we can easily generate the *TeamPlan*-specific structure.

Concepts like *Parallel*, *Decision*, and *Wait* can also be mapped in an one-to-one fashion as illustrated in Table 7.1. In the contrast to concepts like *Loop*, *ParallelLoop*, and *Sequence*, which are not directly supported by JackMM. In the case of *Sequences* this can easily be compensated by connecting the predecessor of a *Sequence* with the first *Activity* of the *Sequence* and the last *Activity* of a *Sequence* with its successor. For the concepts of *Loop* and *ParallelLoop*, we define

Process mappings		
Source	*Target*	*Explanations*
Process	*Plan*	the subprocesses and flows in JackMM are represented by the *Plan's Activities* and *ControlFlows*.
Flow	*ControlFlow*	by connecting the *NodeBases* using *Flows* Sequences of PIM4AGENTS can be represented
ParallelNode	*Parallel*	depending on the execution type (XOR, AND), we set the condition of the *ParalledNode* to ANY or ALL
SendNode	*Send*	the *Event* that is sent in the *SendNode* is used to instantiate the corresponding *Message* in PIM4AGENTS
CodeNode	*InternalTask*	statements inside an *InternalTask* are transformed to *CodeNode*
DecisionNode	*Decision*	the *condition* in PIM4AGENTS is mapped to the condition in JackMM

Tab. 7.1: Mapping between the PIM4AGENTS and JackMM process parts.

templates consisting of several activities (e.g. *Decision* concept used for looping purpose) and filled with the *Activities* contained by the source concepts (i.e. *Parallel* and *ParallelLoop*).

The fourth mapping rule defines how to map the interaction aspect of the PIM4AGENTS that describes how to specify the interaction in a protocol-driven manner to an event-driven manner as it is supported by JACK. Particularly, Mapping Rule 7.4 depicts how *Events* are instantiated in order to be handled and sent within *Plans*.

Mapping Rule 7.4: PIM4Agents:Message → JackMM:Event

- **name**: name of the *Message*
- **eventFields**: the types of variables used to exchange information. For each *content* of a *Message* one variable is instantiated
- **postingMethods**: a default operation that allows assigning the necessary content

As mentioned in Section 7.2, JACK distinguishes between several different types of *Events*. Though, Mapping Rule 7.4 is restricted to *MessageEvents*, i.e. goal events are not covered as the core of PIM4AGENTS does not yet include any goal-oriented concepts. Thus, each *Message* defined as part of the *Environment* and used within the *Send* or *Receive* activity is mapped to an *Event* in JackMM. This is done independent of its type, i.e. whether the *Message* is sent/received in an asynchronous or synchronous manner.

The next mapping rule deals with the generation of *Roles* in JackMM. In PIM4AGENTS, two different role types are distinguished. The *DomainRole* focuses on the role an *Agent* or *Organization* is able to play within a certain domain. The *Actor* can be considered as a generic role used to express between which parties the exchange of messages is proceeded. However, both kinds of *Roles* are linked through the *ActorBinding* concept. This concepts defines, which *DomainRole* is

representing which *Actor* in the context of a certain *Collaboration*. As JACK does not provide any mechanisms to define protocols and due to the fact that the *interaction to behavior* transformation already extracts the necessary information from the *Actor* and assigns it to the *DomainRole*, for deriving the necessary *Roles* in JACK, the *DomainRole* concept is sufficient.

Mapping Rule 7.5: PIM4Agents:DomainRole → JackMM:Role

- **name**: name of the *DomainRole*
- **handles**: collection of *Messages* that are received within any *Plan* contained in the *Capabilities* either required or provided by the *Role* (cf. Mapping Rule 7.4)
- **sends**: collection of *Messages* that are sent within any *Plan* contained in the *Capabilities* either required or provided by the *Role* (cf. Mapping Rule 7.4)

As Mapping Rule 7.5 nicely demonstrates, the attributes a *Role* in JackMM requires can easily derived from the properties a *DomainRole* offers. The *Plans* that are part of a *DomainRole's* required and provided *Capabilities* are given to the *Teams* that finally behave in accordance to this particular *Role*.

The next mapping rule deals with the generation of *NamedRoles* in JackMM. As previously discussed, *NamedRoles* are required by *Teams* and used inside *TeamPlans* to assign tasks and responsibilities across team members. At this, the *NamedRoles* define role container objects that include agent instances as role fillers at run-time.

Mapping Rule 7.6: PIM4Agents.DomainRole → JackMM.NamedRole

- **name**: name of the *DomainRole* plus the string `NamedRole`
- **type**: refer to the *Role* that was generated by applying Mapping Rule 7.5

Beside the *Roles* that are generated on the base of *DomainRoles*, we additionally instantiate *NamedRoles* with Mapping Rule 7.6. The only difference is that only *DomainRoles* are applied to this rule that are required by any *Organization* in PIM4AGENTS since only *Teams* that are not atomic can refer to *NamedRoles*.

The last fundamental mapping rule of the model-to-model transformation transforms the information model described in PIM4AGENTS into the corresponding information model of Jack. For this purpose, we instantiate the so-called *NamedData*.

Mapping Rule 7.7: PIM4Agents:Resource → JackMM:NamedData

Resources an *Agent* has access to in PIM4AGENTS are mapped to *NamedData* an *Agent* or *Team* uses. The *NamedData* concepts refers to so-called external classes that are specified in e.g. Java.

```
texttransformation jackMM2gcode_Event (in e:"http://dfki/jackMM.ecore"){
  e.Event::event2gcode() {
  file ("jack/" + self.name + ".gevent")
  <%<BAPI_Event
      :superclass  "%> self._getFeature("extends") <%"
      :doc
        <BAPI_Text
          :name  "%> self.name <%"
          :filename  "%>self.name <%.gevent"
          :type  "aos.jack.ed.Event"
          :java
  %>  if (self.eventFields.evfields.size() > 0) {<%
          :evfields
          (
  %>  self.eventFields.evfields->forEach(field:e.Evfield) {<%
            <BAPI_EventField
              :name  "%> field.name <%"
              :type  "%> field.type <%"
            >
  %> } <% )        ....
```

Listing 7.1: Partial MOFScript template to generate events

7.3.2 Model to Text Transformation: From JackMM to GCode

Section 7.3 presented the model-to-model transformation between PIM4AGENTS and JackMM, which takes the models conforming to PIM4AGENTS as input and introduces a corresponding representation conforming to the JACK metamodel. In this section, we now concentrate on how the information inside the JackMM models is extracted to generate executable JACK code. Thus, we focus in this section on the model-to-text transformation that involves serializing the JackMM models into GCode, which is used by JACK for internal representation.

In MOFScript, several serialization rules (i.e., templates) are created following the structure of the source MOF-based metamodel of JackMM. This means that the information regarding the concept itself as well as the references to other concepts are extracted and assigned to the template's attributes. For the serialization of JackMM models, we create a template for the concepts *Event*, *Role*, *NamedRole*, *Agent*, *Plan*, *Team*, and *TeamPlan*. For each instance of the mentioned concepts in the JackMM model, a new file is generated. Apart from serializing the main concepts, additionally, a template is created that generates a project file that contains a reference to all newly created JACK files. By importing the project file into JDE, all other JACK files are automatically imported. The imported artifacts can, in an final step, be refined if necessary and compiled to generate Java code representing the implementation.

To provide the reader with a feeling of how the MOFScript templates are defined, Listing 7.1 presents a segment of the template that serializes an *Event* from JackMM. For additional details on the MOFScript syntax, please refer to the official documentation that can be found at the MOFscript web page at http://www.eclipse.org/gmt/mofscript/.

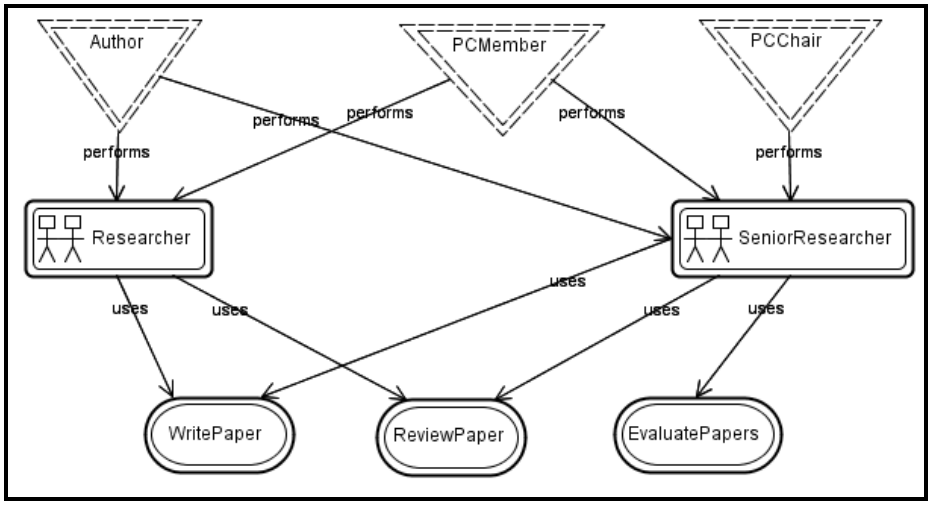


Fig. 7.6: The generated agent view of the CMS example.

7.3.3 Illustrative Example: Conference Management in JACK

In the remainder of this section, the model transformations presented in the previous section are illustratively discussed. The generated JackMM models bases on the PIM4AGENTS'S-based CMS example from Section 5.3.2.1.

7.3.3.1 Agent View of CMS in JACK

The generated agent view of the JackMM-based CMS example is illustrated in Fig. 7.6. Mapping Rule 7.2 instantiated two teams, i.e., the *Researcher* and *SeniorResearcher* team. Similar to the PIM4AGENTS'S-based CMS example, the *Researcher* performs the *Author* and *PCMember* roles, the *SeniorResearcher* performs the *Author*, *PCMember*, and *PCChair* roles. All roles are instantiated by Mapping Rule 7.5. Furthermore, a set of team plans, i.e., *EvaluatePapers*, *WritePaper*, and *ReviewPaper*, are created by Mapping Rule 7.3 that are used by the *SeniorResearcher* and *Researcher* teams.

7.3.3.2 Team View of CMS in JACK

The generated JACK team view is depicted in Fig. 7.7. Apart from the agent concept in PIM4AGENTS, Mapping Rule 7.1 creates non-atomic teams for any organization in PIM4AGENTS. Hence, the teams *ConferenceOrganization* and *ReviewOrganization* are generated. The *ConferenceOrganzation* requires the *AuthorNamedRole*, *PCChairNamedRole*, and the *PCMemberNamedRole*. The latter is also required by the *ReviewOrganization*, in addition to the *ReviewerNamedRole*. All named roles result from applying Mapping Rule 7.6. Lastly, the *ReviewerOrganization* performs the *PCMember* role that is again produced by Mapping Rule 7.5.

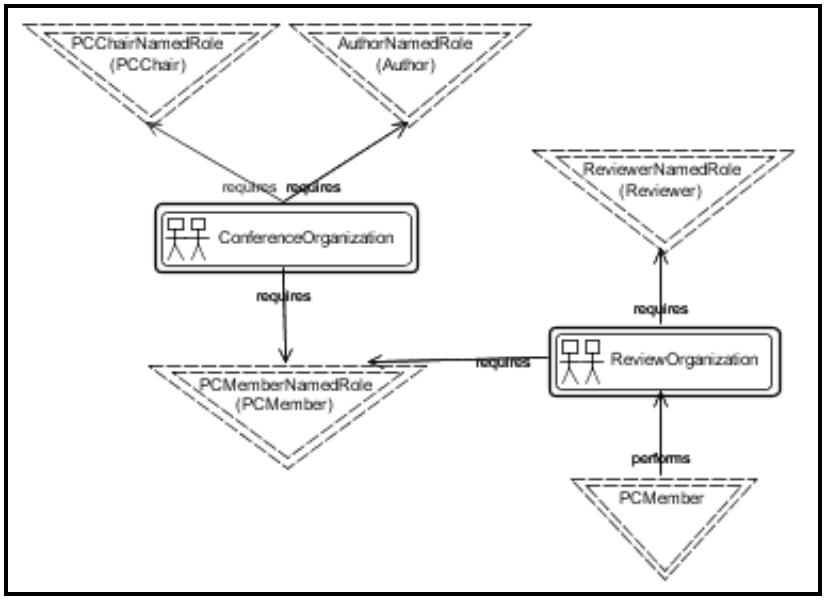


Fig. 7.7: The generated team view of the CMS example.

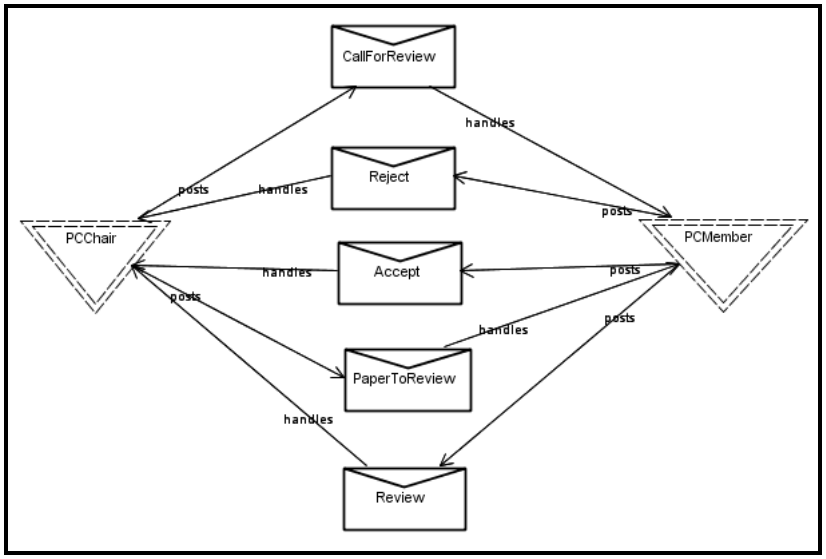


Fig. 7.8: The JACK representation of the interaction view of CMS.

7.3.3.3 Interaction View of CMS in JACK

JACK does not naturally allow to model AIPs as the order in which events are sent and handled can only be specified within plans. The abstract modeling is, in contrast, supported through designing the involved roles in the interaction and events exchanged by them. The abstract view on the PIM4AGENTS-based *CallForReviews* protocol is depicted in Fig. 7.8. This view includes the roles

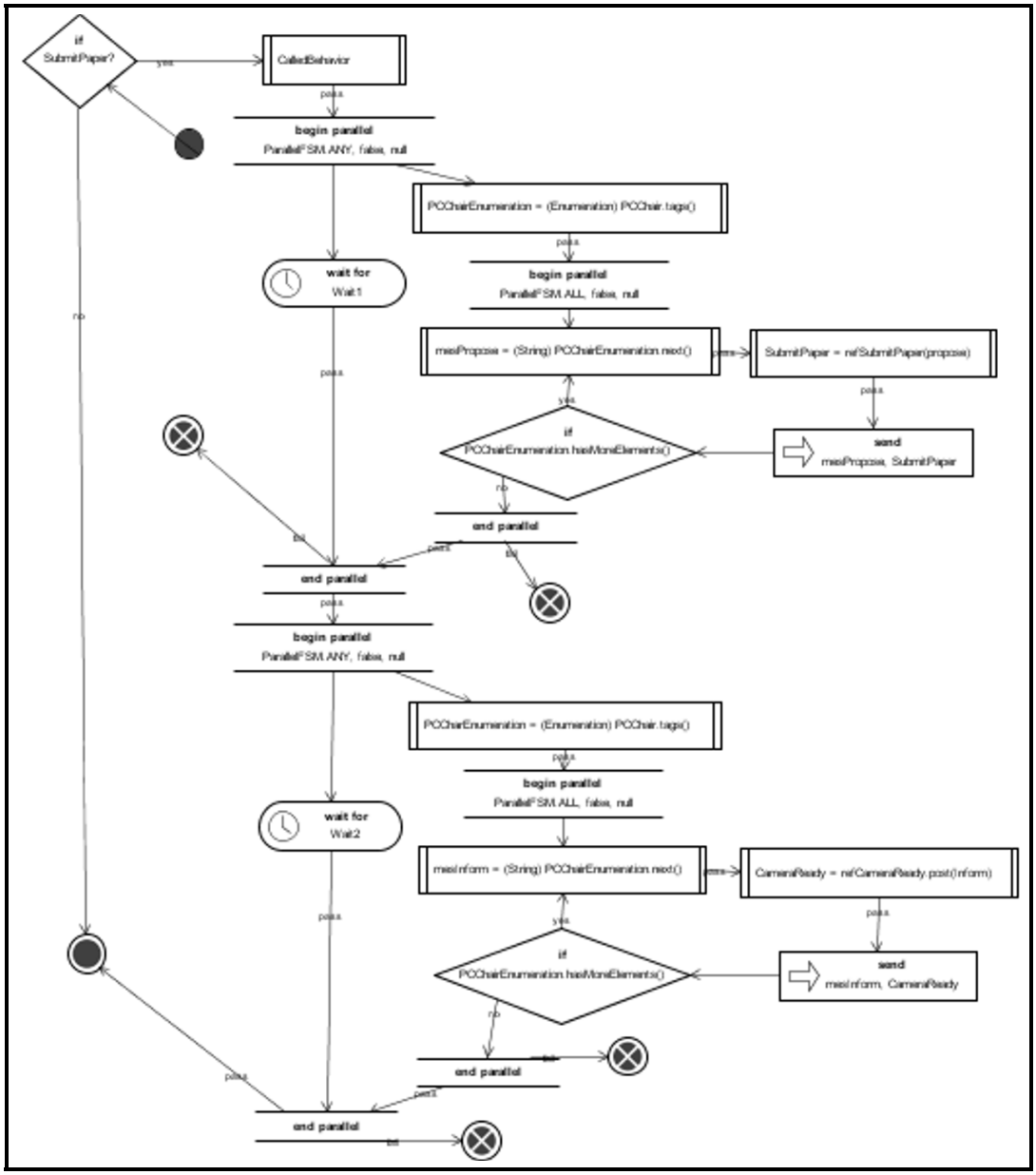


Fig. 7.9: The JACK representation of the process view of CMS.

PCChair and *PCMember* that either handle or refuse the produced events *CallForReview, Reject, Accept, PaperToReview*, and *Review*. These events are generated by Mapping Rule 7.4.

7.3.3.4 Process View of CMS in JACK

One of the generated process models of the CMS example is illustrated in Fig. 7.9. The team plan shown bases on the *SubmitPaper* behavior illustrated in Fig. 5.12. The *SubmitPaperTeamPlan* handles the *CallForPaper* event. The team plan itself consists of two parts. In the first part the team decides whether to submit a paper or not. The decision is directly derived from the decision in the *SubmitPaper* plan. If the decision evaluates to true the paper is written in the code node *CalledBehavior* within the time frame *Wait1*. After writing the papers, they are submitted in parallel using the parallel construct in JACK to all potential entities playing the *ChairActor*. If the *PaperDeadline* expires, the *ChairActor* evaluated the papers received and selected the best of them for presentation at the conference. The according authors get the *AcceptsPaper* event, the others receive a *RejectPaper* event. These events are handled outside the *SubmitPaperTeamPlan*. For each of the accepted papers, the *AuthorActor* prepares the final versions and sends them to the *ChairActor*.

7.4 PSM Agent Modeling Process

In Section 5.4.2.2, the DSML4MAS's methodology process model was presented. In this process model, the execution platforms of JACK and JADE are considered as platform-specific and thus related modeling is part of the implementation phase. The idea of combining the development process of abstract modeling languages and agent-based infrastructures is not new (e.g. (Molesini et al.; 2009)). However, in the DSML4MAS methodology, this is done in an automatic manner utilizing the model transformation between DSML4MAS on the PIM level and JACK and JADE on the PSM level.

The process model of the implementation phase is depicted in Fig. 7.10. The process takes the design (possibly including the deployment) made with DSML4MAS and starts by the decision whether to choose JACK or JADE as execution platform for the DSML4MAS design. Which agent platform to choose mainly depends on the requirements the generated implementation should fulfill, i.e., autonomous reasoning facilities provided by the intelligent entities or fast message exchange. In either case, the model transformation between DSML4MAS and the chosen execution platform is executed. For detailed information on the model transformation between DSML4MAS and JADE, we refer the interested reader to (Gründel; 2009; Hahn et al.; 2009a).

As a next step, the generated design is either refined and/or completed. As neither JACK nor JADE offers guidelines or a development process detailing how to get an executable implementation, the different tasks (e.g. `create JACK team`, `create JACK role`, etc.) are arranged in parallel. After finalizing the implementation phase, the design can be compiled, and finally, executed in a last step. This means that the design made with DSML4MAS is guided through three model transformations: The first one allows to transfer parts of the analysis phase into an architectural specification, the second and third then use the design made with DSML4MAS to generate an implementation that could—if necessary—be further refined on the different AOPLs.

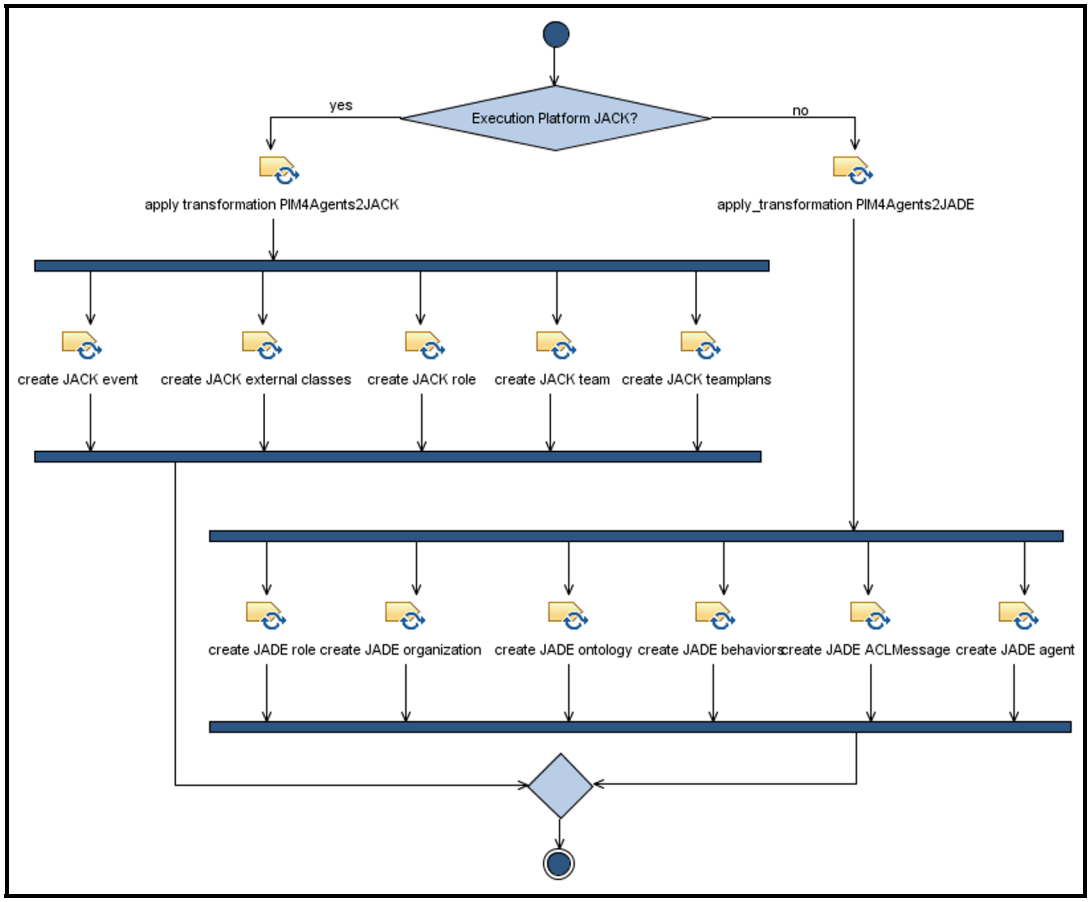


Fig. 7.10: The implementation phase of the DSML4MAS process.

7.5 Bottom Line

The current state of the art in developing MASs is to design agent systems by applying an AOSE methodology and take the resulting design artifact as a base to manually code the agent system with some agent-oriented programming platform. The fact that agent implementation is developed completely manually from the design may tend to the divergence of design and implementation.

To close the gap between design and implementation, this chapter is devoted to the implementation phase of the DSML4MAS methodology's process model with special emphasis on the model transformation between DSML4MAS and the AOPL of JACK. This transformation aims at closing the gap between design and implementation in AOSE. For this purpose, we firstly illustrated the metamodel of JACK (JackMM) and secondly defined a vertical model transformation consisting of (i) a model-to-model transformation between PIM4AGENTS and JackMM and (ii) a model-to-text transformation from JackMM to JACK GCode, an XML-like structured file that can be imported into the Jack IDE. The conference management system served as base to demonstrate how the model transformations work in practice.

8. Agent-Based Service-Oriented Architectures

To reduce the interoperability barrier between MASs and existing mainstream software engineering approaches is one of the main objectives of this dissertation. This is, in our view, of particular importance to get agent-based system design adopted by industry. Service-orientation has become the leading paradigm for modern IT system design and development as service-oriented system design has great potential for improving the efficiency and quality of the IT systems. The idea behind Service-Oriented Architectures (SOAs) is to promote services as the basic building blocks, which provide access to any type of problem solving facility regardless of its technical realization via a standardized interface (Alonso et al.; 2003). This facilitates the interoperability among heterogeneous components and resources, enables the seamless integration of previously separated systems, and supports the reuse and substitution of system components by decoupling the usage of IT facilities from their actual implementation (Erl; 2005).

As previously discussed, for achieving interoperability between two modeling tools in terms of transparent model exchange, current best practices (cf. (Tratt; 2005)) comprise creating model transformations based on mappings between concepts of the different metamodels. Therefore, to reduce the interoperability gap between SOA and MASs, we select an MDD approach and accordingly developed a generic model transformation between SoaML—the new standard for modeling SOAs proposed by the OMG—and DSML4MAS.

Scope of this Chapter SOAs as an approach to design and implement modern information systems aim to support business process management within an organization and across organizational borders. At this, services are employed to perform tasks within these processes and processes themselves can again be exposed as services. SOAs as an architectural style for distributed systems are nowadays considered as mainstream in enterprise computing. Compared to earlier approaches, SOAs put a stronger emphasis on loose coupling between the participating entities in a distributed system.

Web Services are the technology that is most often used for implementing SOAs. They are a standard-based stack of specifications that enable interoperable interactions between applications that use the Web as a technical foundation (Booth et al.; 2004). The emphasis on loose coupling also means that the same degree of independence can be found between the organizations that build the different parts of an SOA. The involved teams only have to agree on service descriptions and policies at the level of abstraction prescribed by the different Web Service standards. As various kinds of systems can be used to implement SOAs, the recent trend is to apply principles of MDD by (i) modeling SOAs in an abstract manner and (ii) providing model transformations between this abstract specification and the underlying platform-specific systems. As such, MASs became very popular as both, SOAs and MASs, share several commonalities.

In this respect, the goal of this chapter is to develop a model-driven process to automatically translate any service description made with an underlying modeling language for services into DSML4MAS. Fig. 8.1 graphically illustrates the underlying approach. Starting with defining the

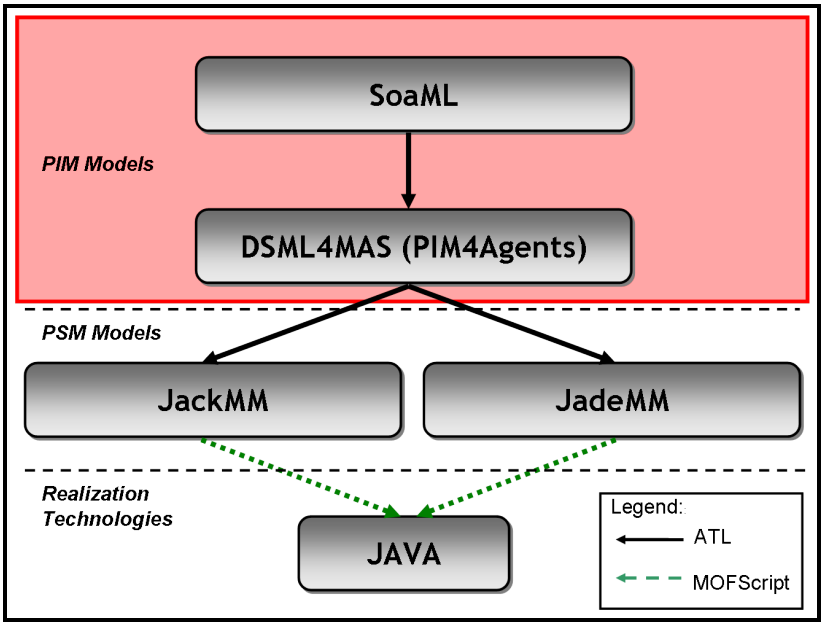


Fig. 8.1: The scope of this chapter: Model transformation between SOAs (i.e. SoaML) and MASs (i.e. DSML4MAS).

SOA using the Service-Oriented Architecture Modeling Language (SoaML), the developer takes the service-oriented design and applies the model transformation that automatically produces the corresponding agent-based description conforming to PIM4AGENTS. Afterwards, the agent-related design can be refined and used to produce AOPL-based code out of it. Therefore, the domain experts either apply the model transformation to JACK or JADE. After final refinements and adjustments, the design can be executed within the selected execution platform.

Structure of this Chapter Section 8.1 introduces the core principles of SOAs. In Section 8.2, we debate on the role of agents in SOAs and give reasons why agents are a necessary ingredient for SOAs. Subsequent, Section 8.3 introduces the new emerging standard for modeling SOA called SoaML developed under the umbrella of OMG, followed by Section 8.4 giving the abstract mapping between SoaML and PIM4AGENTS, which is the DSML4MAS's approach to abolish the technical interoperability barrier between SOAs and MASs. In Section 8.5, the process model of the DSML4MAS methodology is refined to further include SOA modeling. Section 8.6 then names the advantages when employing DSML4MAS as SOA execution engine and Section 8.7 concludes this chapter.

8.1 Service-Oriented Architectures—An Introduction

Industry is increasingly interested in executing business processes that span multiple applications. This demands high-levels of interoperability and a flexible and adaptive business process management. The general trend in this context is to have systems assembled from a loosely coupled collection of services. SOAs appear to be a natural environment in which agent technology can

be exploited with significant advantages. The remainder of this section intends to give a brief introduction on the terms SOAs and services and discuss their benefits.

The literature in the SOA field provides divers definitions of the term software architecture. For our purposes, the definition given in (Blanke et al.; 2004) of the term software architecture fits best. In accordance to Blanke et al., a software architecture "... is a set of statements that describe software components and assigns the functionality of the system to these components. It describes the technical structure, constraints, and characteristics of the components and the interface between them. The architecture is the blueprint for the system and therefore the implicit high-level plan for its construction." Two definitions of a service-based software architecture are given in the following.

Definition 8.1.1 (SOA, according to IBM)

A service-oriented architecture (SOA) is an application framework that takes everyday business applications and breaks them down into individual business functions and processes, called services. An SOA lets you build, deploy and integrate these services independent of applications and the computing platforms on which they run.

The most important statement of Definition 8.1.1 is that an SOA is an abstract specification of the service architecture that does not make any assumption about the underlying execution platform. Thus, techniques are needed to ensure that the abstract specification can be transferred to the underlying more execution-oriented platforms. MDD seams to naturally offer an environment for generating code based on the SOA description. Especially in distributed systems, like in the case of MASs, the exchange of messages is an important mechanism to coordinate the entities involved. How to interact in SOAs is often defined through the entities' interfaces, which can be used, in accordance to Definition 8.1.2, in different manners.

Definition 8.1.2 (SOA, according to Worldwide Web Consortium (2004))

A set of components which can be invoked, and whose interface descriptions can be published, discovered and invoked over a network.

SOA aims to promote software development in a way that leverages the construction of dynamic systems that can easily adapt to volatile environments and be easily maintained. The decoupling of the systems' parts enables the re-configuration of the system components according to the users needs and the systems environment. Service-oriented computing emerged as an evolution of the component-based development with the aim to support the loose coupling of system parts in a better way than existing component-based technologies. In the course of globalization, services are in particular used to provide certain (business) features to the outside that can be combined to services to support new business models.

Any service-oriented environment is expected to support several basic activities like the creation, discovery, invocation, and binding of service. In addition to these basic activities there are other activities that need to take place in order to take full advantage of SOAs. These activities include service composition, management and monitoring, billing, and security. To fulfill these basic activities, three generic roles normally interact in an SOA:

Service provider A service provider is the party in the SOA that provides software applications for specific needs as services. Service providers publish, unpublish, and update their services so that they are available on the Internet. The service provider is in general the owner of the service and thus holds the implementation of the service.

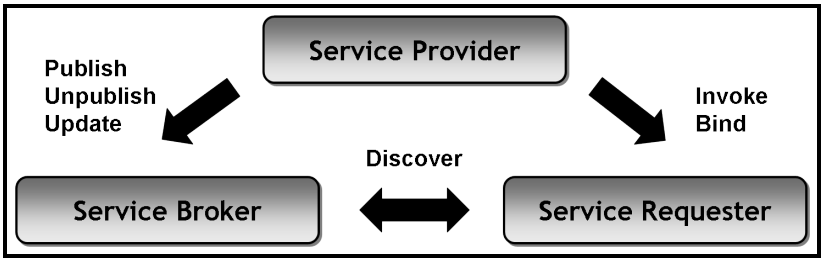


Fig. 8.2: Three generic roles of SOAs.

Service requester A requester is the party that has a need that can be fulfilled by a service available on the Web provided by the service provider. A requester could be anything, a human user accessing the service through an interface (i.e. web browser). The service could either offer a very simple functionality or complex functionalities that include other services. Optionally, if the service is not known to the requester, it finds the required services via a service broker and binds to services via the service provider.

Service broker A service broker provides a repository of service descriptions, where service providers publish their services and service requesters find services and obtain binding information for these services. The most well-known example of a service broker is UDDI (Universal Description, Discovery, and Integration). For our needs, the role of the service broker is not necessary.

8.1.1 Service

The composing of services allows presenting the logic of different levels of granularity and promotes re-usability and the creation of abstraction layers. The following definition takes some of the basic ideas of services into account.

Definition 8.1.3 (Service, according to Papazoglou and Georgakopoulos, 2003)

Services are self-describing, open components that support rapid, low-cost composition of distributed applications. Services are offered by service providers—organizations that procure the service implementations, supply their service descriptions, and provider related technical and related support.

In general, a service can be considered as mechanism to enable the access of certain capabilities. Hence, it is normally open and described through so-called service descriptions that define the capabilities provided and the information needed for access. This access is normally done through the service interfaces. A further characteristic of a service is underlined by the following definition.

Definition 8.1.4 (Service, according to Bennett et al. (2002))

A service is an application function packaged as a reusable component for use in a business process. It either provides information or facilitates a change to business data from one valid and consistent state to another.

The main characteristic of services given by Definition 8.1.4 is that they are reusable. One further important characteristic of a service, which makes it different to agents, is that services are in

general stateless, whereas agents are considered as stateful. This means that services should not be used to manage state information, rather to provide loosely coupled software systems. However, like agents, they abstract from the underlying logic. The only part of a service that is visible to the outside world is what is exposed via the service's description and formal contract. The underlying logic (beyond what is expressed in the description and formal contract) is invisible and irrelevant to service requesters. Section 8.2 is devoted to discuss the differences between services and agents in more detail.

Services are loosely coupled, dynamically locatable software pieces, which provide a common platform-independent framework that simplifies heterogeneous application integration. Services base on a SOA and communicate by exchanging messages based on XML. Some function-oriented approaches like WSLD[1] (Web Services Description Language) and BPEL4WS[2] (Business Process Execution Language for Web Services) have provided guidelines for defining service compositions (see (Koehler and Srivastava; 2003; Casati and Shan; 2001). However, the technology to compose services has not kept pace with the rapid growth and volatility of available opportunities (cf. (Sheng et al.; 2002)). While the composition of services requires considerable efforts, its benefits can be short-lived and may only support short-term partnerships that are formed during execution and disbanded on completion (cf. Sheng et al. (2002)).

Service composition can be conceived as two-phase procedure, involving planning and execution (cf. (McIlraith and Son; 2002)). The planning phase includes determining series of operations that are needed for accomplishing the desired goals from a user query, customizing services, scheduling execution of composed services and constructing a concrete and unambiguously defined composition of services ready to be executed. The execution phase involves processes of collaborating with other services to achieved desirable goals of the composed services.

8.1.2 Service Composition

Services utilized within a SOA may be provided by different organizations or partners. Often, a service provided by one organization may be constructed by aggregating services provided by other organizations. This is especially for Business-to-Business (B2B) and Business-to-Customer (B2C) transactions very common. Two styles of composing a single service from multiple services can be distinguished:

Choreography In accordance to Barros et al. (2005a), a choreography describes how a collection of services collaborate in order to achieve a common objective. It therefore describes from a global perspective how the involved services interact in terms of message exchange, time constraints and data flow. Hence, a choreography emphasizes on the interaction (i.e. the global process), but does not make any assumption on internal actions of the services. Any global process is characterized by the facts that (i) the participating services are treated equally and (ii) the control is shared among the participating partners.

Fig. 8.3 depicts a simple choreography between three services, where *Service1* invokes *Service2* and *Service3* by sending a message.

Orchestration In accordance to Barros et al. (2005a), an orchestration model describes both the communication actions and the internal processes in which a service engages. However, normally the orchestration is defined, in contrast to a choreography, from the perspective of

[1] http://www.w3.org/TR/wsdl

[2] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

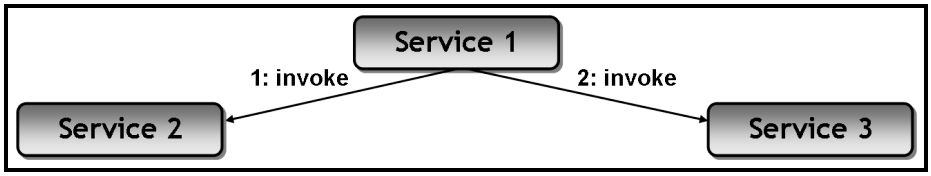


Fig. 8.3: The choreography between services.

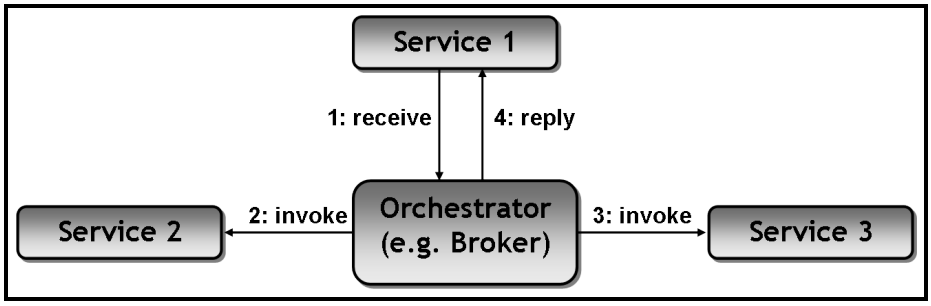


Fig. 8.4: The orchestration of services.

a single service provider. The internal processes include any related activity like invoking external legacy systems as well as information passing. In contrast to choreographies, orchestrations are executable process since they are intended to be executed by an orchestration engine like BPEL4WS.

Fig 8.4 depicts a typical orchestration of two services (i.e. *Service2* and *Service3*). At this, *Service1* requests the Orchestrator for a certain service. As such a service description is not available to the Orchestrator, it combines the capabilities of *Service2* and *Service3* by direct invocation. The result is returned to *Service1*.

Even if orchestration and choreography are normally exclusive high-level process management patterns, it is important to note that a real-life system may use a combination of both. For example, it is common for choreographed services to consist of orchestrated participants. In this setting, the global business process is defined by the choreographed interaction, but each participant may define an internal business process through orchestration.

8.1.3 Benefits of Service-Oriented Architectures

As previously mentioned, SOAs are increasingly utilized in industry as services are employed for designing the enterprise and its ICT support. In this section, we explore the benefits of SOA compared to object-orientation, component-based development, and business process engineering. We furthermore investigate how agent technologies can support the efficient service-oriented computing.

Flexibility Services orientation offers a level of flexibility exceeding that of, for instance, object-oriented development. Flexibility is in particular improved due to the fact that services are dynamically invoked. This allows the service providers to continuously improve the service's functionality. Moreover, due to the fact that the service description is offered to the

outside, the end users can search for the particular service that fits best into their needs and requirements. Similar to SOAs, one of the main characteristics of MASs is flexibility, which is achieved due to features like self-management. For instance, in case of errors during the composition or invocation of services, the BDI paradigm allows the agents to flexibly adapt to changes in the surrounding environment, i.e. the "best" service providers can be chosen to dynamically recover the service composition and to finally achieve the goal achievement.

Complexity reduction Services reduce complexity by encapsulation, which means that a service may be the aggregation of a number of other services through service composition. For the purpose of invocation, the service provider only needs to know the interface of the combined service, but not the implementation of how their behavior is coordinated. Encapsulation is the key to cope with complexity, flexibility, scalability, and extensibility. To reduce the complexity of MASs, social structures (e.g. concepts like role, institution, organization, etc.) are utilized. This has been applied, for instance, by Schillo in (Schillo; 2004) demonstrating how holonic[3] organizations improve the robustness (e.g. scalability, flexibility, etc.) of the whole MAS.

Interoperability Services promote interoperability by minimizing the requirements for shared understanding, i.e. a service description and a collaboration and negotiation protocol are the minimal requirements for shared understanding between a service provider and a service user. Moreover, by allowing legacy applications to be exposed as services, SOAs greatly facilitate a seamless integration between heterogeneous systems. New services can be created and dynamically published and discovered without disrupting the existing environment.

Loose coupling SOAs accomplish loose coupling through the use of contracts and bindings. The service requester requests the service broker for information about the type of service wanted to use. The broker returns all services it has available that match the requesterŠs criteria. The requester then selects the service provider fitting best its requirements. The idea of loose coupling is also realized in the context of MASs. Therefore, often the concept of a role is used that is performed by the agents. At run-time, only a subset of the agents that could potentially perform the role is actually playing it. Which agent is finally bound to the role is determined during run-time in accordance to some pre-defined evaluation criteria. The contract net protocol is a good example for this kind of late binding as the agents performing the participant role are dynamically allocated.

Even if SOAs and MASs share similar benefits, the agent community has lively but also wide-ranging discussion on the relationship between agents and services. The forthcoming section gives an overview on this debate.

8.2 Agents and Service-oriented Architectures

The previous section demonstrated that SOAs and MASs share several benefits, which is certainly a necessary requirement for combining both paradigms. However, up to now, it is rather unclear how both approaches fit together. The remainder of this section firstly, examines the relationship between agents and services followed by secondly, addressing how agents might improve service-based computing.

[3] see (Koestler; 1967) for a detailed description on the term holon

8.2.1 Relationship between Agents and (Web) Services

Even if the similarities between agent architectures and SOAs have already been recognized (Singh and Huhns; 2005). However, there is still an ongoing discussion about the relation between Web services and agents though. Dickinson and Wooldridge (2005) proposes three relationships between agents and services: no conceptual distinction, bi-directional integration, and agents invoke Web services. However, in our view, only the two last paradigms can actually be kept, as Payne (2008) pointed out fundamental differences between both paradigms, making the first relationship indefensible.

Flexibility and Robustness The approaches used to engineer agents and Web services are fundamentally different to the development of Web services. Typically, when designing the latter, the work flow to reach a goal is well defined at the design time and any changes in the environment under run-time could cause failure in the work flow resulting in the circumstance that the overall goal will not be reached. Since agents are reactive, flexible, social and interact with the environment, they would adopt to changing contexts and environments and either choose an alternative plan or delegate the task to other agents.

Proactiveness Since agents are communicative and social, they respond to both, messages from other agents and changes in the environment, triggering their intention to achieve some goal. This results in proactive behavior as necessary to flexibly react to changes in its surroundings. Web services are typically just reactive as they have to be triggered by some message.

Goal orientation Typically Web services are used to provide access to some resources or facilitate trading across organizations. Hence, Web services are more task-oriented, and will generally perform the task immediately in contrast to agents, which are more goal-oriented. When an agent receives a request, in order to maximize some utility through rational behavior, it can refuse to execute the task if it does not give any gain to reach its overall goal.

Autonomy In the same way as objects, Web services need to be invoked or triggered by some external parties to provide a desired behavior. As we already pointed out, an agent can evolve its own behaviors without direction from its owner or user.

These differences can mainly be attributed to the properties of the notions of agency discussed in Definition 2.1.5. Combining these findings, we can conclude that agents are stateful, meaning that they base on a kind of life cycle (potentially BDI-driven) that drives their behavior and goal achievement. In contrast, services are in general stateless. They are invoked, perform some kind of computation, and pass back the result.

Basically, we agree on the last two relationships between agents and services, however, in a model-driven context, where service are defined on a platform-independent level, we certainly prefer and support the last relationship, i.e. agents invoke the Web services defined with a service-oriented modeling language.

8.2.2 Agents in Service-Oriented Architectures

The main message from the last section is that agents and Web service share some commonalities, however, their underlying principles are different. The question now arises which kind of roles agents should take care in SOAs. In accordance to (Odell; 2007), the core features of agent-based

computing (e.g. flexibility and robustness, proactiveness, goal orientation, and autonomy) can be utilized in the following SOA-based core activities.

Resource provision and request In most of the Web Services Architectures (WSA), the actual service requester and provider will agree on the service description and potentially semantics that will govern the interaction between the requester and provider agents. The agents will then automatically perform services on behalf of the actual requesters and providers. The agents can choose the smartest and most efficient way to accomplish a task by coordinating different agents in the service provision.

Processing support For complex processes, agents can be used to orchestrate and coordinate software processes to solve problems collaboratively or compete intelligently. The provider agents determine the value proposition for the consumer agents and then identify and assemble the services that will form a virtual service for the end consumer. The concept of *self-organization* is of particular importance in this context, as autonomous service agents can freely decide to form organizations to better cope with the requirements of the service requester.

Resource discovery Dynamic service selection is increasingly common as organizations recognize the benefits of its flexibility. If an agent has flexibility in choosing its business partners, it can select them to optimize any kind of quality-of-service criteria, including performance, availability, reliability, and trustworthiness. A requester agent can be assigned to do this, but specialized service-discovery agents are sometimes useful for facilitating this. The idea of late binding is of particular importance as partner services are defined at design-time, but the actual service endpoint for a partner might be fixed at run-time, as long as the service complies with the structure defined at design time. This means that the service endpoint needs to be set at the latest point in time when the actual call to the service is done.

Middleware support Within SOA middleware, agents should be used for functions that require flexibility, autonomy, and robustness. For example, managing application-level fault tolerance, security, performance, and QoS are common uses for agents. In addition, agents can be used as a more intelligent way of handling and propagating changes in ontology and dealing with incompatibilities in vocabularies, semantics, and pragmatics among service providers, brokers, and requesters.

From our perspective, considering their special features, the central role that agents should play in a SOA scenario is to efficiently support distributed computing and to allow the dynamic composition of services. This means that agents should orchestrate services in an intelligent manner and provide (re-)planning mechanisms that allow to efficiently react to changes or perturbations in the SOA.

8.2.3 Related Work on Combining Agents and Service-Oriented Architectures

The concepts of an agent are nowadays often used in the context of SOAs. Especially, in a business context, agents are integrated in service-oriented environments, where agents mainly provide and invoke services. While the traditional workflow systems have been designed for routine tasks, e.g., administrative office processes, more flexibility is needed in business processes that comprise actions performed by customers, or that span multiple companies. Here, agent- and rule-based

approaches may be more adequate. The most flexibility is probably needed in very knowledge-intensive tasks (e.g., decision making), which can only be coarsely modeled in traditional workflow systems, resulting in a low level of system support. In the following, related work in this respect is given.

Nguyen and Kowalczyk (2006) presented a framework called WS2JADE that allows integrating Web services on the agent-platform JADE. The integration is performed through representing a Web service by a gateway agent. The WS2JADE approach allows deploying, composing, and controlling Web services as agent services at run-time.

Padgham and Liu (2007) discuss an agent-based approach to the service composition as a loose form of teamwork in JACK. In particular, this is done by incorporate Web services as team members in JACK providing the execution engine for the composite services developed. The BDI architecture of JACK further supports a failure recovery mechanism that enables failing team members to be replaced.

Jennings et al. (2000) propose ADEPT to develop agent-technology for business processes. It provides the conceptualization and implementation of an agent-based system for managing corporate-wide business processes. The ADEPT philosophy was founded upon two key notions: (i) developing responsibility for provisioning and managing the business, and (ii) making the problem-solving components reactive and proactive so they can respond to unexpected situations.

Kinny (1999) proposes the Agentis framework. In this framework, a service agent receives a service request, is tasked with achieving the service objectives and keeps track of its environment in terms of the service context. The agents use a library of sub-processes, each described with its process objectives, process context and process logic. The sub-process logic itself is similar to standard workflow systems, and is described using an extension of UML activity diagrams.

Nissen (2000) proposes a design of a set of agents to perform activities associated with the supply chain process in the area of E-Commerce. In (Stormer and Knorr; 2001), the agents have been used as part of the infrastructure associated with the workflow management system itself in order to create an agent-enhanced workflow management system (WfMS).

Zeng et al. (2001) propose an approach that combines agents with workflows to effectively integrate cross-enterprise workflows. Agents are used to encapsulate (i.e., wrap) services which are able to execute workflow tasks. Based on the requirements of tasks, the system searches for agents with matching capabilities. The relevant agents are used to execute the tasks, which are dynamically composed by the system in order to provide the whole service.

Savarimuthu et al. (2005) discuss how an agent-based architecture can be used to bind and access Web services in the context of executing a workflow process model. They use an example from the diamond processing industry to show how an agent architecture can be used to integrate Web services with WfMS.

Blake and Gomaa (2005) describe an adaptation of software agents as a possible solution for the composition and enactment of cross-organizational services. Their approach details design aspects of an architecture that would support this evolvable service-based workflow composition. The internal coordination and control aspects of such an architecture is addressed. These agent developmental processes are aligned with industry-standard software engineering processes.

8.2.4 Model-Driven Integration of Agents and Service-Oriented Architectures

Apart from the wealth of literature about business process modeling, enterprise application integration and SOAs, to our knowledge, a model-driven approach for the integration of MASs and SOAs has not been investigated so far. Cabri et al. (2007) provide an overview of agent-based modeling approaches for enterprises. Penserini et al. (2006) describe the Tropos[4] methodology for a model-driven design of agent-based software systems. However, the problems related to the integration of agent platforms and SOAs are beyond their focus. Endert et al. (2007) map BPMN models to BDI agents, but do not consider a model-driven integration of agents and Web services. In recent years, we investigated in two EU projects the model-driven integration of SOAs and (BDI-) agents. The details of these approaches are given in the remainder of this section.

8.2.4.1 The ATHENA project

In the context of the integrated EU FP6 project ATHENA, we developed a model-driven approach for BDI agents based on the JACK development environment. One of the main ideas of ATHENA was to demonstrate how models, which were defined according to the platform independent metamodel for SOAs (called PIM4SOA, (Benguria et al.; 2007)) can be transformed into models that can be compiled into executable code using the metamodel definition for JACK (see (Fischer et al.; 2007; Hahn et al.; 2006b) for a detail discussion of the transformations between PIM4SOA and JackMM). Furthermore, in order to use a Web service within plans of JACK agents, we defined a second transformation that maps the concepts of a metamodel for WSDL (Web service description language) to particular concepts of the JACK metamodel (e.g. Capability). Detailed information on the model-driven framework for the integration of services into agent systems can be found in (Zinnikus et al.; 2007).

The PIM4SOA metamodel was one of the first attempts toward a metamodel for SOAs on a more abstract or platform independent level. However, its expressiveness is limited to the design of rather simple SOA-based scenarios. The PIM4SOA metamodel defines modeling concepts that can be used to model four different aspects of SOAs: services, information, processes and non-functional aspects. The definition of these aspects has been influenced by ongoing standardization initiatives in the area of Web services, standards of SOAs were not considered.

8.2.4.2 The SHAPE Project

SHAPE (short for Semantically-enabled Heterogeneous Service Architecture and Platforms Engineering) was a European Research Project under the 7th Framework Program that develops an infrastructure for the model-driven engineering of service-oriented landscapes with support for various technology platforms and extensions for advanced service provision and consumption techniques. For this, the project defined the necessary metamodels for describing services and related aspects in heterogeneous technology landscapes, developed an integrated tool suite for modeling and deployment, and provided a comprehensive methodology for guiding software engineers and architects through the engineering process of heterogeneous service-based systems. This brought together the world of MDD with the SOA paradigm and integrated other technologies like agents, Semantic Web, Grid, and P2P, combining their respective advantages for the effective development and maintenance of high quality integrated IT systems.

[4] Section 10.2.7 gives further details on the Tropos metamodel and methodology.

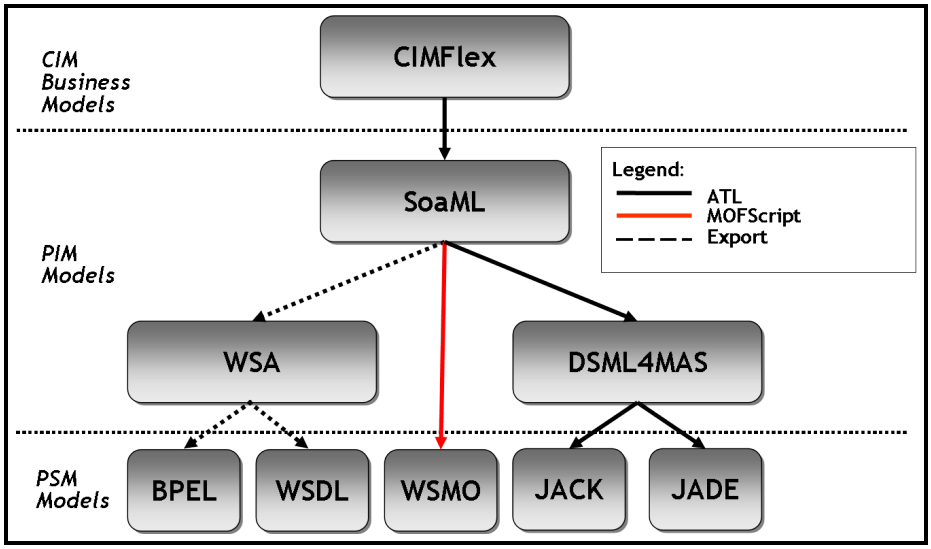


Fig. 8.5: An overview of the SHAPE model transformation architecture and framework.

A central part of the SHAPE technology are model transformations that define the basis for (semi-) automated transformations among several model types, and in particular enable the MDD-based approach for integrated top-down modeling from the CIM level to the PSM level.

The model transformation architecture of SHAPE (cf. Fig. 8.5) illustrates the core language used, their relationship to the abstraction levels CIM, PIM and PSM as well as their relationship to other languages defined through model transformations, either model-to-model or model-to-text. On the highest level, business models encompass business rules, processes, services and other issues such as contracts involving humans and organizations to achieve business goals. These conform to the metamodel of CIMFlex[5] (Hahn et al.; 2010d; Sadovykh et al.; 2009; Elvesæeter et al.; 2010). The middle layer contains the extended SOA models, i.e., the standardized SoaML and extensions for semantically-enabled heterogeneous architectures (ShaML). This model transformation architecture allows the realization of one of the main goals of SHAPE namely to provide a transformation engine that maps business models to SOA/SHA models, which are then transferred to the various execution platforms. Apart from the model transformations available in the DSML4MAS methodology, the model transformation architecture of SHAPE supports the following model transformations:

CIM to PIM transformation: Model transformation between CIMFlex and SoaML The challenge in transforming CIMFlex models to SoaML is to generate the appropriate system relevant constructs for SoaML according to the generic business context on CIM level. CIMFlex supports the model-to-model transformation by making use of ATL.

PIM to PIM transformation: Model transformation between SoaML and Web Services Transferring SoaML models into Web Service models is done through a model-to-model transformation. This transformation produces three kinds of models from a single SoaML model, i.e., it produces an XML schema for information description, WSDL files

[5] CIMFlex is a modeling tool that facilitates modeling of Event-Driven Process Chains (EPCs) (Mendling; 2009) of ARIS and BPMN.

for interface description, and finally BPEL4WS files for behavioral (i.e. orchestration) specification.

PIM to PSM transformation: Model transformation between SoaML and WSMO The model transformation between the metamodel of SoaML and the Web Service Modeling Ontology (WSMO) is specified through a model-to-text transformation using the MOFScript engine.

8.3 Service-Oriented Architecture Modeling Language

Similar to the PIM4SOA metamodel, in recent years, several metamodels and UML profiles for SOAs were developed. For example, in (Johnston; 2006), a UML profile for software services is described, which mainly focuses on the structural aspects of service modeling. In contrast, UML4SOA (Mayer et al.; 2008) as UML extension, mainly focuses on service orchestration and thus on the process modeling. Moreover, independent organizations like W3C (World Wide Web Consortium) or OASIS have worked on a common definition of SOAs. W3C, for instance, defined the Web Service Architecture (WSA[6]), which intends to provide a common definition of a Web service. The OASIS Reference Model for Service Oriented Architecture[7] is, in opposition, an abstract framework for understanding significant entities and relationships between these entities within a service-oriented environment.

However, although there are several recent and ongoing activities on developing MDD-based techniques for supporting the design, development, and maintenance of SOAs (e.g. (Bell; 2008)), there does not exist a standardized metamodel for describing services along with the relevant aspects on their provision and usage in a service-oriented system landscape.

In 2006, the OMG started the standardization process for SOA by issuing a request for proposal for an UML Profile and Metamodel for Services (UPMS). The main objectives of this new standard for services is to (i) enable interoperability and integration at the model level, (ii) enable SOAs on existing platforms through MDA, and (iii) allow for flexible platform choices.

The resulting Service-Oriented Architecture Modeling Language[8] (Object Management Group; 2009b)—a revised submission currently under review—is based on the UML 2.0 metamodel and provides minimal extensions to UML, only where absolutely necessary to accomplish the goals and requirements of service modeling. Like any UML profile, SoaML provides a UML specific version of the metamodel that can be incorporated into standard UML modeling tools.

8.3.1 Modeling Concepts of SoaML

SoaML addresses the conceptual and technological interoperability barrier. It aims at defining platform independent modeling language constructs that can be used to design, re-architect and integrate ICT infrastructure technologies supporting SOA. In the SHAPE project, SoaML is in particular used to achieve the following two objectives:

- SoaML should bridge the gap between the business analysts and the IT developers and support mapping and alignment between enterprise and IT models.

[6] http://www.w3.org/TR/ws-arch/wsa.pdf
[7] http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
[8] http://www.omg.org/docs/ad/08-05-03.pdf

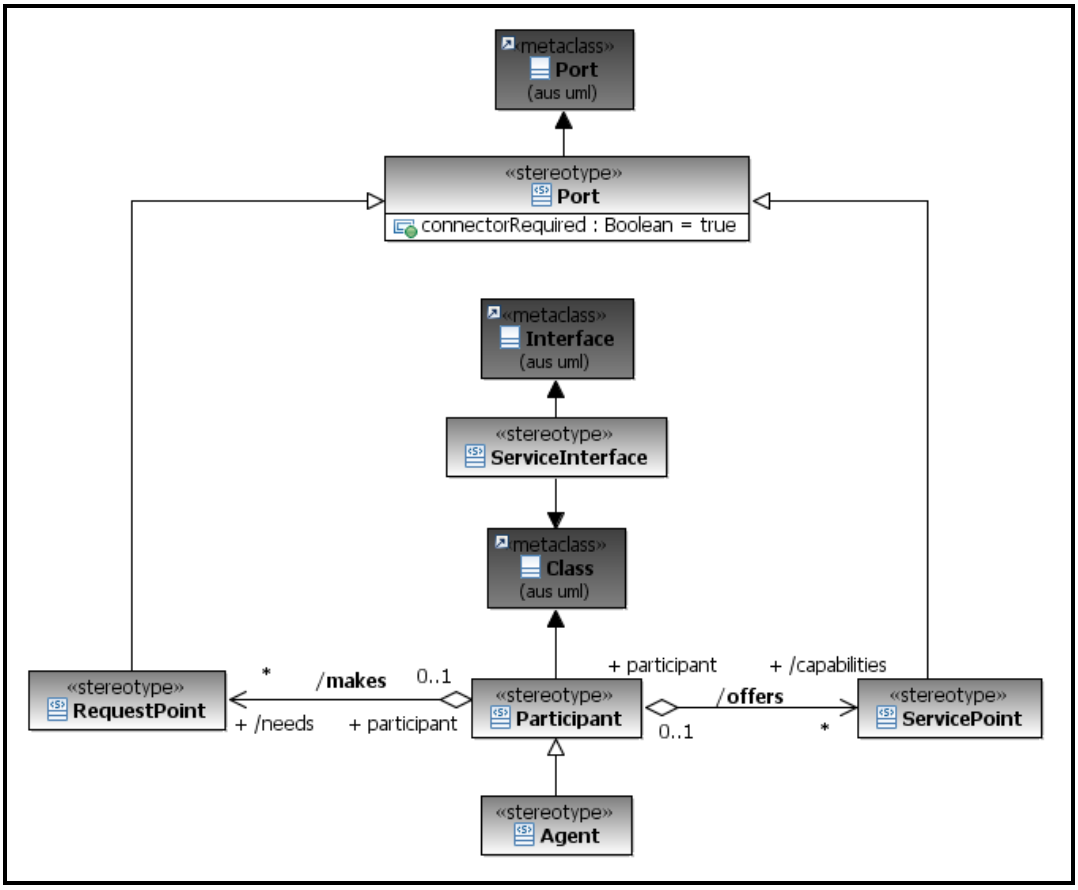


Fig. 8.6: The basic concepts to define SOAs in accordance to SoaML.

- SoaML should define a platform independent abstraction that can be used to integrate and define mappings to Web services, agents, Semantic Web services, peer-to-peer (P2P) and Grid execution platforms.

As aforementioned, SOAs are a way of organizing and understanding organizations, communities and systems to maximize agility, scale and interoperability. SoaML claims to be very flexible in order to support activities of service modeling and design, and to fit into an overall MDD approach. It supports three different perspectives: (i) from the service consumer requesting the service, (ii) service provider advertising a service to those who are interested and qualified to use it and (iii) from a system design describing how consumers and providers will interact to achieve overall objectives.

Moreover, SoaML can be used for both, *bottom-up* as well as *top-down* architecture designs of services (Berre; 2008). For bottom-up definition, services can be considered as atomic and are combined on a low level. In the latter case, the services are very complex and hide necessary functionalities from more atomic services. Fig. 8.6 depicts the core concepts of SoaML.

Even if SoaML provides core concepts for modeling services, neither it specifies any methodology for service design nor makes any assumption about brokering, publishing, discovery, addressing

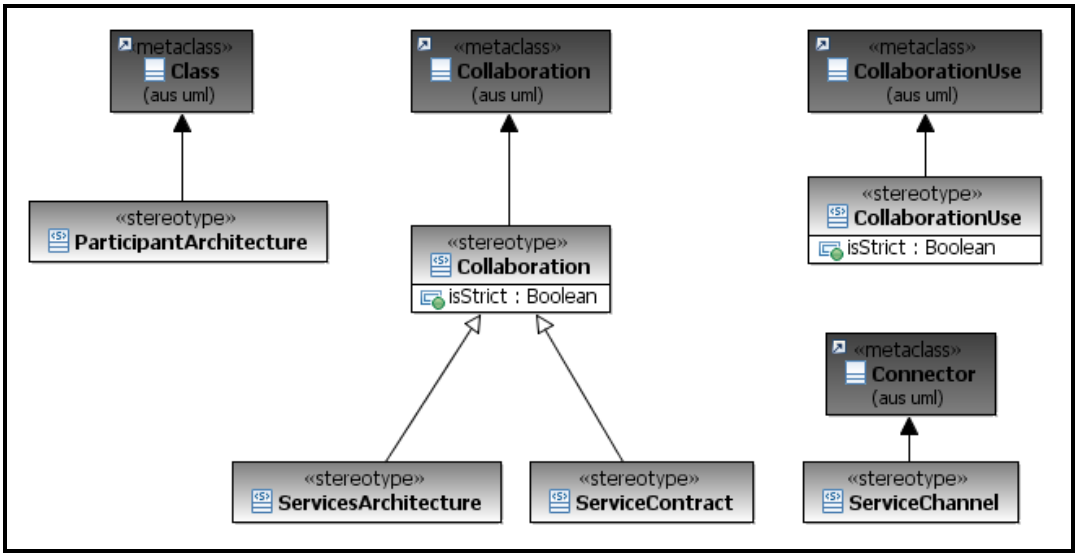


Fig. 8.7: The basic concepts of contracts in accordance to SoaML.

of services or any service runtime, configuration and deployment (e.g. dynamic binding etc.). In the remainder of this section, we discuss the core concepts of SoaML in more detail and illustrate their usage in a small illustrative example.

8.3.1.1 Participant

The key concept of a service is a capability offered by one entity or entities to others through well-defined interfaces. Those entities may be people, organizations or systems and are called *Participants*. In SoaML, capabilities are provided or required by *Participants* through the interaction points, i.e. the *UML Ports*. To express that a *Participant* acts as service provider, the certain *UML Port* is stereotyped as *Service*. In contrast, if the capabilities are required by the *Participant*, meaning that the it acts as service requester, the *UML Port* is stereotyped as *Request*.

The *Service* port is the interaction point where requesters of the service use that particular service. Hence, the *Request* port is the interaction point where the service is consumed. In either case, the port has a type, which describes how to use that service that may be either a *UML Interface* (for very simple services) or a *ServiceInterface* (see Section 8.3.1.2). The type of the service port specifies, directly or indirectly, everything that is needed to interact with that service. This means that it describes the contract between the providers and requesters of that service. The *Request* port is the conjugate of a *Service* port in which it defines the use of a service.

The *Participant* stereotype in SoaML is inheriting from the *UML Component* classifier, and is mainly used to represent software components, organizations, actors or individuals within SOAs. Any *Participant* is defined by the roles it is playing in the various *ServicesArchitectures* (cf. Section 8.3.1.2) and the *Capabilities* (cf. Section 8.3.1.8) it is providing to and requesting from other *Participants*.

8.3.1.2 ServiceInterface

The capabilities and needs of a *Service* or *Request* port are defined through its type, *ServiceInterface* or *UML Interface*. In accordance to (Object Management Group; 2008b), the *ServiceInterface* stereotype is like an interface, but has the additional feature that it can specify a bi-directional service, where both, the provider and consumer, have responsibilities to send and receive messages. The *ServiceInterface* is defined from the perspective of the service provider, having three primary parts (cf. (Object Management Group; 2008b)):

Interfaces are standard *UML Interfaces* that are either realized or used by the *ServiceInterface*. The realized *UML Interface* specifies the provided capabilities, the messages that will be received by the provider (and correspondingly sent by the requester). The *UML Interface* used by the *ServiceInterface* defines the required capabilities, the messages or events that will be received by the requester (and correspondingly sent by the provider). For basic services, i.e., services that are not defined through a contract, two specialized *UML Interfaces* are utilized (see Fig. 8.6). The *ServicePoint* defines the place, where a service is provided, the *RequestPoint* defines the place of a *Participant*, where the service is requested.

ServiceInterface specifies the roles that are performed by the involved entities. They are necessary to define who is providing or requesting a certain service. Hence, the role that is typed by the realized *UML Interface* is played by the service provider, the role that is typed by used *UML Interface* is played by the service requester.

Behavior specifies the interaction between service provider and requester. This is done by any form of interaction protocol that establishes the contract between the roles involved without specifying the internal and private processes. Any *UML Behavior* can be used for specifying the interaction protocol, however, the UML activity diagrams are the most common as they offer both, describing a behavior from the perspective of a single entity (i.e. orchestration) as well as from the perspective of several entities (i.e. choreography) through partitions.

8.3.1.3 Service Contract

In accordance to (Object Management Group; 2008b), a key part of a service is the *ServiceContract* that defines the terms, conditions, interfaces, and choreography that interacting *Participants* must agree to for the service to be enacted. Beside non-functional criteria, the *ServiceContract* is the full specification of a service, which includes all the information, choreography and any other characteristics of the service.

Due to the fact that a *ServiceContract* is stereotyped by *UML Collaboration*, it defines well-defined roles for the service provider and requester. A *Participant* plays a role in the larger scope of a *ServicesArchitecture* and, consequently, also either plays the provider or requester role within the contained *ServiceContracts*. The role of a service broker is normally not modeled in SoaML. In terms of UML, a *UML Collaboration* is a type of structured classifier in which roles and attributes cooperate to define the internal structure of a classifier.

A choreography is certainly an important part of a *ServiceContract* to specify how the roles of the contract exchange information through messages. Important to note is that the choreography defines the message exchange between the provider and requester participants without defining their internal processes. However, the internal process of any entity has to be compatible with the

entity's role within the *ServiceContract*. The choreography of a *ServiceContract* is defined by its *ownedBehaviors* attribute which refer to, for instance, an UML activity diagram.

Similar to the manner *CollaborationUses* are used for collaborations in UML, the concept of a *CollaborationUse* in SoaML allows to define the structure and behavior of the contained parts playing roles in a specific context of *ServicesArchitectures* and *ServiceContracts*. Thus, it exactly represents one particular use of a *UML Collaboration* by specifying the role bindings that binds each role of its *Collaboration* to a part.

8.3.1.4 Services Architecture

A *ServicesArchitecture* provides a top-down view on the composed service. In accordance to (Berre; 2008), the *ServicesArchitecture* is a network of participant roles providing and requesting services to fulfill a purpose. It defines the requirements for the types of *Participants* and service realizations that fulfill those roles. As depicted in Fig. 8.7, the *ServicesArchitecture* is defined as *UML Collaboration* to specify the set of roles collaborating under certain conditions. In the context of SoaML, the roles are normally filled with *Participants* playing a certain position in this *ServicesArchitecture*. A role defines how entities are involved in that collaboration (how and why they collaborate) without depending on what kind of entity is involved (e.g. a person, organization or system). Each use of a service in a *ServicesArchitecture* is represented by the use of a *ServiceContract* bound to the roles of participants in that architecture. Both service contracts and participants can be reused when composing different services in other *ServicesArchitectures*.

8.3.1.5 Participant Architecture

The concept of *ParticipantArchitecture* is the high-level services architecture of a participant. It defines how a set of internal and external *Participants* or *Agents* use services to implement the responsibilities of the corresponding *ParticipantArchitecture*. Any *ParticipantArchitecture* is implemented by a *Participant* or *Agent*. In contrast to a *ServicesArchitecture*, a *ParticipantArchitecture* provides and requires services through the *ServicePoint* and *RequestPoint*, respectively.

8.3.1.6 Agent

In SoaML, the *Agent* concept extends *Participant* with the ability to be active, participating components of a system. *Agents* in SoaML are specialized as they (i) possess the capability to offer and request services and (ii) can have an internal structure and ports. They collaborate and interact with their environment. An *Agent's* behavior is treated as its own tread of control, life-cycle, or what defines its emergent or adaptive behavior.

8.3.1.7 Message Type

As previously mentioned, for the purpose of defining choreographies, SoaML provides UML behaviors like activity or sequence diagrams to specify the message exchange between requesters and providers. The kind of information that is exchanged between the partners is defined in the message diagram using the *MessageType* concept. Hence, a *MessageType* as a kind of *ValueObject*[9] is mainly used to explicitly identify the data and information to be exchanged through a

[9] A *ValueObject* is a classifier that can be freely interchanged between participants.

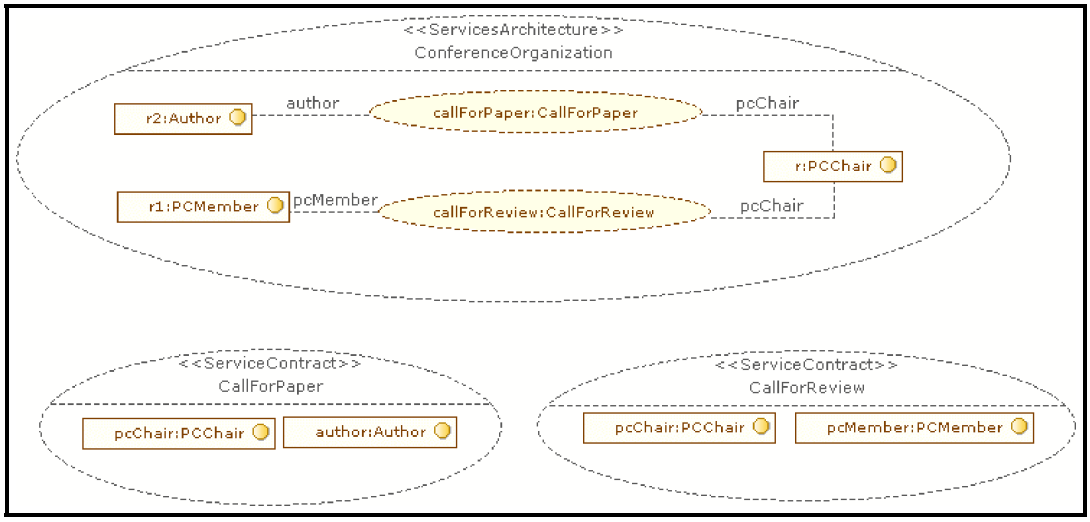


Fig. 8.8: The abstract view on the conference management system using the ServicesArchitecture concept from SoaML.

service. A *MessageType* describes domain or service-specific content and does not include any protocol-specific information. A *MessageType* is stereotyped as a *UML DataType* and thus can have associations (i.e. aggregation, composition) with other *MessageTypes* or *DataTypes* to define complex objects.

8.3.1.8 Service Capability

As aforementioned, SoaML offers two opportunities to describe services: from a global perspective using *ServicesArchitectures* and from a local perspective through *ParticipantArchitectures*. A third option is to describe a combined service on a more abstract level, regardless of whether the service is provided or requested. This can be expressed through the concept of a *ServiceCapability*, which is stereotyped by *UML Class*. The *ServiceCapability* allows organizing functionalities needed to provide, i.e., a *ServiceCapability* may use other *ServiceCapabilities*, which define the functionalities needed by the super *ServiceCapability*. For describing the functionalities, a *ServiceCapability* may have any kind of *UML Behaviors* defining the methods of its provided operations and are linked to *Participants* or *ParticipantArchitectures* that provide the certain capability through *UML Realizations*.

8.3.2 Illustrative Example: Conference Management System using SoaML

To illustrate the concrete usage of SoaML we again apply the conference management system (CMS) as example. The CMS already served as example when discussing the concrete syntax of DSML4MAS in Section 5.3.2.1 and demonstrating the mappings between DSML4MAS and JACK in Section 7.3.3. Hence, the reader is already familiar with the basic architectural requirements of this scenario and can focus on how these requirements are designed when applying the service-oriented approach of SoaML. In order to model the CMS example with SoaML, we used the Modelio

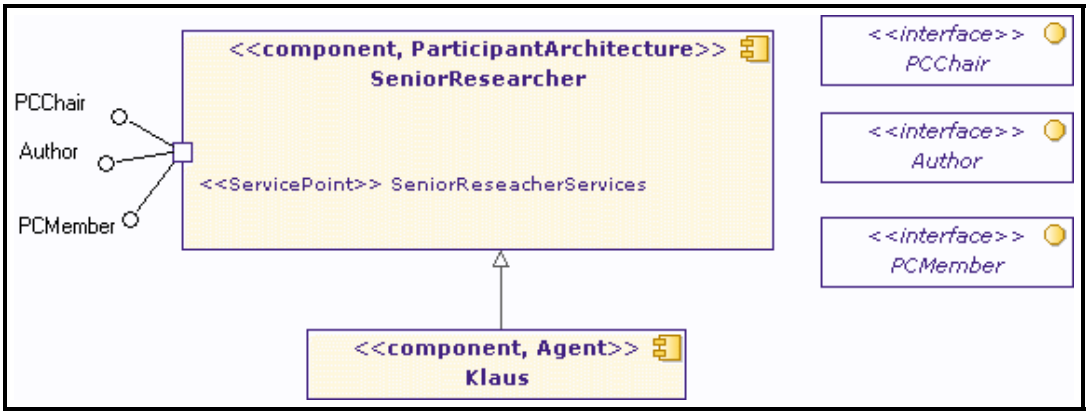


Fig. 8.9: The abstract view on the *SeniorResearcher* participant architecture.

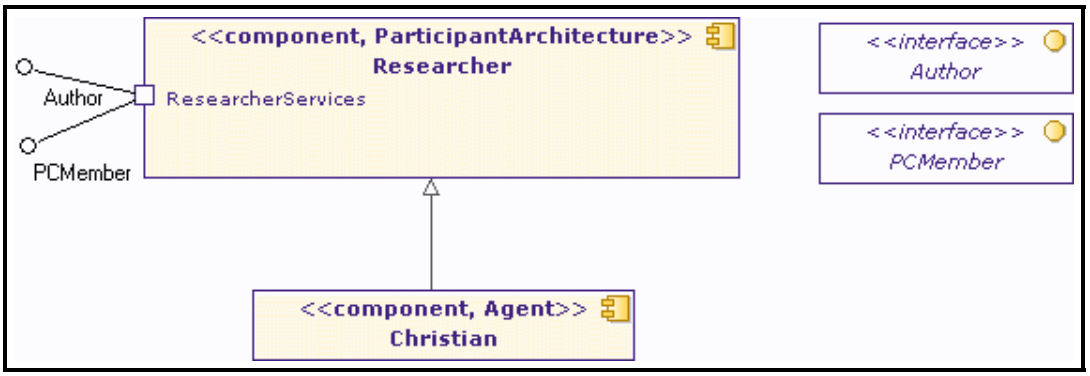


Fig. 8.10: The abstract view on the *Researcher* participant architecture.

tool suite also supporting SoaML. Modelio is a commercial product of the French company SOFTEAM[10]

In order to model the relationship between all entities involved, we choose the concept of *ServicesArchitecture* as it allows defining how kinds of entities work together for some purpose. Fig. 8.8 depicts the *ConferenceOrganization* defined as services architecture.

The collaboration roles (i.e. *PCChair*, *Author*, and *PCMember*) act as placeholder for the participants that realize the services architecture. They define how the entities inside the services architecture interact. For this purpose, two collaboration uses (i.e. *CallForPaperUse* and *CallForReviewUse*) are defined. Those refer to their particular service contract (i.e. CallForPaper and CallForReview) defining their type (cf. Fig. 8.8).

The concrete entities performing the *author*, *pcChair*, etc. roles are specified using the *ParticipantArchitecture* concept of SoaML. This means that, as depicted in Fig. 8.9 and Fig. 8.10, both the *SeniorResearcher* and the *Researcher* are modeled as *ParticipantArchitectures*. The *SeniorResearcher* offers the *SeniorResearcherServices* service that includes the *PCMember*, *Author*, and *PCChair* interfaces. The *Researcher*, on the other hand, offers the *ResearcherServices* service that

[10] SOFTEAM is one of the most renowned specialists in modeling techniques in Europe, http://www.softeam.com/

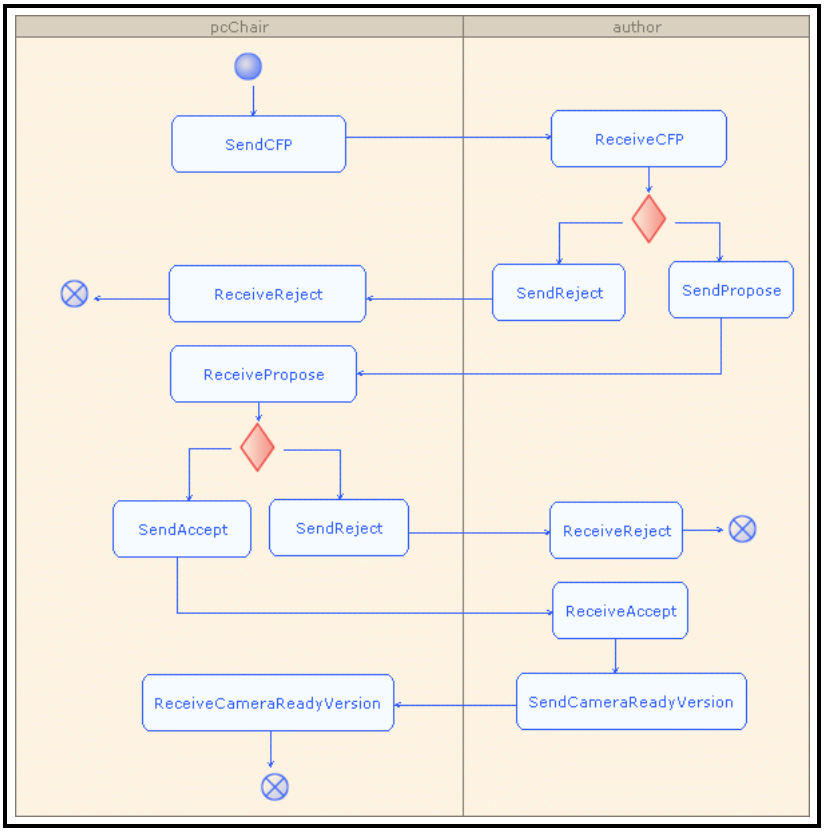


Fig. 8.11: The concrete interaction of the *CallForPaper* service contract.

includes the *PCMember* and *Author* interfaces. The *SeniorReseacher* is implemented by the agent *Klaus*, the *Researcher* is implemented by the agent *Christian*.

The concrete interaction between the *author* and *pcChair* roles is depicted in Fig. 8.11. The activity diagram describes how a "call for paper" request is sent by the *author* role through the *SendCFP* action. The request is received by the *pcChair* role and evaluated. Based on the evaluation, either the request is rejected by the *SendReject* action, or proposed by the *SendPropose* action. After getting the accept message through the *ReceiveAccept* action, a camera ready version is generated by the *SendCameraReadyVersion* action. Otherwise, if the author receives a reject, the interaction process stops.

8.4 Model Transformation: From SoaML to DSML4MAS

In Chapter 7, we presented how to derive code from a DSML4MAS design by depicting the model transformations between DSML4MAS and JACK. However, as MASs do not exist in pure isolation, mechanisms need to be explored how to combine MASs with other available software engineering approaches. As SOAs and their corresponding modeling language SoaML describes IT system in a very abstract manner, they provide a nice opportunity to illustrate how to utilize agent-based computing in service-oriented environments.

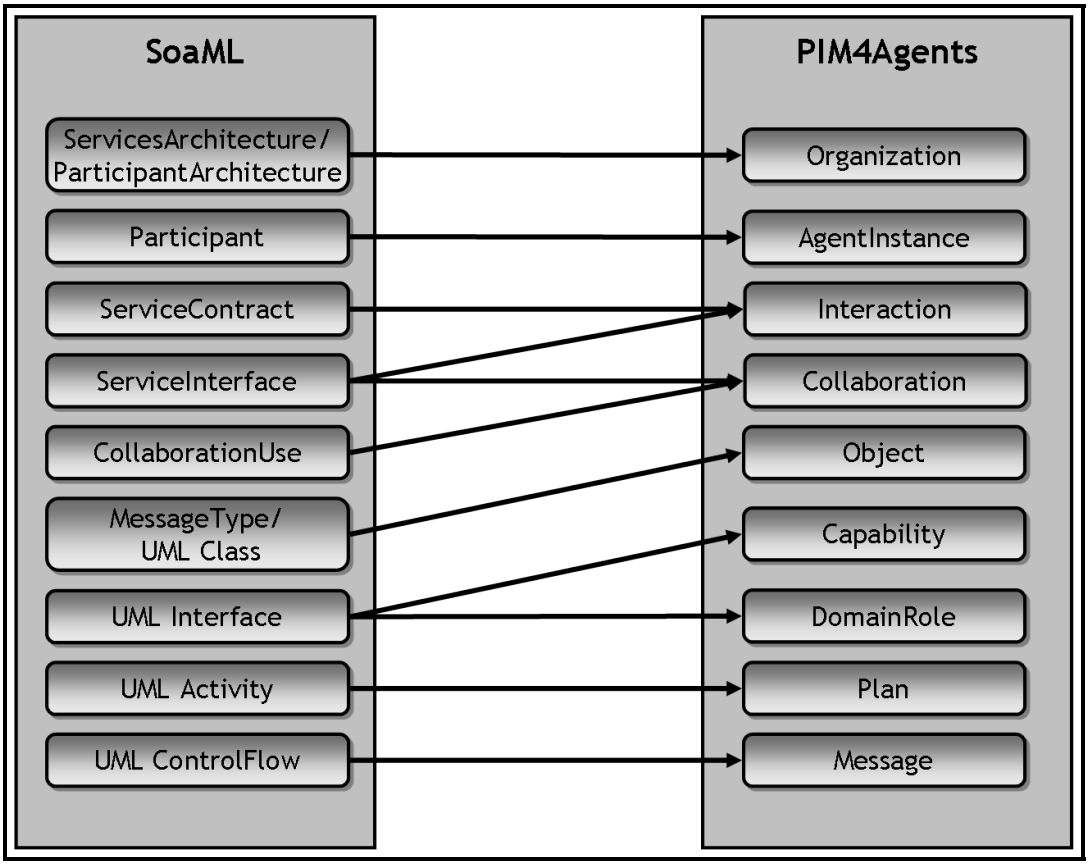


Fig. 8.12: The basic mapping rules to transform SoaML specifications into PIM4AGENTS.

8.4.1 Model-to-Model Transformation: From SoaML to PIM4AGENTS

Similar to the code generation techniques presented so far, the approach chosen for combining SOAs and MASs bases on the principles of MDD. In this particular case, this is done through a generic model transformation between the SoaML profile and PIM4AGENTS. By comparing SoaML and PIM4AGENTS, we derive a set of basic mapping rules, which are depicted in Fig. 8.12 and precisely examined in the remainder of this section.

The first mapping rule investigates the mapping between the concepts of *ServicesArchitecture* in SoaML and *Organization* in PIM4AGENTS.

Mapping Rule 8.1: SoaML:ServicesArchitecture → Organization

- **name**: name of the *ServicesArchitecture*
- **requiredRole**: collection of roles part of the *ServicesArchitecture*
- **interaction**: UML Activity diagrams used by the *ServicesArchitecture's ServiceContracts* to specify he interactions between the entities involved
- **organizationUse**: any *CollaborationUse* defining how the involved roles collaborate

As Mapping Rule 8.1 illustrates, the concept of a *ServicesArchitecture* nicely corresponds to the concept of an *Organization* in PIM4AGENTS as both refer to roles that interact in accordance to some predefined processes. However, and this is the main differences between both constructs, a *ServicesArchitecture* does not perform any role to the outside, which is the case for *Organizations*. Hence, the *Organizations* generated on the base of the *ServicesArchitectures* are utilized as a social structures providing the space for interaction. In consequence this means that the generated *Organizations* do neither own *Plans* nor perform *DomainRoles* to the outside. Thus, the *Organizations* are not autonomous entities in the service-oriented MAS, but should rather be considered as a form of grouping the necessary autonomous entities to fulfill service. Likewise, the resulting *Organizations* do not own any kind of *knowledge*, *capability*, nor *resource*.

In the same way as a *ServicesArchitecture* specifies the top-down view on a service, applying service choreographies to describe the interaction between its roles, a *ParticipantArchitecture* defines how a service is orchestrated. Since an *Organization* in PIM4AGENTS offers mechanisms to describe the interaction from a global but also from a local perspective, an *Organization* is also the best match for a *ParticipantArchitecture* as expressed by Mapping Rule 8.2. One might think that a *Participant's* perfect match would be the *Agent* concept on the PIM4AGENTS side. However, the semantics of both terms is different, as the *ParticipantArchitecture* requires roles as parts, the *Agent* concept in PIM4AGENTS, in contrast, only performs roles to the outside, but does not require any role (cf. Section 4.3.1). Nevertheless, as an *Organization* in PIM4AGENTS is defined as specialization of *Agent*, we do not loose any expressiveness when using the *Organization* concept as target instead.

Mapping Rule 8.2: SoaML:ParticipantArchitecture → Organization

- **name**: name of the *ParticipantArchitecture*
- **requiredRole**: collection of roles the *ParticipantArchitecture* makes use of
- **interaction**: any UML Activity diagram used within the *Participant's ServiceContracts* (cf. Mapping Rule 8.3)
- **organizationUse**: collection of *CollaborationUses* utilized within the *ParticipantArchitecture* (cf. Mapping Rule 8.5)
- **performedRole**: collection of roles the *ParticipantArchitecture* is bound to within any *ServiceContract*, *ServicesArchitecture*, or *ParticipantArchitecture* outside its own boundaries
- **capability**: any *UML Operation* provided by its *Service* port (cf. Mapping Rule 8.8)
- **behavior**: collection of *UML Behaviors* constraining the *ParticipantArchitecture's* behavior (cf. Mapping Rule 8.7)
- **knowledge**: collection of *UML Classes* (expressed through the *ownedAttribute* attribute) the ParticipantArchitecture makes use of (cf. Mapping Rules 8.11 and 8.10)

In contrast to a *ServicesArchitecture*, a *ParticipantArchitecture* depicts a concrete entity in the system described. Thus, the target *Organization* may perform a *DomainRole*, which is either required

inside any *ServicesArchitecture* or *ServiceContract* or even in other *ParticipantArchitectures*. Moreover, the *Organization* may own certain *knowledge* used by the source *ParticipantArchitecture*.

As previously mentioned in Section 8.3.1.3, the main purpose of a *ServiceContract* is to define the roles that agree on the contract and how they interact with each other which is expressed through any kind of *UML Behavior*[11]. Hence, for adequately representing a *ServiceContract* in PIM4AGENTS, the right choice is a *Collaboration*. This generated *Collaboration* defines how the *DomainRoles* of its *Organization* are bound to *Actors* of *Interactions*.

Mapping Rule 8.3: SoaML:ServiceContract → Interaction

- **name**: name of the *ServiceContract*
- **actor**: collection of the *ServiceContract's collaborationRoles*[12]

Apart from *ServiceContracts*, we also apply Mapping Rule 8.3 when transforming a SoaML *Collaboration*, as a *Collaboration* is the generalization of a *ServiceContract* (cf. Fig. 8.7). For instantiating *Interactions*, Mapping Rule 8.3 only introduces *Actors* and doe not make any assumption about the *ACLMessages* exchanged by the involved *Actors*. However, as SoaML does not use a similar concept to *ACLMessage*, only simple *Messages* are produced by Mapping Rule 8.9.

Mapping Rule 8.4: SoaML:ServiceInterface → Collaboration

- **name**: name of the *ServiceInterface*
- **interactionInstance**: collection of UML Activity diagrams specifying how *UML Interfaces* interact with each other (cf. Mapping Rule 8.8)
- **binding**: collection of *UML Interfaces* either realized or used by the *ServiceInterface*
- **actorBinding**: collection of *UML ActivityPartitions* describing the *ServiceInterface ownedBehaviors*

A *CollaborationUse* in SoaML defines how to use a *ServiceContract* in terms of role bindings. Hence, it defines which roles of a *ServicesArchitecture* are bound to which roles of the particular *ServiceContracts*. In Section 4.9.4, a similar concept has been introduced for PIM4AGENTS, namely the concept of an *ActorBinding* defining which *DomainRole* is bound to which *Actor* of an *Interaction*. As the *ActorBindings* are contained by *Collaborations*, we map the *CollaborationUse* as follows:

[11] UML Behaviors can be described in four different ways: Activity UML Diagram, Use Case UML Diagram, Interaction UML diagram, and State Machine UML diagram

Mapping Rule 8.5: SoaML:CollaborationUse → Collaboration

- **name**: name of the CollaborationUse
- **interactionInstance**: collection of UML Activity diagrams of the *CollaborationUses's* type (i.e. *ServicesArchitecture* or *ServiceContract*) expressed through the *ownedBehavior* association (cf. Mapping Rule 8.8)
- **binding**: collection of roles assignments expressed through the *roleBinding* associations used to instantiate the particular *DomainRoleBindigs*
- **actorBinding**: for each *roleBinding* association one unique *ActorBiding* is instantiated

The semantics of *CollaborationUse* (SoaML) and *Collaboration* (PIM4AGENTS) nicely correspond to each other, as both are used to describe how a grouping of entities (*ServiceContract* in the case of SoaML and *Organization* of PIM4AGENTS) are used for a specific purpose. Therefore, Mapping Rule 8.5 defines how the bindings are mapped from a *CollaborationUse* to a *Collaboration*. Beside the *DomainRoleBindings* and *ActorBindings*, moreover, a link between these generated *Bindings* is introduced to specify that the *AgentInstance* performing the *DomainRole* is playing the particular *Actor* in the *Interaction* defined by the *CollaborationUse's Collaboration* (see Mapping Rule 8.3).

Mapping Rule 8.6: SoaML:Participant, SoaML:Agent → AgentInstance

- **name**: name of the *Participant*
- **agentType**: *ParticipantArchitecture* realized by the *Participant* or *Agent* (cf. Mapping Rule 8.2)
- **memberOf**: any *Participant* or *Agent* instantiating any parent *ParticipantArchitecture*
- **members**: any *Participant* or *Agent* contained by the *Agent's* or *Participant's ParticipantArchitecture*

For modeling any kind of service composition (i.e. choreography or orchestration), UML Activity Diagrams are used in SoaML as they allow to model a process from the perspective of a single entity, but also offer the concept of a partition to describe how several entities interact with each other. As detailed by Mapping Rule 8.7, for mapping UML Activity Diagrams, we instantiate a number of *Plans* in PIM4AGENTS. The actual number of them depends on whether the activity diagram is partitioned through the *UML ActivityPartition* concept (i.e. choreography) or not (i.e. orchestration). In case of the former, for each partition, one *Plan* is generated, which defines the local view of each entity of the *Protocol*. In case of the latter, only a single *Plan* is generated describing how services and agents are orchestrated.

Mapping Rule 8.7: UML:Activity → Plan

- **name**: name of the *UML Activity*
- **flows**: collection of *UML ActivityEdge* of type *UML ControlFlow*
- **steps**: collection of *UML ActivityNode* (cf. Table 8.1 for details on the particular mappings)
- **preCondition**: *precondition* attribute of a *UML Behavior*
- **postCondition**: *postcondition* attribute of a *UML Behavior*
- **informationFlow**: collection of *UML ActivityEdge* of type *UML ObjectFlow*
- **localKnowledge**: *ownedParameter* attributes of the *UML Activity*

Mapping Rule 8.7 specifies which kind of information from UML Activity Diagrams is extracted to fill the body of a *Plan*. Table 8.1 roughly illustrates how the different types of *UML ActivityNodes* are grounded into *Activities* of the PIM4AGENTS behavior view.

In general, a *UML Interface* defines a collection of operations and/or attributes that ideally define a set of processes. In order to represent this in an adequate manner in PIM4AGENTS, the concept of a *Capability* depicts the perfect match, as both, operations as well as attributes can be included into the *Capability's Plans*. The concrete mapping rule is defined as follows:

Mapping Rule 8.8: UML:Interface → Capability

- **name**: name of the *UML Interface*
- **behavior**: for each *UML Operation* part of a *UML Interface* one *Plan* is defined that implements the *UML Operation* and makes use of the *UML Attributes* part of the *UML Interface*

Beside the *Capabilities* instantiated by Mapping Rule 8.8, furthermore, a *DomainRole* for each *UML Interface* is established. Hence, any *Organization* produced by Mapping Rule 8.2, may (i) perform *DomainRoles* if its corresponding *ParticipantArchitecture* offers services through *ServicePoints* and (ii) requires *DomainRoles* if its corresponding *ParticipantArchitecture* request services through *RequestPoints*.

As depicted in Table 8.1, any message exchanged between agents and organizations in PIM4AGENTS are derived from *UML ControlFlows* modeled across partitions of *UML Activity Diagrams*. This is expressed by Mapping Rule 8.9.

Mapping Rule 8.9: UML:ControlFlow → Message

- **name**: name of the *UML ControlFlow's* source *UML Action*
- **content**: type of the *UML ObjectFlow's* source pin (by applying Mapping Rule 8.10 or Mapping Rule 8.11)

In SoaML, *UML Class Diagrams* are used to represent any kind of information accessed by the entities in the SOA. *UML Classes* and their attributes serve as input parameters to invoke services.

Process mappings		
Source	*Target*	*Explanations*
UML Activity Diagram (Partition)	Plan	in case of partitions, the particular information from each partition is used as input for Mapping Rule 8.7
UML ControlFlow	*ControlFlow*	one-to-one mapping, *UML ControlFlows* across partitions are used to identify the exchange of messages
UML ObjectFlow	*InformationFlow*	one-to-one mapping
UML ForkNode, UML JoinNode	*Parallel*	any *UML Activity* between the *ForkNode* and *JoinNode* is transformed and integrated into the *Parallel* activity of PIM4AGENTS
—	*ParallelLoop*	no direct support on SoaML/UML; *Send/Receive* activities are integrated into a *ParallelLoop*
—	*Loop*	no direct support on SoaML/UML; need to be defined though *UML ControlFlows*; model transformation detects cycles and automatically instantiates a *Loop* activity of PIM4AGENTS
UML DecisionNode, UML MergeNode	Decision	any *UML Activity* between the *DecisionNode* and *MergeNode* is transformed and integrated into the *Decision* activity of PIM4AGENTS
—	*Sequence*	*UML Activities* connected through *UML ControlFlows* are transformed and integrated into the *Sequence* activity of PIM4AGENTS
UML Activity	*Send*	any *UML Activity* that owns an outgoing *ControlFlow* crossing a *UML ActivityPartition*
UML Activity	*Receive*	any *UML Activity* that owns an ingoing *ControlFlow* crossing a *UML ActivityPartition*
UML *InitialNode*	*Begin*	one-to-one mapping
UML *FinalNode*	*End*	one-to-one mapping
UML OpaqueAction[13]	*InternalTask*	one-to-one mapping

Tab. 8.1: Mapping between *UML Activity Diagrams* of SoaML and *Plans* in PIM4AGENTS.

To represent this in an adequate manner in PIM4AGENTS, the information contained by a *UML Class* is utilized to generate *Objects* containing the necessary information used by *Agents* (i.e. *Organization*) or *Roles*.

Mapping Rules 8.11 and 8.10 define how the information used by *ParticipantArchitectures* is transformed to *Knowledge* the corresponding *Organization* in PIM4AGENTS has access to.

Mapping Rule 8.10: SoaML:MessageType → Object

- **name**: name of the *MessageType*

- **attribute**: collection of *UML Attributes* of the *UML Class*

A *MessageType* in SoaML represents a special mechanism to define documents sent between the parties of, for instance, a choreography. In PIM4AGENTS, we do not distinguish between different kinds of attachment, i.e. any kind of *Knowledge* can be defined as the *Message's content*. Apart from a *MessageType*, any general *UML Class* can be attached to a *Message* as its content.

Mapping Rule 8.11: UML:Class → Object

- **name**: name of the *UML Class*
- **attribute**: collection of *UML Attributes* of the particular *UML Class*

The mapping rules presented in this section define a generic transformation path from SoaML to DSML4MAS. In the remainder of this section, we demonstrate for the CMS SoaML model depicted in Section 8.3.2 how these mappings rules work in practice to generate a corresponding PIM4AGENTS model, which can be further used as input for the vertical transformations to JACK or JADE.

8.4.2 Illustrative Examples: Conference Management System in DSML4MAS

Section 8.3.2 focuses on the conference management system design using SoaML. Now, we focus on the generated PIM4AGENTS CMS model. Therefore, we illustrate the main diagrams of PIM4AGENTS and how they evolve from the model transformation and the underlying SoaML design.

8.4.2.1 Multiagent System Diagram

The generated MAS diagram is depicted in Fig. 8.13. It includes an abstract view on the generated MAS. Apart from the involved organizations (i.e. *ConferenceOrganization*, *SeniorResearcher*, and *Researcher*) and domain roles (i.e. *Author*, *PCMember*, and *PCChair*) that are detailed in the organization diagram (cf. Section 8.4.2.2), it further includes an environment called *CMS* that includes all necessary resources created by either Mapping Rule 8.10 or Mapping Rule 8.11.

8.4.2.2 Organization Diagram

The generated organization diagram is depicted by Fig. 8.14. This content of this diagram bases on the results produced by (i) Mapping Rule 8.1 that generates the *ConferenceOrganization* organization from the *ConferenceOrganization* services architecture and (ii) Mapping Rule 8.2 producing the *SeniorResearcher* and *Researcher* organization. The organizations have access to the capabilities *Author*, *PCMember*, and *PCChair* that were instantiated by Mapping Rule 8.8. To define how the organization coordinate its members, the domain roles *Author*, *PCMember*, and *PCChair* were introduced that are either required or performed by the organizations. The interaction protocols CallForReview and CallForPaper are utilized by the ConferenceOrganization to coordinate its

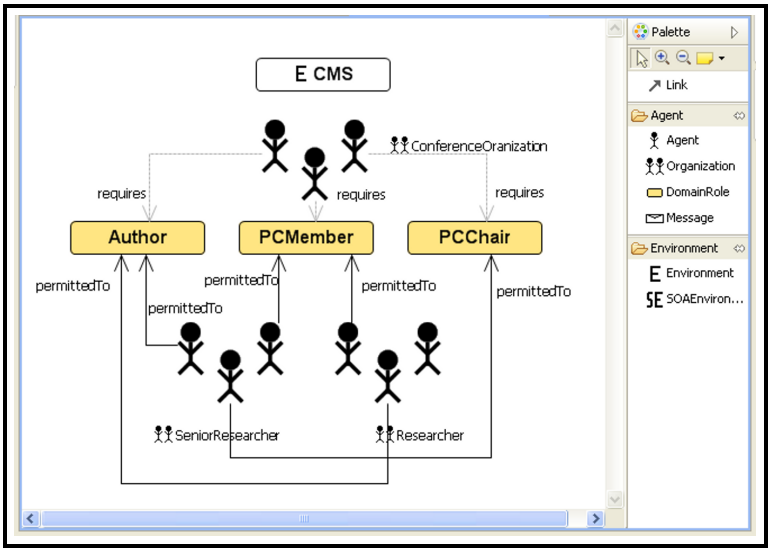


Fig. 8.13: The MAS diagram of the generated PIM4AGENTS CMS model.

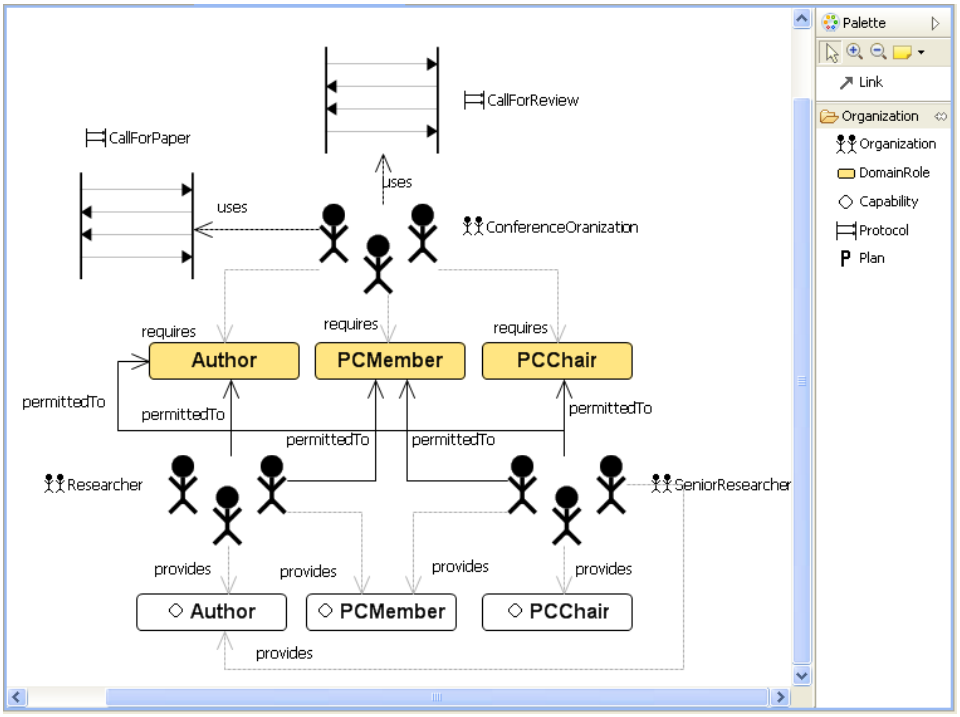


Fig. 8.14: The organization diagram of the generated PIM4AGENTS model.

associated members. These protocols were instantiated by applying Mapping Rule 8.3 on the service contracts depicted in Fig. 8.8.

8.4.2.3 Collaboration Diagram

The generated collaboration diagram is depicted by Fig. 8.15. The digram includes the cooperations of the *ConferenceOrganization* organization. Both cooperations, i.e. *callForPaper* and *callForReview*, were instantiated by Mapping Rule 8.5. For each of the domain roles a collaboration uses, a unique domain role binding is instantiated. Each of them is linked to the actors of the collaboration's interaction through the actor binding.

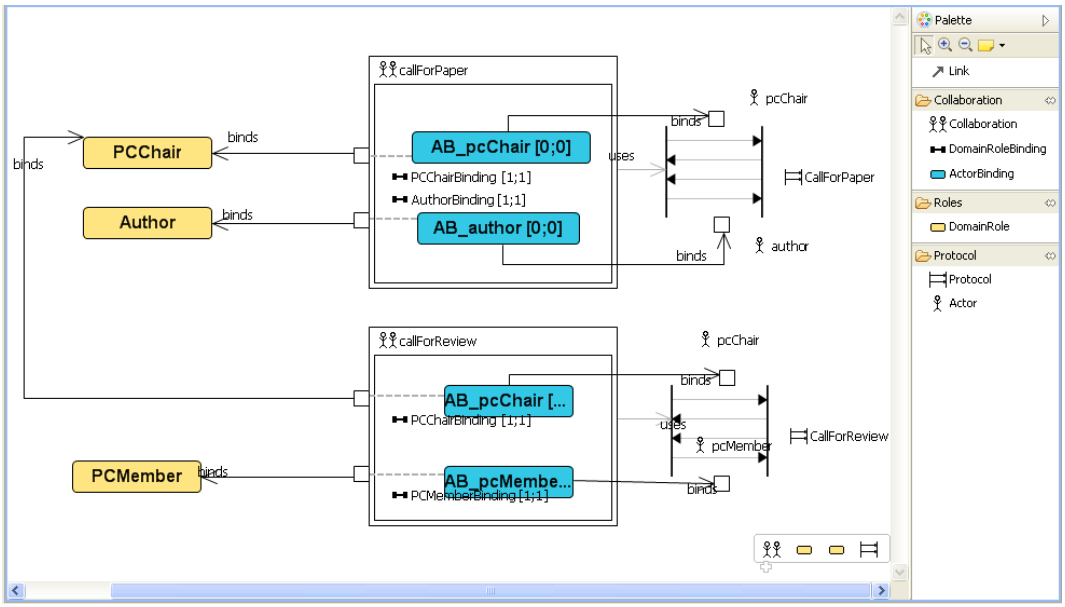


Fig. 8.15: The collaboration diagram of the generated PIM4AGENTS model.

8.4.2.4 Deployment Diagram

The generated deployment diagram is depicted by Fig. 8.16. The deployment is based on Mapping Rule 8.6 that transforms the agents *Christian* (see Fig. 8.10) and *Klaus* (see Fig. 8.9) from the SoaML CMS specification to the agent instances of PIM4AGENTS. The memberships of these agent instances are deduced from the collaborations and service contracts in which the particular agents in SoaML participate.

8.4.2.5 Plan Diagram

Fig. 8.17 depicts the behavior diagram of the *PCChair's* plan to interact with the *Authors*. It is based on the *pcChair's* part of the choreography of the *CallForPaper* service contract depicted in Fig. 8.11. It starts with sending the *SendCFP* message to any agent instance playing the *author*

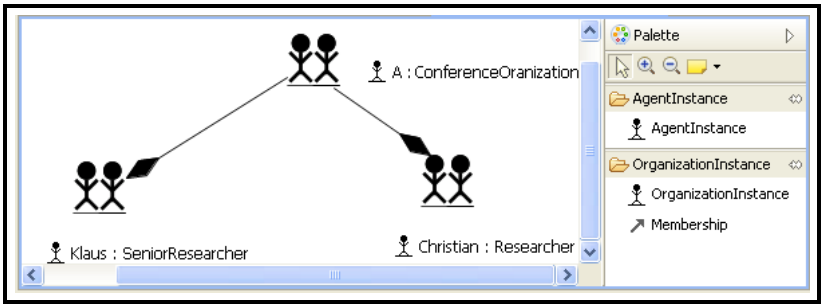


Fig. 8.16: The deployment diagram of the generated PIM4AGENTS model.

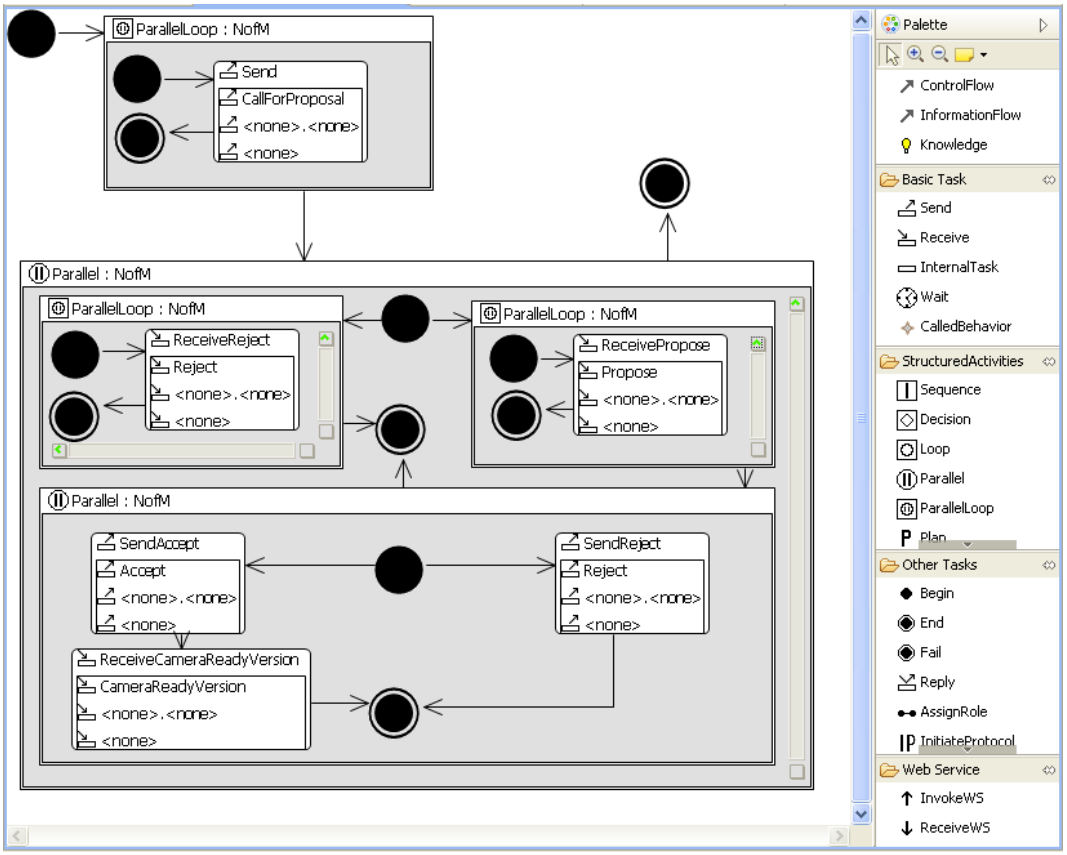


Fig. 8.17: The behavior diagram of the generated PIM4AGENTS model.

actor. When receiving the answers by the *ReceiveReject* and *ReceivePropose* activities, the *PCChair* decides based upon the reviews of the *PCMembers*, which papers are accepted and rejected. Accordingly, the *Author* submitted a paper either gets a *SendAccept* or *SendReject* message sent

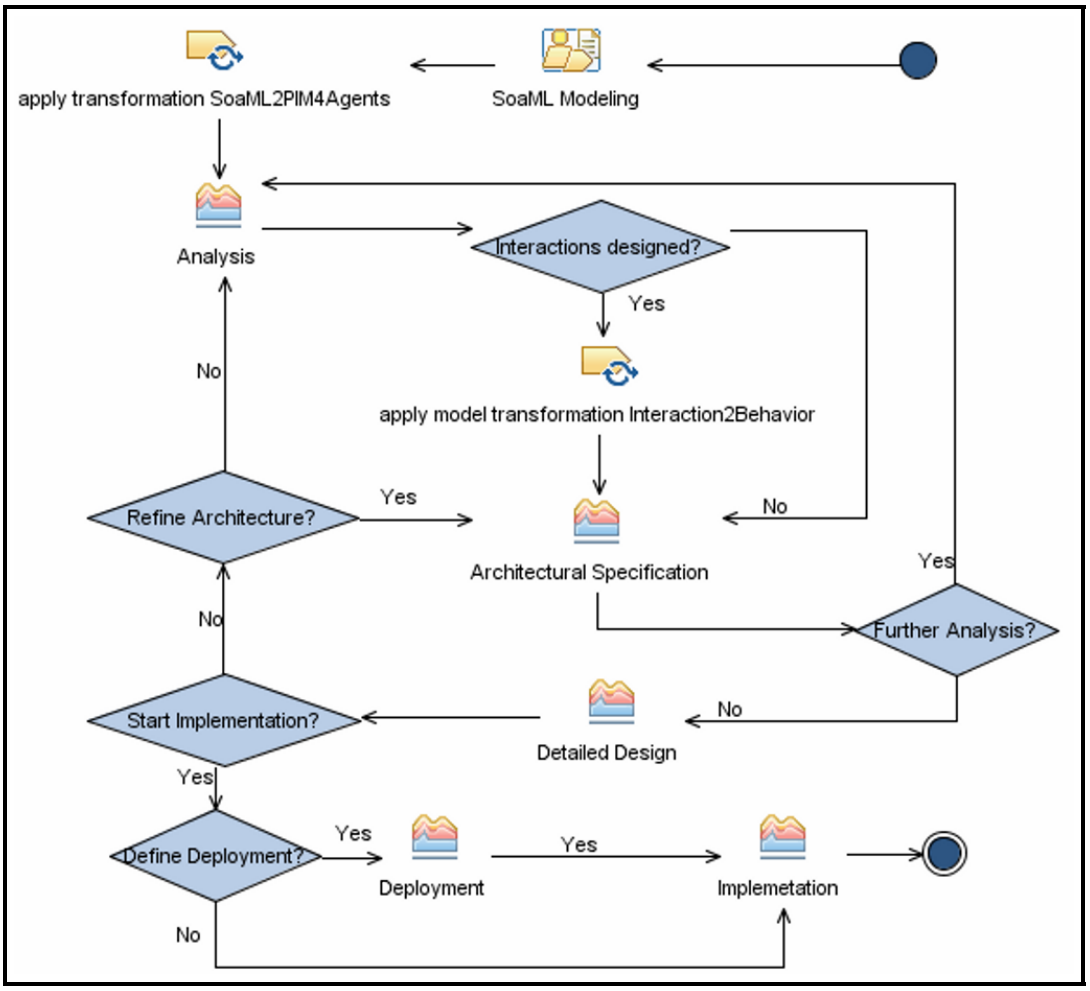


Fig. 8.18: The SOA-related process of DSML4MAS.

by the *SendAccept* and *SendReject* activities, respectively. Afterward, the *PCChair* waits for the *SendCameraReadyVersion* message received by the *ReceiveCameraReadyVersion* activity.

8.5 PIM Service-oriented Architecture Modeling Process

When embedding MASs in a service-oriented setting, the methodology process proposed in Section 5.4.2.2 needs to be slightly adapted as the model-driven transformation between SoaML and DSML4MAS now guides to a large extend the agent-based design. The adapted SOA-related process model is depicted in Fig. 8.18. Comparing this with the DSML4MAS methodology previously presented in Fig. 5.18, the main change is the SoaML modeling phase on the PIM level and the mapping to DSML4MASThe DSML4MAS process activities, as well as, the model transformation between DSML4MAS and JACK/JADE on the PSM level remain untouched.

Analysis phase Instead of specifying the system requirements using DSML4MAS, in the SOA-related setting, the analysis already starts using SoaML or even on a more computational-independent level using business languages like BPMN. If SoaML is used to analyze the scenario, normally, the system designer disposes use case diagrams and sequence diagrams to define the interaction across the actors specified in the use case diagram. These are then used inside service contracts of either services architectures or participants architectures. UML diagrams serve as design method to define the information model used by the service requesters and providers.

Architectural specification phase Similar to the analysis phase, the architectural specification phase is mainly influenced by SoaML and its collaboration and contract modelings. The architecture is further refined in terms of service architectures and participant architectures utilizing the design made on the analysis phase. In addition, the internal behaviors of the participant architectures are declared in this phase. The generated SoaML design is then translated into DSML4MAS concepts that might be further refined in terms of the agent-based computing paradigm.

Detailed design phase After refining the service-based design in terms of agent concepts, the generated plans are completed and the bindings between actors and domain roles are created. Parts of the detailed design phase like the behaviors of the participants are either done on the SoaML level or completed by the model-to-model transformation. Even if most of the design is already in place in this phase, the developing process can be further detailed by refining the design using DSML4MAS constructs and design mechanisms of the analysis phase.

Implementation phase As a last step, the generated agent instances are further refined in the implementation phase in terms of adding their memberships. Afterwards, the model transformations to JACK and JADE are applied. Potentially, the code is further refined to execute the design made.

This extended DSML4MAS methodology allows to automatically close the gap between SOAs and MASs. At this, the SOA design is used as an abstract design that is as part of the original DSML4MAS methodology process further refined in terms of agent-based concepts. As debated in the forthcoming section, this agent-based SOA has several benefits compared to traditional Web service-based SOAs.

8.6 DSML4MAS as Web Service Execution Engine

MASs are normally not considered as the standard execution technique for SOAs. Standard Web service description formats like WSDL and orchestration engines like BPEL might be more appropriate. However, the overall SHAPE model transformation architecture, as proposed in Section 8.2.4.2, proofs that MASs and in particular DSML4MAS is an interesting option when executing abstract service description as agent systems in general offer valuable features that are worth to investigate in the SOA context.

For this reason, we presented in this chapter a feasible integration between SOAs and MASs. Therefore, we apply a model-driven approach in order to automatically transfer a service description into an abstract MAS that can be further transformed to generate executable code

implementing the abstract service architecture. In particular, we provided a link between SoaML—the new standard for SOAs by the OMG—and PIM4AGENTS by grounding the concepts of SoaML into PIM4AGENTS. This generic model mapping can be realized as PIM4AGENTS is more expressive with respect to defining interactions and behavior necessary to represent orchestration and choreography in an adequate manner. Based on the generated PIM4AGENTS models, the vertical transformations to the agent implementation platforms can be applied, to execute the business description made with SoaML in an intelligent manner using agent systems. From a research transfer point of view, the following lessons could be learned:

- Evidently, MDD is a necessary ingredient for SOAs, in particular, a model-driven, agent-based approach offers additional flexibility and advantages when agents are tightly integrated into a service-oriented framework.
- The SoaML profile supports the range of modeling requirements for SOAs, including the specification of systems of services, the specification of individual service interfaces, and the specification of service implementations. SoaML allows designing services top down through the concepts of *ServicesArchitectures* and *ServiceContracts* as well as bottom up by designing the service architecture by the concepts of *ParticipantArchitectures* and *ServiceCapabilities*.
- The agent paradigm is not new to SOAs, however, the presented approach establishes one of the first proposals combining SOAs and the agent world through a generic model transformation. This allows to automatically transform the business requirements during IT design and development to a MAS that captures the business requirements, but additionally supports the execution in an intelligent manner.
- Evidently, a model based approach is a step in the right direction as design-time tasks are separated from run-time tasks, which allows performing them graphically. Moreover, it is easier to react to changes of the different interacting partners as only the models have to be adapted but not the run-time environment. The service-oriented design built upon SoaML can be detailed and refined using more agent-oriented concepts provided by DSML4MAS and finally enhanced in terms of platform-specific implementation details.
- The PIM4AGENTS metamodel is expressive enough to support a generic mapping between SoaML and necessary parts of UML—in particular UML Activity diagrams—on the one hand and PIM4AGENTS on the other hand. Most notably, the service choreography and orchestration described by SoaML can nicely be mapped to PIM4AGENTS, which allows representing service architectures from an external and internal perspective. This is not supported by the most known Web service orchestration engine BPEL4WS.
- In its current version, SoaML offers only a kind of semantics expressed in natural language. Through the model transformation to DSML4MAS, the modeling constructs are grounded into analog concepts of PIM4AGENTSwhose semantics are clearly defined through the formal Object-Z specification.

The presented model-driven framework to specify agent-oriented applications represents a necessary step in order to build interoperable agent systems on the PIM level. This is an important step towards bringing MASs into industry as any service description built upon SoaML can be automatically transformed to make use of the advantages the supported agent platforms offer. Moreover, especially for the composition of services, agent systems further bring in the following characteristics.

- Agents improve flexibility and robustness as they are reactive, flexible, social and interact with the environment. Consequently, they would adopt to changing contexts and environments and either choose alternative plans or delegate the tasks to other agents.

- Agents add proavtiveness as they are communicative and social. They typically respond to both, messages from other agents and changes in the environment. This results in proactive behavior, whereas Web services are typically just reactive.
- Agents add goal orientation. Web services for instance are task-oriented as they exist to provide access to some resources. Agents, in contrast, are goal-oriented, which allow them to execute tasks in a more flexible manner. This is of particular importance in the case of failure recovery. This means that if a plan to achieve an agent's goal fails, it might apply other available plans to achieve this goal.

8.7 Bottom Line

In this chapter, the DSML4MAS approach to the integration of SOAs and MASs has been discussed. For this purpose, the basic concepts of SOAs and services and their core characteristics and benefits were discussed. In the same manner, we also debated the differences between the MAS and SOA paradigms. Summarizing this debate, we can conclude that agents are stateful, whereas services are in general stateless. To establish the interoperability between SOAs and MASs, we utilize the recently standardized Service-oriented Architecture Modeling Language (SoaML) and define a model transformation between SoaML and PIM4AGENTS. This model transformation enables the generic mapping between SOA-based concepts into the agent-based vocabulary given by the abstract syntax of DSML4MAS.

Even if services and agents are to some degree different, this model transformation is feasible due to the fact that SOAs and MASs share the same benefits and characteristics like flexibility, loose coupling, etc. and use similar views to describe the design (e.g. the concept of an agent is part of both SoaML and PIM4AGENTS). On the one hand, the presented MDD approach to reduce the interoperability barriers between MASs and standard business languages like SOA is an important step toward making MASs more attractive for industrial usage as SOAs are nowadays the preferred approach to design distributed software systems in real-world scenarios. On the other hand, agent-based computing offers several features and characteristics that enable a more intelligent way of executing and composing Web services, i.e. improving flexibility and robustness, adding proactiveness and goal-orientation. Section 9.1 illustrates how to apply SoaML and the model transformations to DSML4MAS in an industrial use case to model the supply chain of the Saarstahl AG.

Part IV

Use Case and Evaluation

9. DSML4MAS in Industrial Use Cases

In this chapter, we demonstrate how to apply the contributions of this dissertation in two industrial use case scenarios. For this purpose, we firstly investigate a real-world scenarios from the steel industry at the Saarstahl AG and secondly describe how the DSML4MAS approach can be utilized in the oil domain of Statoil. Due to its nature, the first use case emphasizes on the Saarstahl's service architecture expressed by SoaML and DSML4MAS, the second use case concentrates on DSML4MAS itself and on the code generation for JACK.

Structure of this Chapter In Section 9.1, the supply chain of the Saarstahl AG is introduced by depicting (i) the SOA-based supply chain and (ii) the automatically produced agent-based supply chain of the Saarstahl AG. Section 9.2 then focuses on the Mongstad terminal of Statoil and illustrates how to model its requirements using DSML4MAS. Finally, Section 9.3 concludes this chapter.

9.1 Model-Driven Integration of the Saarstahl Supply Chain

Saarstahl AG, with its locations in Völklingen, Burbach and Neunkirchen, is a German steel manufacturing company with global presence on the steel production market. In particular, Saarstahl AG specializes in the production of wire rod, steel bars, and semi-finished products of various grades as well as constructional steel and broad flanged beams.

Saarstahl AG has grown historically, meaning that the existing IT infrastructure has grown over the last decades, and currently consists of several loosely integrated systems. This consequently means that the main challenge when describing the complete supply chain of Saarstahl is to enable the interaction between new components like the agent-based planning system of the steelworks MasDISPO (Jacobi et al.; 2007) and older solutions like the semi-finished product management system implemented in Cobol.

In order to improve the intra-organizational interoperability, Saarstahl focuses on integrating the existing systems into a system architecture on organization level, along with the introduction of new technologies such as MASs. To solve these interoperability issues, Saarstahl sees the combination of SOAs and MASs as a key success factor.

9.1.1 Supply Chain of the Saarstahl AG

The use case scenario of the Saarstahl AG is a proof of concept for designing the main processes within the Saarstahl's supply chain based on the results of DSML4MAS. Several challenges have to be addressed when it comes to a service-oriented design of the complete supply chain. From the viewpoint of Saarstahl it is fundamental that (i) business requirements that are specified by

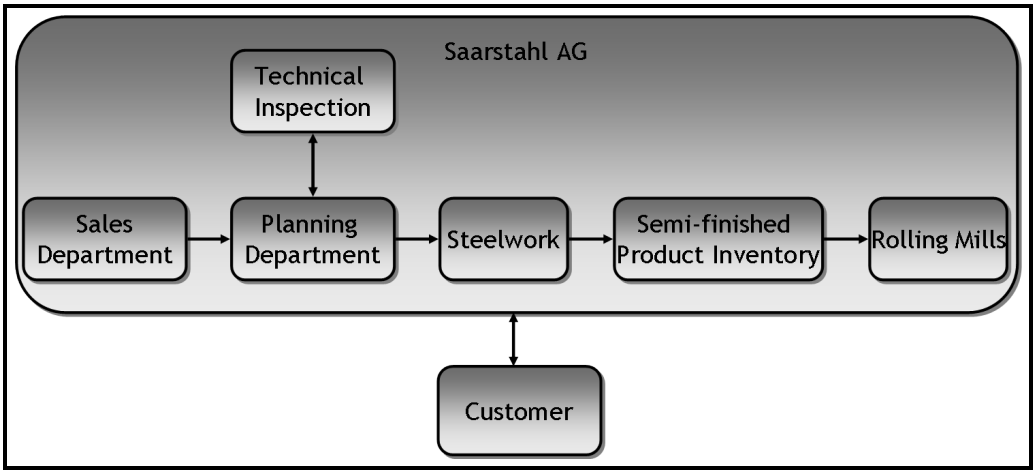


Fig. 9.1: The partial supply chain of the Saarstahl AG.

Saarstahl can easily be translated into a running system and (ii) existing systems holding strategic information (e.g. data bases) can be re-used within the SOA to keep the high product quality they currently hold. Both requirements are naturally supported by the DSML4MAS approach:

- The DSML4MAS methodology provides a full transformation path from the service level to MASs through MDD and thus allows Saarstahl to define executable artifacts in a very abstract manner on the SOA level.
- The DSML4MAS methodology supports the integration of existing legacy systems that are situated at different locations (e.g. Völklingen, Burbach and Neunkirchen) through a combination of Web services and agent-based systems. Using MASs offers various advantages that allow increasing efficiency during run-time.

The central benefit of introducing a service-oriented landscape is a higher degree of integration and interoperability among the separated systems: the legacy systems can be wrapped beyond service interfaces, so that information can be exchanged more easily and with less need for human intervention. Another benefit is that old systems can be replaced by new implementations without interrupting the production process. The need for integrating SOAs and MASs makes the framework presented in this thesis an interesting approach.

The core components of Saarstahl's supply chain are depicted in Fig. 9.1. The steel works in Völklingen is of central relevance to the complete supply chain of Saarstahl as it, on the one hand, receives orders from customers and, on the other hand, is the starting point for producing the requested goods. Furthermore, the efficiency of the supply chain, i.e. the time needed for achieving the requested product, strongly affects the order that can be produced within a certain time slot.

Given a working plan, the planning system schedules the execution of each order along the production chain. It monitors production on a rough (weeks) and detailed (days and hours) level, and executes an online detailed planning and scheduling for the different manufacturing phases. It has to detect problems in the production and handle them in order to return to normal production. The rough working plan for each manufacturing phase is calculated on demand, before final order commitment. Depending on delivery date, order size and vertical integration certain capacities at specified aggregates have to be roughly allocated. The overall objective of Saarstahl's production is

to provide the right amount the intermediated product close to but not later than the time that it was requested. Derived from customer's orders, each order has a fixed finishing date on each processing level. In the rolling mills, the duration of a rolling group, which are heats of kind and equal format that are to be casted in a single and uninterrupted casting session at a specific casting aggregate, lies between several hours up to several days, depending on the stock of orders. A heat may thereby contain several order positions of similar quality, but equal format.

The Saarstahl AG expects that utilizing DSML4MAS helps to ensure the necessary vertical integration of detailed and rough planning concerning one station and the horizontal integration for the exchange among several partners. Rough planning inside a rolling mill influences the detailed planning inside the steelwork and vice versa. Furthermore, the increase of transparency and better support for the necessary data exchange to improve the overall planning is expected using the DSML4MAS approach.

9.1.2 Service-oriented Supply Chain of the Saarstahl AG

This section gives the reader are detailed overview on how to design the Saarstahl's supply chain using SoaML. To bring the relationship between customer orders and production closer together, we focus on the steel work and in particular on the processes within the supply chain (depicted in Fig 9.1). For this purpose, we need to take any production critical aspect into account and model the corresponding internal as well as external processes. Saarstahl expects by the use of SOAs a better information exchange and hence an increased transparency along the supply chain. The supply chain of Saarstahl is again designed with the Modelio tool suite.

The overall SOA architecture of the Saarstahl AG is as follows: All actors except the costumer belong to the Saarstahl system. The costumer is able to purchase products of Saarstahl by filling a purchase order form which is transfered to the sales department. The sales department then registers the order in the Saarstahl system, which produces a production schedule for the order. Scheduling is done by the planning department, the central actor in this scenario. However, before any resource is allocated for the order, the planning department validates its feasibility of production with the support of the technical inspection. The result is then reported to the sales department, which informs the costumer accordingly. If the order is feasible, the planning department activates the processing on the scheduled date. The first step of processing is to search the inventory for available material fitting the order's requirements. Available material is then assigned to the order. If there is not enough material available, the planning department schedules a melting job at the steelworks. For every completed melting job, the steelworks sends data on quality and quantity to the semi-finished product inventory. After material has been produced for the order, the planning department validates the quality requirements and releases the material, if they are not satisfied. This material is then marked as available in the inventory. When the order quantity is completely allocated, the order is transfered to the rolling mills management system. This represents the final step of the investigated use case of Saarstahl.

9.1.2.1 Customer

The costumer interacts with Saarstahl in order to purchase some product. The resulting SoaML model hence contains a SoaML participant representing the *Costumer* entity and a participant representing the Saarstahl AG. Fig. 9.2 depicts the participant architecture representing a generic *Costumer*. The *ParticipantArchitecture* stereotype is used in order to emphasize that this model is rather a specification of a participant, instead of a specific instance. The specific instance is

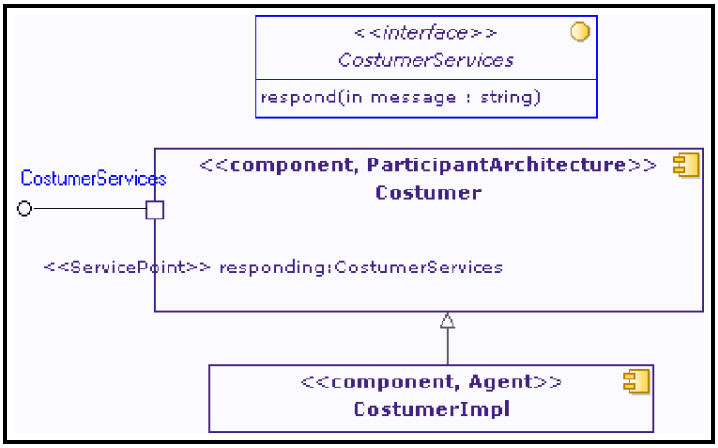


Fig. 9.2: The *Customer* participant architecture

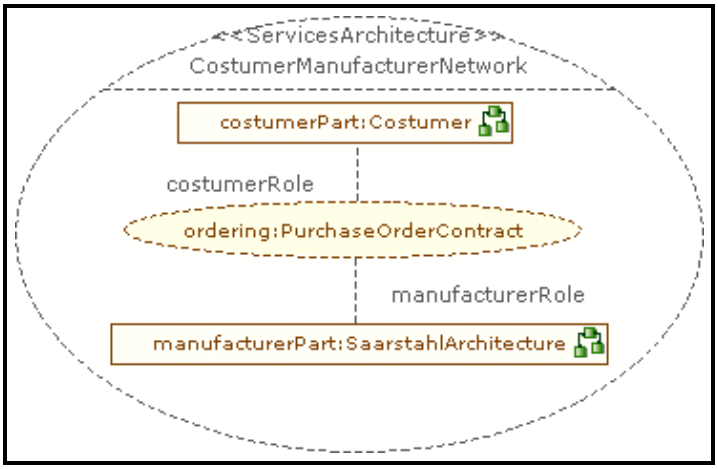


Fig. 9.3: The interaction between *Customer* and *SaarstahlArchitecture* is defined through the *CustomerManufacturerNetwork* service architecture.

represented by the *CostumerImpl* that realizes the abstract costumer specification. The *Costumer* participant architecture provides a single respond service, which is part of the *CostumerServices* interface provided through the responding service point.

The collaboration between costumer and Saarstahl is modeled by the service architecture depicted in Fig. 9.3. The *CustomerManufacturerNetwork* services architecture includes two roles, i.e., *costumerPart* and *manufacturerPart*, which are typed by *Costumer* and *SaarstahlArchitecture*, respectively. Both interaction partners have to agree to the contract specified by the *Purchase-OrderContract* service contract. This service contract names the roles that are part of the contract and specifies how these roles interact with each other through exchanging messages. The concrete order of these message is given by the activity diagram *PurchasingProcess* depicted in Fig. 9.4.

The *PurchasingProcess* consists of two roles, i.e. *costumerRole* and *manufacturerRole* that are bound by the *costumerPart* and the *manufacturerPart*, respectively. The process starts with the action *send_purchase_order* by the *customerRole* requesting a new order from the *manufacturerRole*.

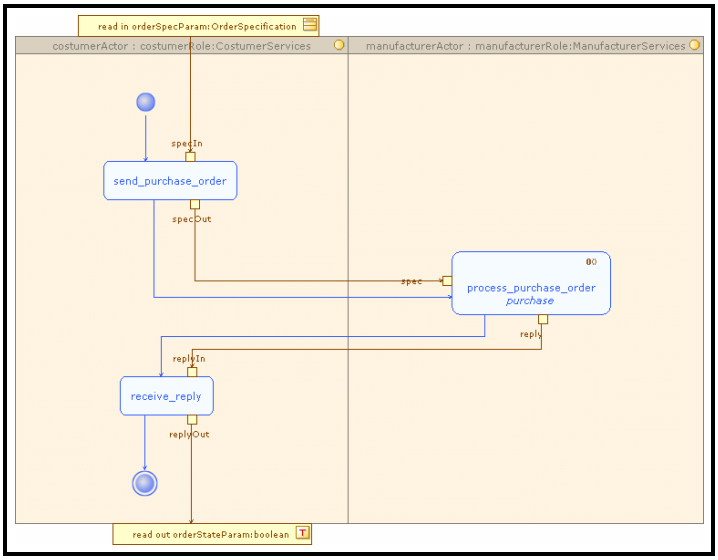


Fig. 9.4: The choreography between the *customerActor* and *manufacturerActor*.

The call operation action *process_purchase_order* of the *manufacturerRole* evaluates the request and sends the result of this evaluation back to the *costumerRole*. The result is then received by this role in the *receive_reply* action.

9.1.2.2 Saarstahl Architecture

As previously mentioned, the *SaarstahlArchitecture* (see Fig. 9.5) includes any actor of the Saarstahl AG supply chain. This means that the *SaarstahlArchitecture* includes the participant architectures *SalesDepartment*, *PlanningDepartment*, *Steelwork*, *Order*, *RollingMill*, *SFProductInventory*, and *TechInspection*. The internal communication between these parties is done through service contracts, e.g. *SchedulingContract*, *MeltingContract*, *SendReportContract*, *ActivateOrderContract*, *ReleasMaterialContract*, *PostMeltingContract*, *CheckFeasibilityContract*, and *AllocateMaterialContract*. To the outside, the *SaarstahlArchitecture* offers the purchasing service through the interface *ManufacturerService*. The concrete implementation is done through the *SaarstahlImpl* agent.

9.1.2.3 Planning Department

The planning system of the steelworks (see Fig. 9.6) is a collection of software components like MasDispo (Jacobi et al.; 2007) with the purpose of long-term and short-term planning and scheduling inside the melting shop in the production lifecycle of an order. The production data of heats is sent to the semi-finished product management system after melting job completion. Furthermore, this system interacts with the order management system to retrieve production rules for orders and to provide production progress data.

The *PlanningDepartment* modeled as participant architecture offers the service *PlanningDepartmentServices* to the outside including the methods *registerOrder*, *requestCommitment*, and *determineLMST*. The concrete implementation of the *PlanningDepartment* participant architecture is done through the *PlanningDepartmentImpl* agent.

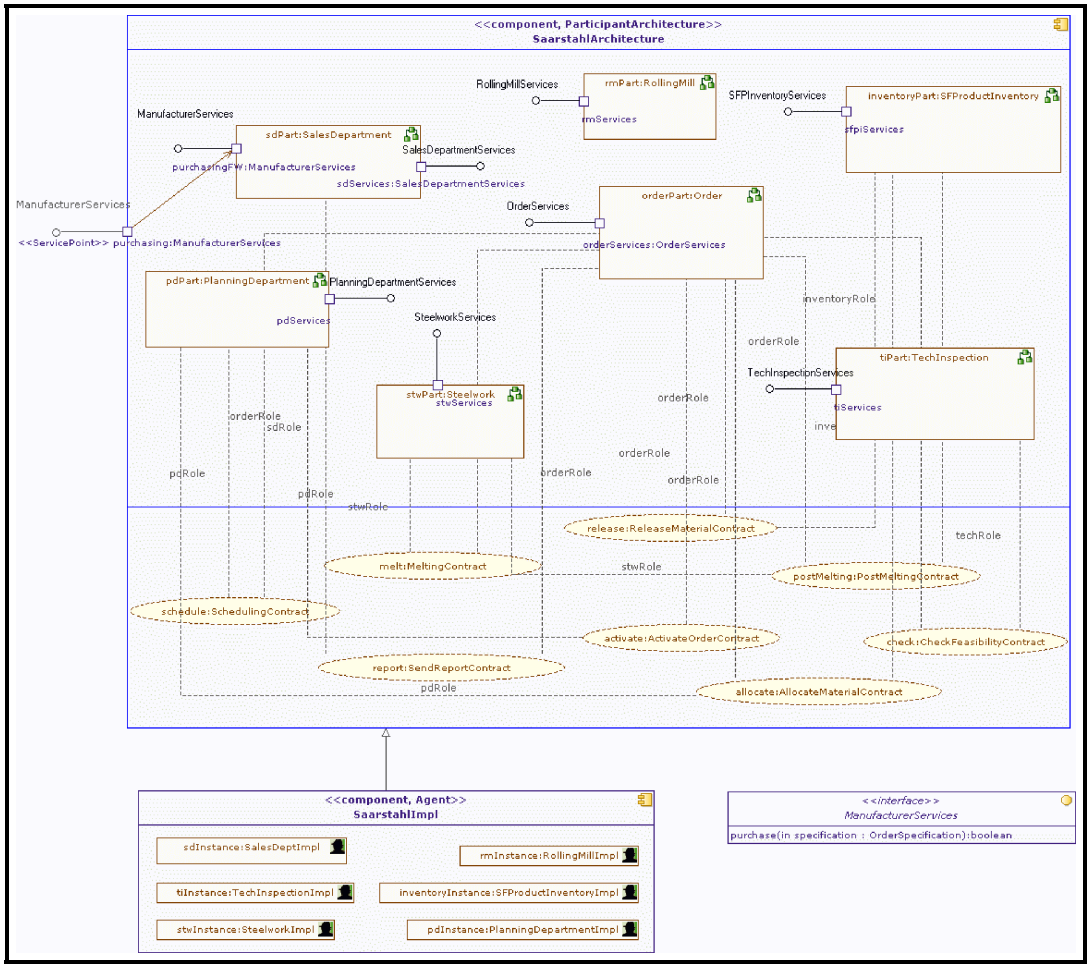


Fig. 9.5: The *SaarstahlArchitecture* participant architecture.

9.1.2.4 Sales Department

The sales department of the Saarstahl AG triggers the purchase process of customers. Therefore, it registers the order in the Saarstahl system, produces a production schedule for the order, and evaluates if the order can be produced by Saarstahl. The *SalesDepartment* participant architecture is depicted in Fig. 9.7. It offers the *purchasing* and *sdService* services through the interfaces *ManufacturerServices* and *SalesDepartmentServices*, respectively. The *SalesDeptImpl* agent implements the abstract *SalesDepartment*.

9.1.2.5 Semi-finished Product Inventory

The semi-finished product component manages the inventory for semi-finished products. When a new order is ready for production, this component is used for querying the inventory for available semi-finished products that fit to the order's requirements. The corresponding database contains the data of all semi-finished products and the assignments between physical material and concrete

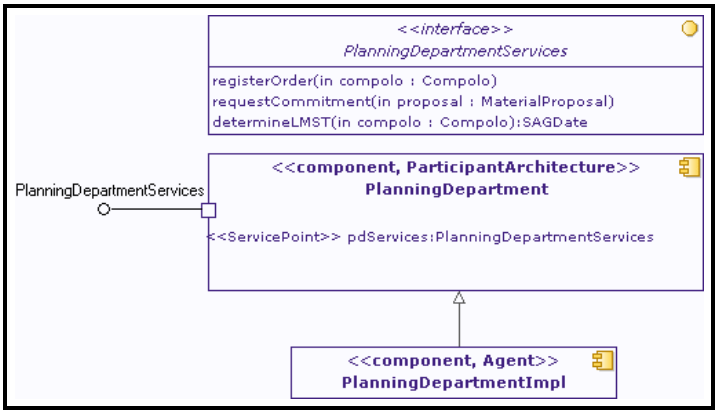


Fig. 9.6: The *PlanningDepartment* participant architecture.

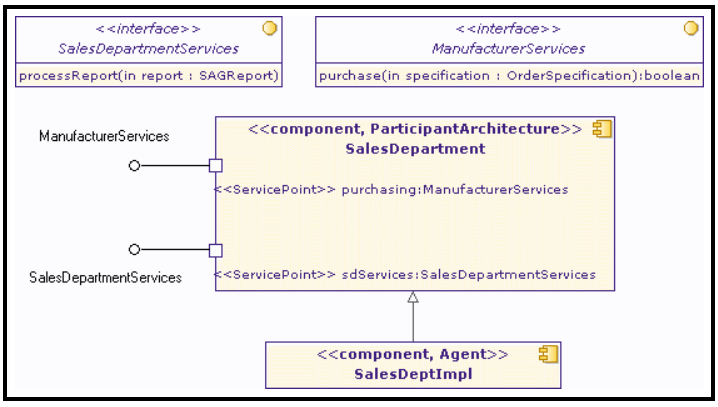


Fig. 9.7: The *SalesDepartment* participant architecture.

orders that have not yet been allocated. If an order is completely assigned, its data is removed and is transmitted to (i) the rolling mills management system for further processing and (ii) the order management system. In the Saarstahl use case, the semi-finished product component is realized as SoaML participant architecture called *SFProductInventory* (cf. Fig. 9.8). To query the inventory for available semi-finished products, this participant provides the *SFPInventoryServices* service to the outside. The *SFProductInventory* is implemented through the *SFProductInventoryImpl* agent.

9.1.2.6 Rolling Mills

The rolling management system is used to plan and schedule the rolling campaigns. Rolling is the first production step to be determined, because the rolling date is used to compute deadlines for all other production steps. This system provides information to the order management system and activates orders in the order queue when necessary. Rolling campaign data is stored in the rolling plan database. Fig. 9.9 depicts the *RollingMill* realized as participant architecture in SoaML. The *RollingMill* participant architecture provides the service *RollingMillServices* to the outside and is implemented by the *RollingMillImpl* agent.

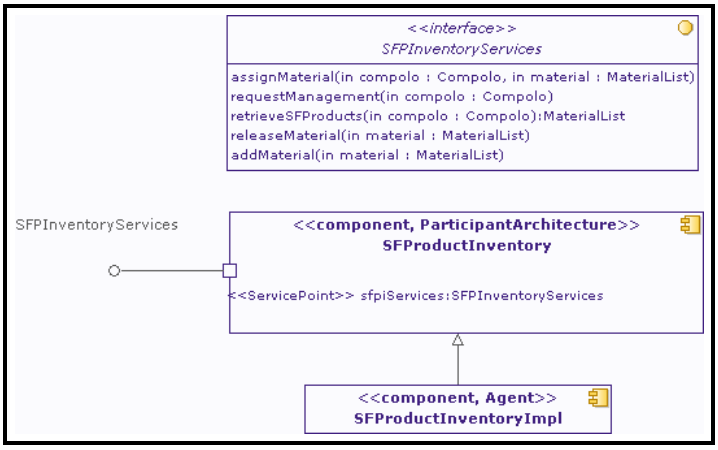


Fig. 9.8: The *SFPInventory* participant architecture providing the *SFPInventoryServices* service.

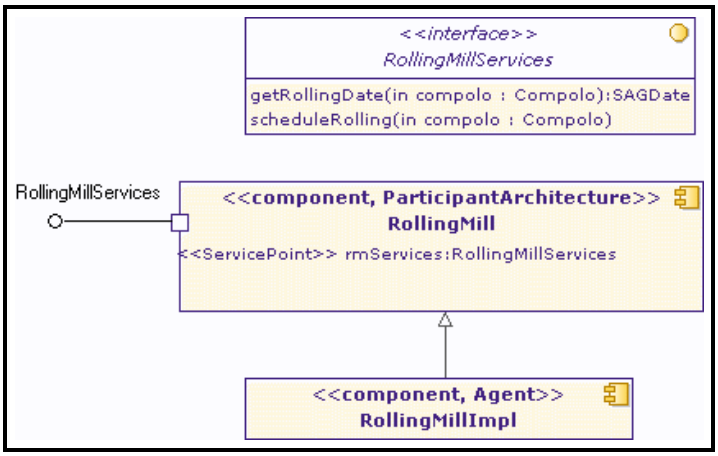


Fig. 9.9: The *RollingMill* participant architecture.

9.1.2.7 Order

Order transparency and tracking within the complete lifecycle of an order from order entry until invoicing is achieved with the help of the order management system, which uses the order database to access the complete data of an order. Fig. 9.10 depicts the *Order* entity realized as participant architecture in SoaML. The *Order* includes the attributes *compolo* of type *Compolo*, *rollingDate* of type *SAGDate*, *eventQueue* of type *ProductionEventQueue*, and *Lmst* of type *SAGDate*. These different kinds of information are modeled as UML classes. Additionally, the *Order* offers the *orderServices* services through the *OrderServices* interface.

9.1.3 DSML4MAS-Based Supply Chain of the Saarstahl AG

The first abstract service description of the Saarstahl supply chain has been done using SoaML. This description includes the involved entities, the cooperation structures, as well as, the services

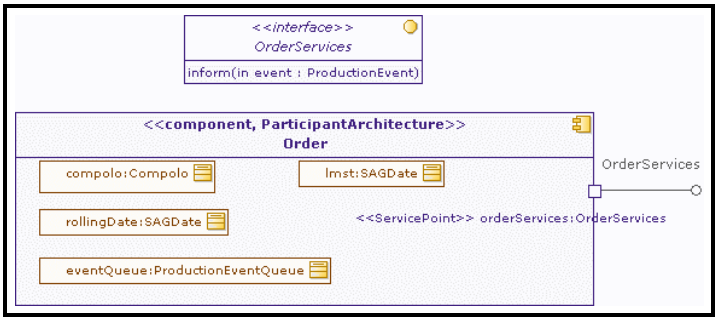


Fig. 9.10: The *Order* participant architecture.

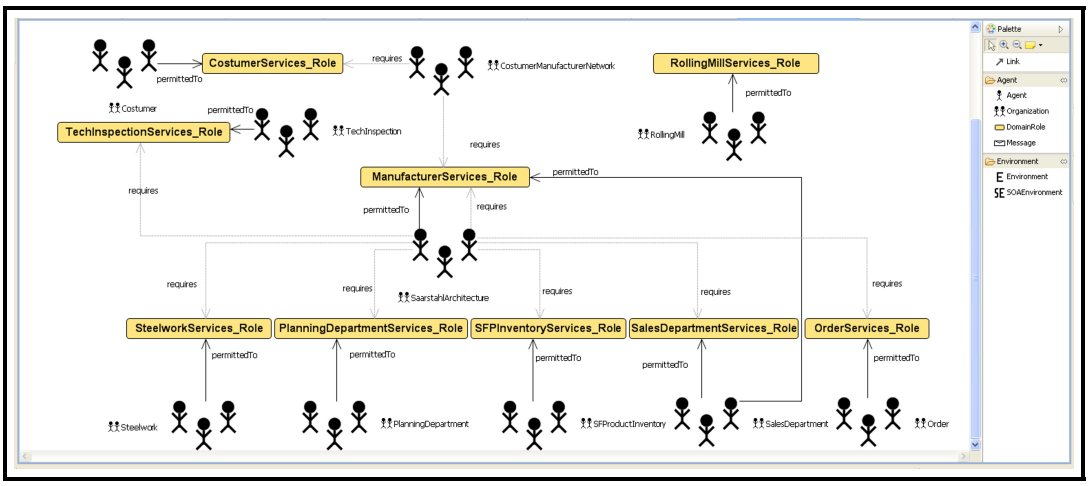


Fig. 9.11: The generated PIM4AGENTS MAS diagram (part 1).

that are requested and provided. Based on the SOA description, the model transformations of the DSML4MAS methodology are applied to generate an agent-based design of the supply chain.

In this next step of the DSML4MAS methodology, based on the developed SoaML model, the model transformation between SoaML and DSML4MAS is utilized. This transformation automatically generates a number of diagrams that are discussed in the following.

9.1.3.1 Multiagent Diagram

The generated MAS diagram is depicted in Fig. 9.11. Mapping Rule 8.2 generates the organizations *Customer*, *SaarstahlArchitecture*, *Steelwork*, *PlanningDepartment*, *SFProductInventory*, *SalesDepartment*, *Order*, and *RollingMill*. The details of these generated organizations are further refined in the organization diagram in Section 9.1.3.2. Mapping Rule 8.1 generates the *CustomerManufacturerNetwork* that requires the domain roles *CostumerServices_Role* and *ManufacturerServices_Role*. Further domain roles are instantiated, e.g. *RollingMillServices_Role*, *SteelworkService_Role*, *PlanningDepartmentService_Role*, *SPFInventoryServices_Role*, *SalesDepartmentServices_Role*, and *OrderServices_Role*.

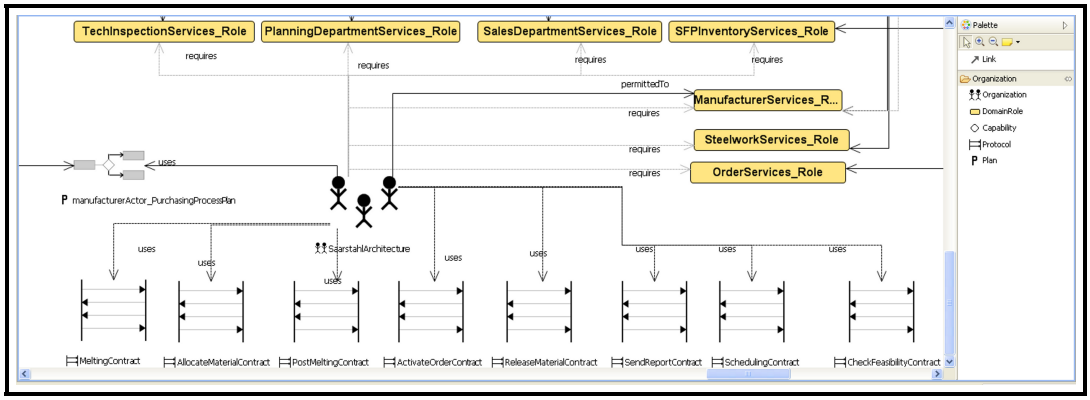


Fig. 9.12: The generated PIM4AGENTS organization diagram (partly).

9.1.3.2 Organization Diagram

The partial generated organization diagram is depicted in Fig. 9.12. For space reasons, we focus in the *SaarstahlArchitecture* and its characteristics. It requires the domain roles of *SteelworksServices_Role*, *SFPInventoryServies_Role*, *OrderServies_Role*, *SalesDepartmentServices_Role*, *TechInspectionServices_Role*, *PlanningDepartmentServices_Role*, and *ManufacturerServices_Role* and performs the *ManufacturerServices_Role*. The UML Interfaces are the source for these generated DomainRoles. The skeleton of the interaction protocols *MeltingContract*, *AllocateMaterialContract*, *PostMeltingContract*, etc. are generated by Mapping Rule 8.3. This means that, for instance, in the case of the *PostMeltingContract*, only the actors *orderActor*, *stwActor*, and *inventoryActor* are generated by Mapping Rule 8.3. The order in which messages are exchanged by these tree actors is only described by the internal behaviors of the autonomous entities bound to these actors.

An example of such an internal behavior is given in Section 9.1.3.4, discussing the body of the *SaarstahlArchitecture's manufacturerActor_PurchasingProcessPlan* plan.

9.1.3.3 Collaboration Diagram

The organization *SaarstahlArchitecture* includes several collaborations, in accordance to Mapping Rule 8.5, one for each collaboration use the source participant architecture makes use of. In the remainder of this section, we emphasize on the *postMelting* collaboration.

This collaboration defines in which manner the actors of the *PostMeltingContract* are bound to the domain roles the *SaarstahlArchitecture* either performs or requires. In this case, the *OrderServices_Role* is bound to the *orderActor* through the domain role binding *orderActor_ActorBinding*, the *SFPInventoryServices_Role* is bound to the *stwActor* actor through the domain role binding *inventoryActor_ActorBinding*, etc.

9.1.3.4 Behavior Diagram

Mapping Rule 8.7 generates a list of plans based on the activity diagrams of the Saarstahl supply chain. Fig. 9.14 illustrates the generated plan based on the activity diagram of Fig. 9.4. The plan

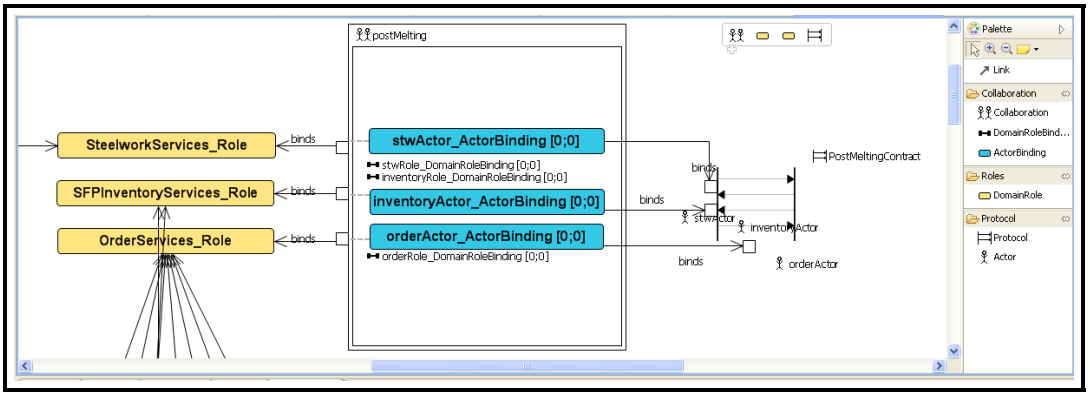


Fig. 9.13: The *postMelting* collaboration of the *SaarstahlArchitecture* organization.

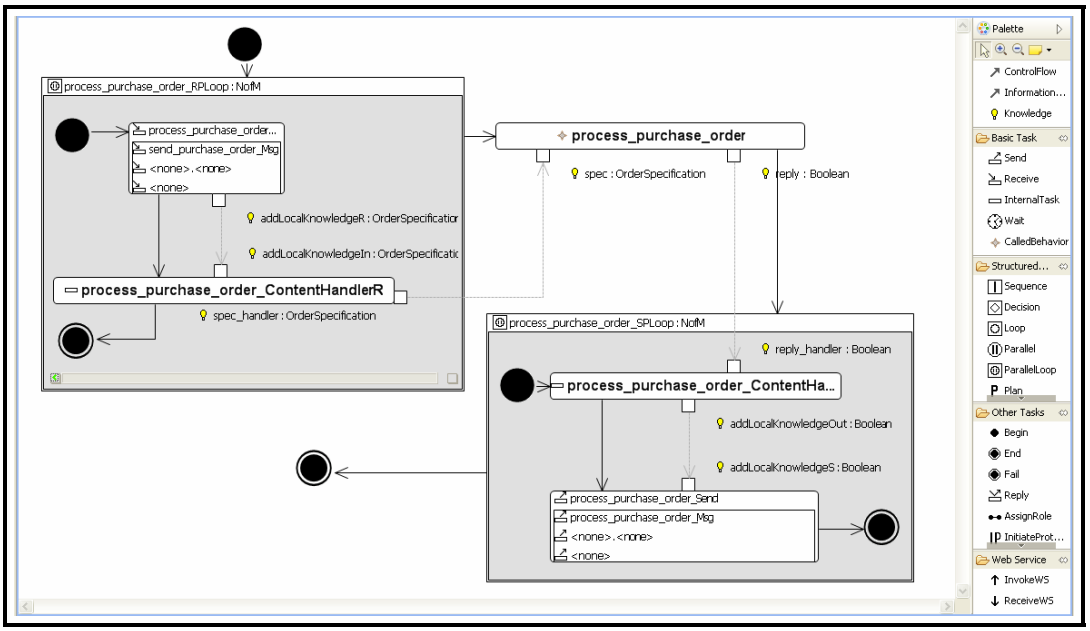


Fig. 9.14: The generated PIM4AGENTS plan for the *manufacturer*.

represents the view of the *manufacturer* and includes the activities for receiving the order from the *customer* and sending the corresponding answer.

9.1.3.5 Environment Diagram

Fig. 9.15 depicts the generated environment diagram. It includes all information and data modeled on SoaML level through UML class diagrams. For this purpose, Mapping Rule 8.11 takes any UML Class of SoaML and transfers the information contained into objects (e.g. *ProductionEvent*, *Material*, *Compolo*, etc.). Apart introducing objects, the primitive types String, Boolean, ,and Integer are automatically introduced.

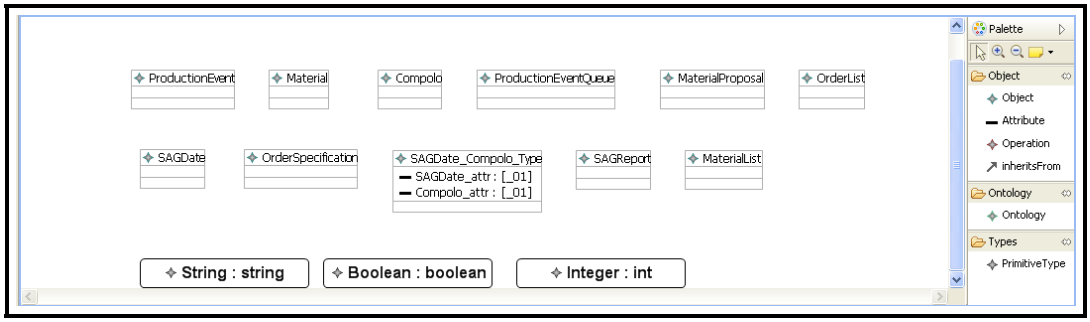


Fig. 9.15: The generated PIM4AGENTS environment diagram.

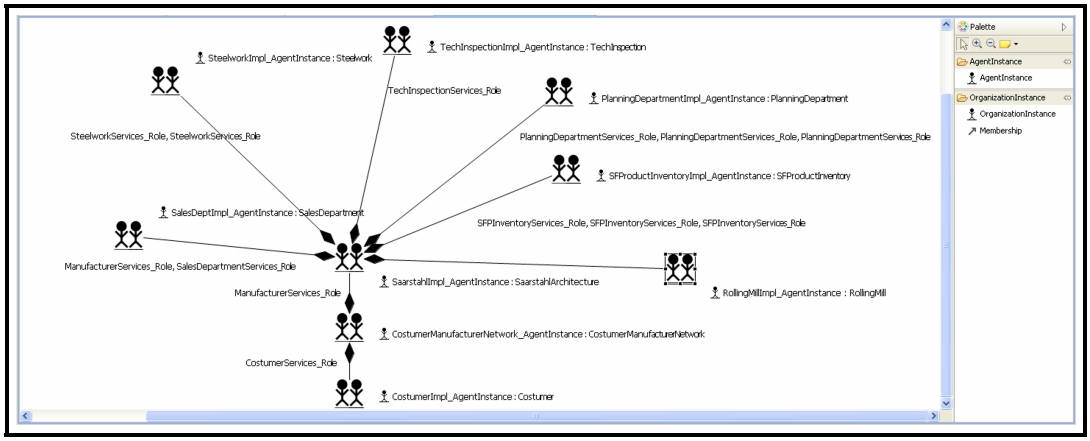


Fig. 9.16: The generated PIM4AGENTS deployment diagram.

9.1.3.6 Deployment Diagram

The generated deployment diagram is depicted in Fig. 9.16. Mapping Rule 8.6 generates for any participant and agent of the SoaML specification, one particular agent instance. At this, for any instance of a participant architecture that is part of another participant architecture, one membership is instantiated.

9.1.4 Relevance for Saarstahl

Saarstahl AG identified two major benefits by applying the DSML4MAS approach. Firstly, interoperability of existing IT-solutions supporting specified problems like a short term planning for a steelwork, a detailed planning system for a rolling mill, or some inventory management systems in between is improved. Secondly, there is a possibility of wrapping existing legacy systems of Saarstahl behind participants of the Saarstahl SOA. Thus, a SOA is created on top of the legacy systems, the generated MAS allows the flexible orchestration of Web services representing these legacy systems. The implemented system, moreover, eases the replacement of legacy systems, as new IT-solutions can be tested in parallel to legacy systems for a period of time. Agents encapsulating a legacy system are able to forward requests to the legacy system, as well as, to the new systems.

The resulting DSML4MAS design can be further extended in accordance to the DSML4MAS methodology and translated into code. For this purpose, in the case of Saarstahl's supply chain, we applied the model transformations from DSML4MAS to JADE. The reasons are twofold. Firstly, in the described settings, there is no need to provide the agents with BDI reasoning capabilities as the decisions the agents have to make a rather simple. Secondly, Saarstahl is in favor of an open source solution that could later on be internally used without additional costs.

9.2 Scheduling Product Cargos at Statoil

Statoil, a Norwegian integrated oil and gas company, is the leading operator on the Norwegian continental shelf and are also experiencing strong growth in its international production. In 2007, Statoil merged with Norsk Hydro's oil and gas division and counts now 29,000 employees and operates in 40 countries. Production in 2009 was approximately 2 million barrels oil equivalents per day. Statoil is one of the world's largest sellers of crude oil and a substantial supplier of natural gas to the European market. It has substantial industrial activity and operates 1,803 service stations in Scandinavia, Poland, the Baltic states and Russia.

9.2.1 The Scheduling Problem

The Mongstad refinery and terminal faces an intricate scheduling problem related to blending and loading of refined product cargos, which represents a defined volume of a product quality sold to a customer for delivery on a ship. Mongstad uses a two stage production process: The refinery produces components first, some of them are at second blended into a product according to some blending formula for shipment. The blending is mostly directly done onto ships, although there are some product storage tanks. This is due to the high number of different specifications and their dynamic nature.

Production of product cargoes thus depends on the availability of components, the physical constraints of the blenders, availability of appropriate jetties and the estimated arrival time of the ships to load the cargo. The main challenge at the Mongstad terminal is to optimally schedule the loading and unloading of the many ships that's arriving on an almost daily basis. The capacity of the terminal in terms of the number of ships per day is limited by a number of factors.

Firstly there are a limited number of jetties available, and each jetty has size limitations in terms of the size of the ships, particularly length and depth, and also restrictions on what products it can deliver to the ships. For product cargoes, Mongstad has four available jetties, where jetty number 2 has the largest capacity, and is the default for all cargoes if it is free. Jetty number 3 is the only jetty capable of exporting colored gasoil for the Norwegian home market. Jetty 8 and 9 are the same as jetty 2 with some capacity limitations. Serving all four jetties are the product blenders. There are two blending stations that both can blend gasoline or gasoil. This means that at most two ships can be loaded at a time. As already mentioned, most of the blending is directly on the ship, and the process includes sampling the blend at regular intervals to make sure the product is on-spec. If the blend is off-spec, it can be adjusted by adding more of one component

The components that make up the final product are perhaps the most important part. To be able to deliver a cargo, all the required components for the product must be available in the require quantity. Having the correct amounts of the correct components is a big part of getting the scheduling of the ships optimized. Sometimes when a blend goes off-spec and cannot be saved

it has to be pumped off the ship and recovered for reprocessing in the refinery, a process called slopsing. Whenever this happens, there is a risk to run short on one or more of the components, which creates a possible scheduling conflict for the planned cargoes and arriving ships. The planners responsible for the scheduling of cargoes work with product traders in the trading organization to work out a monthly lifting program based on the market. They then work with the refinery to plan what components are required to fulfill this lifting program, and also with crude oil traders to supply the required crude oil to the refinery. Assigning ships to handle the cargoes are handled by the operations staff in the trading organization, and the schedulers at the Mongstad refinery receives an advance notice of the estimated arrival time (ETA) of each ship 72, 48 and 24 hours prior to the estimated arrival time. The ultimate goal of the scheduling is to load the ships with the specified cargo within the lay-time window assigned to the ship. The cargoes on the lifting program are initially scheduled with a pretty wide five day lay-time window, which as the ETA approaches is narrowed into a three day window. Demurrage is the cost for delaying a ship beyond its assigned lay-time window and can be quite expensive. On the other hand, if the ship misses its lay-time window, the terminal operators is not required to prioritize the ship.

If we step back and take a look at this, what it boils down to is two scheduling problems interlocking with each other. The first scheduling problem is optimization of ships and cargoes based on the lifting schedule, leading to a need for certain quantities of components. The second is the optimization of the refinery production, basically determining the required crude oil qualities and quantities for producing the specified components. Combined, these two scheduling problems touch a large part of the wet supply chain.

9.2.2 DSML4MAS-Based Scheduling Product Cargos at Statoil

In this section, the application domain of the Mongstad refinery process is described and analyzed using DSML4MAS. The domain is highly dynamic, and decisions have to be made that under a high degree of uncertainty and incompleteness. Cooperation and coordination are two very important processes that may help to overcome these problems. Corresponding to the physical entities in the domain, there exist several autonomous artificial entities in the MASs (e.g. refinery, jetties, etc.) that can communicate with each other in accordance to pre-defined protocols.

To adequately describe the Mongstad scheduling problem, we apply the DSML4MAS methodology process discussed in Chapter 5. Therefore, we start with the analyze phase by modeling the core interaction between the entities involved in the Mongstad scenario. Afterwards, we apply the endogenous model transformations to derive part of the remaining phases.

9.2.2.1 Interaction View of Statoil

Although the CNP, previously presented in Section 6.3.1, is a powerful and popular task assignment mechanism, one cannot ignore the fact that the solutions CNP produces are sometime quite far from an optimal solution. The reason for this is that, although each of the individual task assignments in the CNP is a centralized optimal decision based on the current situation, a sequence of such decision is as a whole not optimal and can in some case actually turn out to be rather poor. The reason for this is that once a decision is done it is never reconsidered even when the situation changes by newly incoming orders. To overcome this problem of the CNP we adopted the Simulated Trading (ST) (Bachem et al.; 1993) procedure. ST can be used for two different purposes:

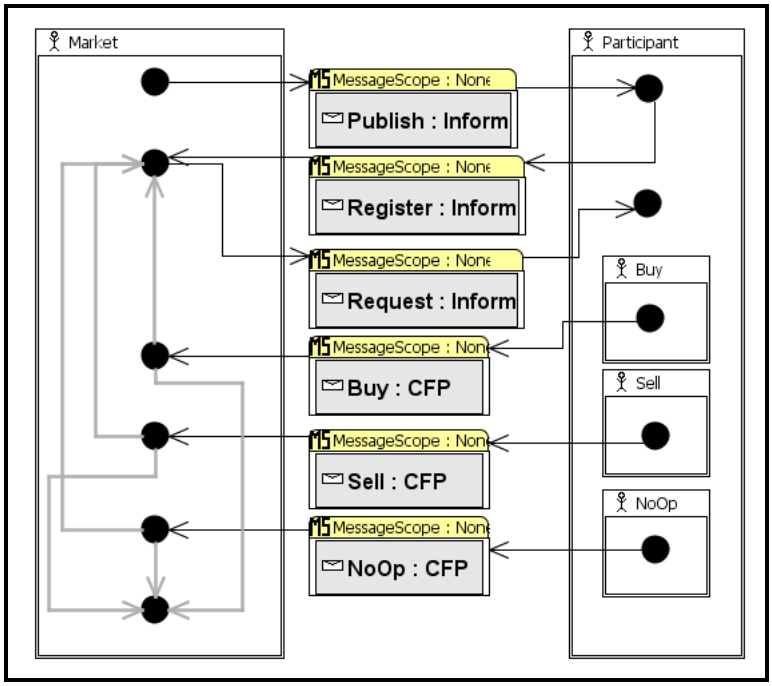


Fig. 9.17: The PIM4AGENTS interaction diagram of the Simulated Trading Protocol.

- Dynamic re-planning: If a participant realizes that it cannot satisfy the time constraints of an order because of unforeseen problems, it can initiate an ST process leading to an order reallocation satisfying the time constraints.
- Iterative optimization: Starting from the initial CNP solution, ST may be initiated to yield a better order allocation.

In (Bachem et al.; 1992), Bachem et al. present a parallel improvement heuristic for solving routing problems with side constraints. Their approach deals with the problem that n customers order different amounts of goods which are located at a central depot. The task of the dispatcher is to cluster the orders and to attach the different clusters to trucks which then in turn determine a tour to deliver the cluster allocated to them.

In a similar manner, we use STP to schedule ships at the Mongstad terminal. The overall procedure starts with a set of feasible schedules, which are obtained by CNP. The schedules are represented as an ordered list of orders that have to be produced. To guide the improvement of the initial solution, an additional processor, a market is added to the system. The task of the market is to coordinate the exchange of costumers orders between the different jetties. To do this, it collects offers for buying and selling orders coming from the jetties in the system.

The main idea is to let STP simulate a stock exchange, where the jetties can offer their current orders at some specific "saving price" and may buy orders at an "insert price". While getting sell and buy offers the market maintains the trading graph and tries to find an order exchange that optimizes the global solution.

Fig. 9.17 depicts the ST protocol in DSML4MAS notation. Like the CNP, it can be described in a nice manner using DSML4MAS. For this purpose, we introduce five actors, i.e. *Market*, *Participant*, *Buy*,

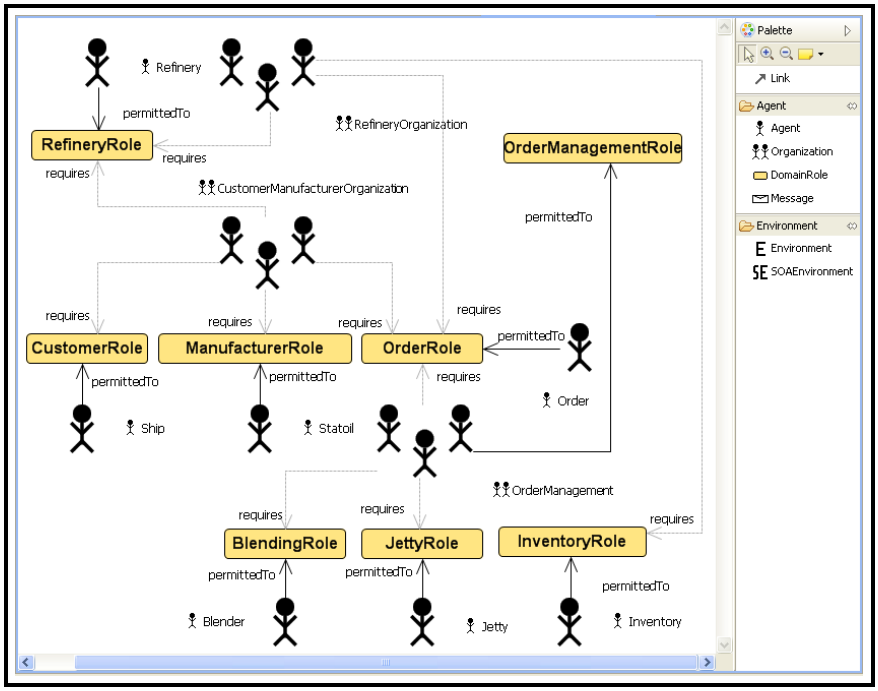


Fig. 9.18: The PIM4AGENTS MAS diagram.

Sell, and *NoOp*. The latter three are subactors of the *Participant* actor. The exchange of messages is as follows: The *Market* starts by informing the *Participants* about new orders that need to be assigned. This is done through the *Publish* ACL message. Based on the requested components, the *Participants* start to register. After receiving the *Request* message, based on their current situation, they start to send out either *Buy*, *Sell*, or *NoOp* messages.

9.2.2.2 Multiagent View of Statoil

Fig. 9.18 depicts the MAS diagram of the Statoil Mongstad scenario. It includes the basic autonomous entities like the organizations *OrderManagement*, *RefineryOrganization*, and *CustomerManufacturerOrganization*, and the agents *Refinery*, *Order*, *Statoil*, *Ship*, *Inventory*, *Jetty*, and *Blender*. Moreover, the domain roles from the role view are utilized in the sense that these are either declared as performed or required by the autonomous entities. The domain roles *RefineryRole*, for instance, is required by the *CustomerManufacturerOrganization* organization and performed by the *Refinery* agent.

9.2.2.3 Organization View of Statoil

Fig. 9.19 depicts the three core organizations *OrderManagement*, *RefineryOrganization*, *CustomerManufacturerOrganization* and their required and performed domain roles. Moreover, the diagram represents the interaction protocols used by these organizations to coordinate their members. The *CustomerManufacturerOrganization*, for instance, uses the *SimulatedTradingProtocol* and the *NoticeProtocol* to instantiate and schedule the customers orders. The *SimulatedTradingProtocol* is

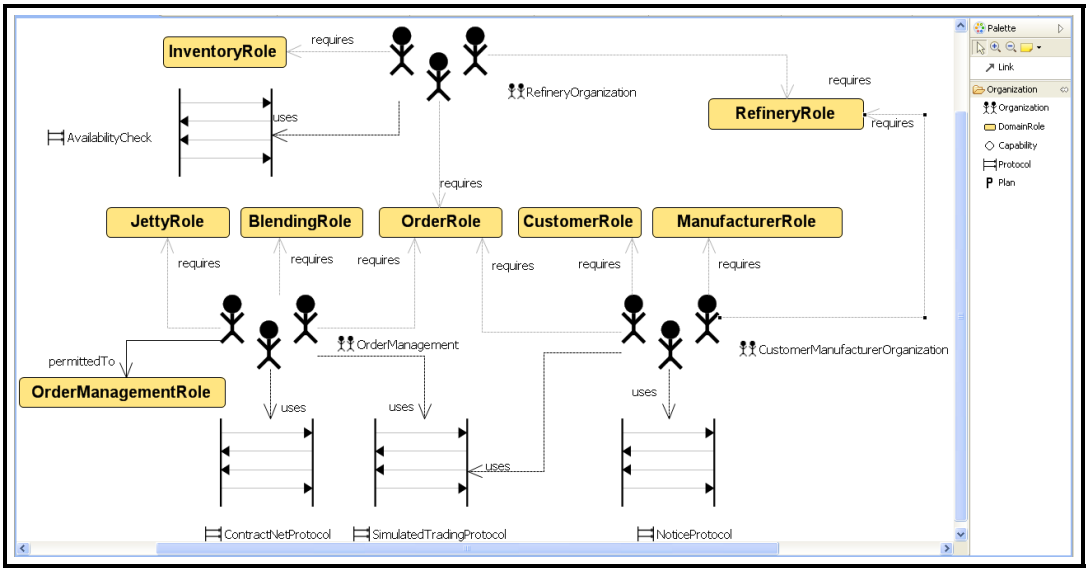


Fig. 9.19: The PIM4AGENTS organization diagram.

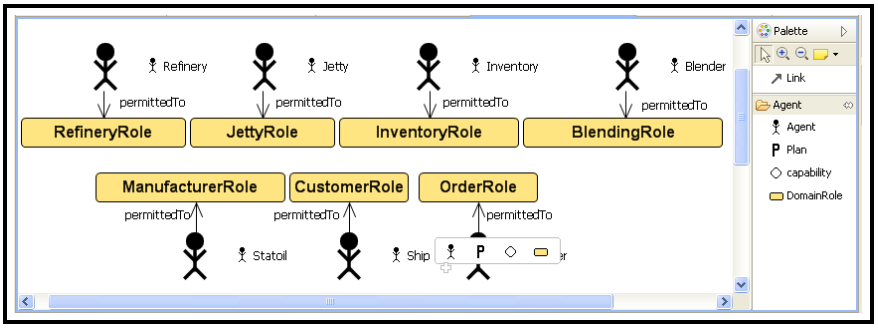


Fig. 9.20: The PIM4AGENTS agent diagram.

also used in combination with the *ContractNetProtocol* to schedule the orders assigned to the *Jetty* agents.

9.2.2.4 Agent View of Statoil

The agents of the Mongstad case are depicted in Fig. 9.20. These are the agents *Refinery*, *Jetty*, *Inventory*, *Blender*, *Statoil*, *Ship*, and *Order*. In addition, the agent diagram illustrates the agents' performed roles *RefineryRole*, *JettyRole*, *InventoryRole*, *BlendingRole*, *ManufacturerRole*, *CustomerRole*, and *OrderRole*.

9.2.2.5 Behavior View of Statoil

Fig. 9.21 depicts the plan for any participant of the Simulated Trading protocol produced by the endogenous model transformation of the DSML4MAS process model. The plan starts with

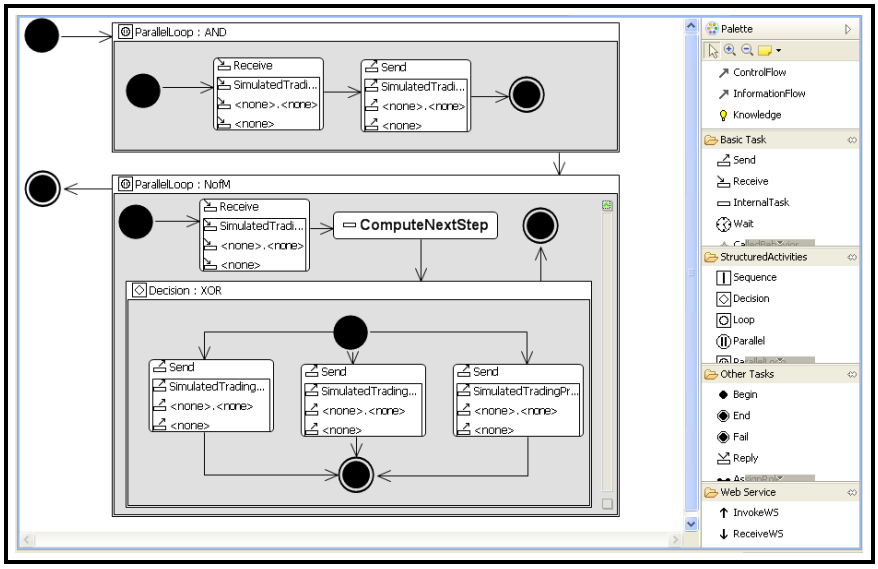


Fig. 9.21: The PIM4AGENTS behavior diagram.

receiving the *SimulatedTradingProtocolPublish* in the *ReceiveSimulatedTradingProtocolPublish* task and sending the corresponding answer *SimulatedTradingProtocolRegister* message in the *SendSimulatedTradingProtocolRegister* task. This is done in a parallel loop manner expressing that more than one message could be received and sent, depending on the number of agent instances bound to the *Market* actor, which is normally one in the case of the Simulated Trading protocol. After receiving the *SimulatedTradingProtocolRequest* message in the *ReceiveSimulatedTrading-ProtocolRequest*, the next steps are computed in the internal task *ComputeNextStep*. Depending on this evaluation, either the *SimulatedTradingProtocolBuy*, *SimulatedTradingProtocolSell*, or *SimulatedTradingProtocolNoOp* message is sent in the corresponding send activity.

9.2.2.6 Deployment View of Statoil

Even if the scenario to be described is rather dynamic with ships arriving at and leaving the Mongstad terminal, all involved agent instances are introduced during design time in the deployment view (cf. Fig. 9.22). The deployment of the Statoil case includes the agent instances *Jetty9*, *Jetty8*, *Jetty3*, *Jetty2* of type *Jetty* as well as the agent instances *Blender1* and *Blender2* of agent type *Blender*. The agent instance *ShipManagement* of type *CustomerManufacturerOrganization* integrates the agent instances *Ship1*, *Ship2*, *Ship3* all of type *Customer*, *Statoil* of type *Statoil*, the *Refinery* of type *Refinery*, etc.

9.2.3 JACK-Based Scheduling Product Cargos at Statoil

In accordance to the DSML4MAS process model, in order to generate code, either the JACK or JADE platforms and their corresponding model transformations can be selected. In the Statoil scenario, the model transformation to JACK is the preferred choice. In the following, the generated team and process views of JACK are given.

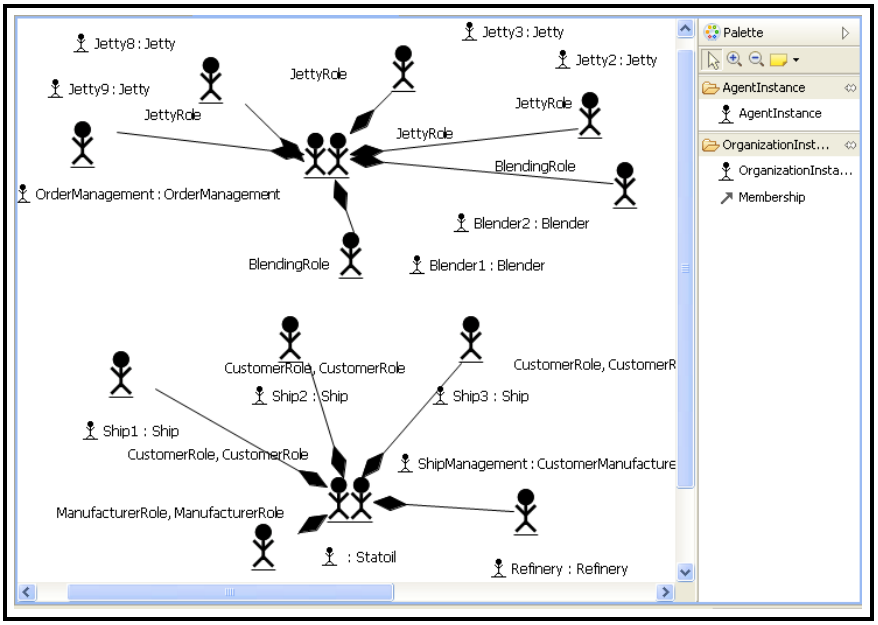


Fig. 9.22: The PIM4AGENTS deployment diagram of the Mongstad scheduling problem.

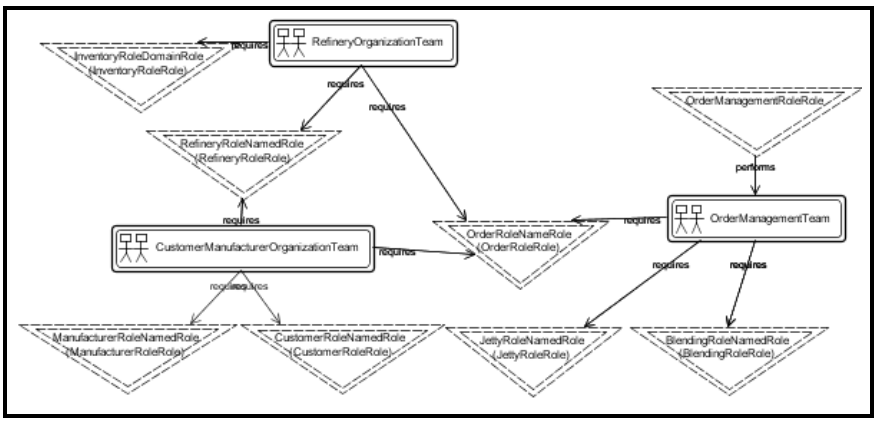


Fig. 9.23: The team view of the generated JACK model based on the organizations in the PIM4AGENTS model.

9.2.3.1 Team View of Statoil

The corresponding team view of the generated JACK model is depicted in Fig. 9.23. It illustrates the teams *RefineryOrganizationTeam*, *CustomerManufacturerTeam*, and *OrderManagementTeam* produced by Mapping Rule 7.1. Moreover, the named roles *InventoryRoleNamedRole*, *Refinery-RoleNamedRole*, *OrderRoleNamedRole*, *ManufacturerRoleNamedRole*, *CustomerRoleNamedRole*, *JettyRoleNamedRole*, and *BlendingRoleNamedRole*.

Any team generated on the base of an agent in the PIM4AGENTS model is depicted by Fig. 9.24. Consequently, Mapping Rule 7.3 produces the teams *Refinery*, *Jetty*, *Inventory*, *Blender*, *Statoil*,

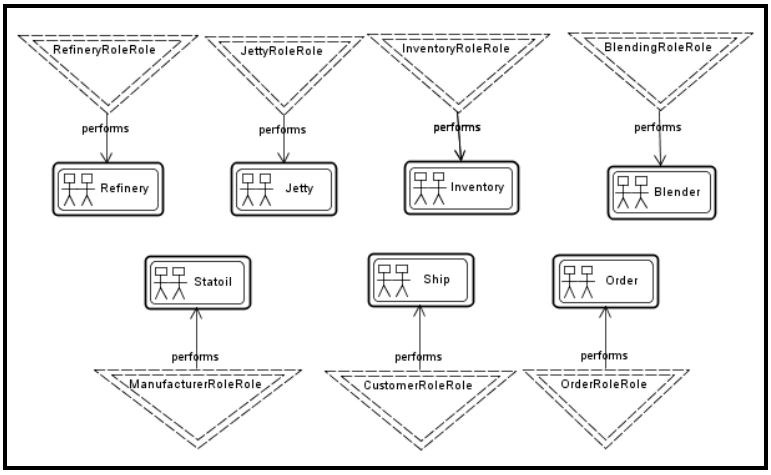


Fig. 9.24: The team view of the generated JACK model is based on the agent types in the PIM4AGENTS model.

Ship, and *Order*. Moreover, these teams perform the roles generated by Mapping Rule 7.5. These are *RefineryRoleRole*, *JettyRoleRole*, *InventoryRoleRole*, *ManufacturerRoleRole*, *CustomerRoleRole*, and *OrderRoleRole*.

9.2.3.2 Process View of Statoil

Fig. 9.25 illustrates parts of the team plan produced by Mapping Rule 7.3 on the input given in Fig. 9.21. The team plan starts with receiving the *RequestEvent* by the *Market* agent(s). When handling this event, the participant agent(s) starts to decide based on a pre-defined cost function whether to buy, sell, or perform no operation. The particular event produced by Mapping Rule 7.4 (e.g. *SellEvent*, *BuyEvent*) is then sent to the *Market*. If a good solution has been found, this process stops, otherwise the *Market* again requests operations.

9.2.4 Relevance for Statoil

The overall objective of Statoil is to increase the support for flexible event and action management. At this, Statoil is interested in an improved scheduling algorithms for future IT implementations at Mongstad and hence reduced costs at the Mongstad terminal.

By using DSML4MAS, Statoil was able to model the basic architecture of the Mongstad terminal case study. The produced design could then transferred into a corresponding PIM4AGENTS model, which was then further refined in terms of the distributed algorithms. For this purpose, the agent interaction protocols Contract Net Protocol and the Simulated Trading Protocol were utilized. The refined design was translated to the agent platform JACK, and further refined in terms of an adequate cost function. Apart from the manual refinements, the code generator for JACK nearly produces executable code.

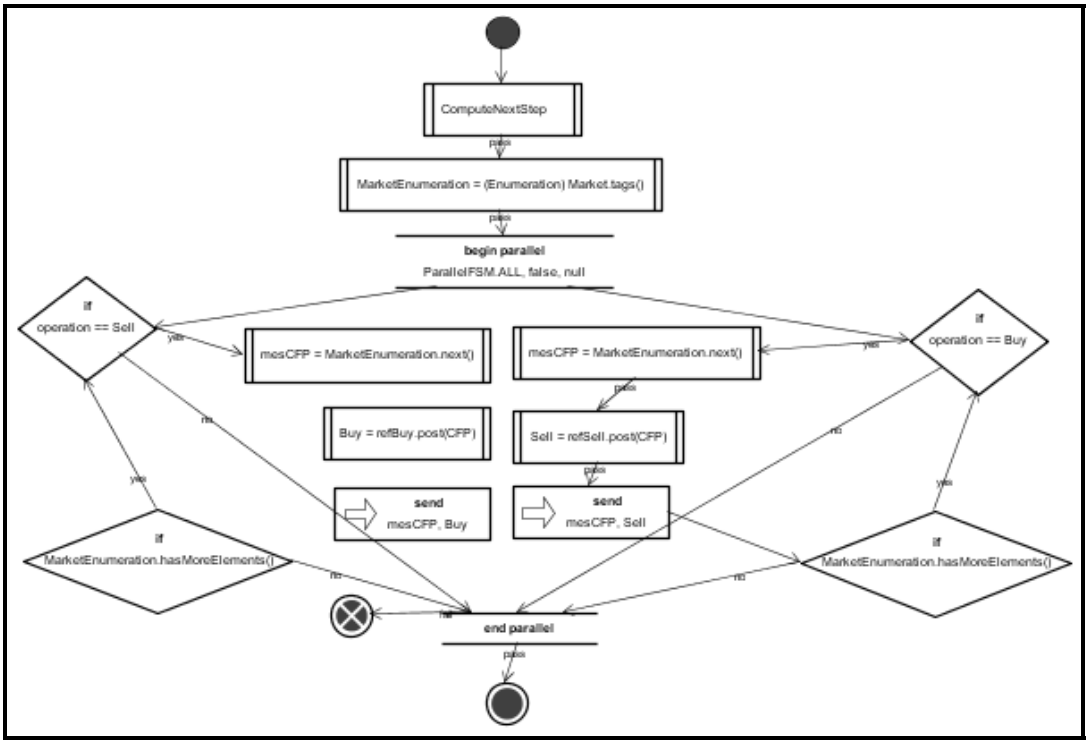


Fig. 9.25: The team plan of the generated JACK model.

9.3 Bottom Line

To demonstrate the practical usefulness of DSML4MAS and how this may increase the possibility of AOSE in being adopted by industry, in this chapter, we indicated how to utilize DSML4MAS in two real-world industrial scenarios.

The first use case deals with the production chain of the Saarstahl AG, which consists of a multitude of specialized and complex metallurgical manufacturing processes with several dependencies among them. The use case execution includes (i) the service-oriented modeling of the Saarstahl AG supply chain using SoaML, (ii) the model transformation between SoaML and DSML4MAS and (iii) the generated design on the DSML4MAS level. In a last step, the generated DSML4MAS model is then transformed to an executable JADE implementation. The main advantage, Saarstahl sees in the presented approach is to improve the interoperability of the existing legacy systems along the supply chain. The interaction between them can easily be described on an abstract level and stepwise transformed to an executable implementation.

The second use case deals with the requirements of the Statoil terminal at Mongstad. For the specification of this use case scenario, we apply the DSML4MAS methodology by starting with analyzing the problem domain using interaction protocols provided by DSML4MAS. These specifications are mapped to corresponding concepts of the remaining diagrams, which are in a second step further refined to meet the detailed requirements. Finally, the generated DSML4MAS model is transformed to JACK.

10. Comparison with State of the Art in Agent-Oriented Software Engineering

This chapter aims at providing the reader with a systematic and comprehensive evaluation of agent-based design methods. Hence, the emphasis of this chapter is to explore and analyze the similarities and differences between DSML4MAS and ten of the most cited AOSE methodologies and modeling languages. Ideally, such an analysis is based on a comprehensive evaluation framework, which allows to clearly identify the strengths and weakness of the chosen agent methodologies. The presented study is therefore based on a feature analysis approach and is inspired by the work done by Tran and Low (Tran and Low; 2005). Our feature analysis approach bases on 19 criteria, grouped by five categories. While some of these criteria base on the work of Tran and Low, others have been purposefully introduced to emphasize on the evaluation of complex systems related features (e.g. notation, viewpoints, etc.).

Structure of this Chapter In Section 10.1, the evaluation framework used in this chapter to compare DSML4MAS with existing agent-based design approaches is proposed. Afterwards, Section 10.2 discusses the most interesting and well-known agent-oriented software engineering methodologies and modeling languages and names the advantages and disadvantages of each single approach with respect to our evaluation framework. Section 10.3 compares the results of this evaluation with the characteristics and features of DSML4MAS. Finally, Section 10.4 concludes this chapter.

10.1 Evaluation Framework

Several evaluation frameworks have been proposed to adequately evaluate and compare existing agent-based methodologies or other approaches to model MASs. Examples are for instance the evaluation frameworks presented in (Cuesta et al.; 2003; Sudeikat et al.; 2004; Cernuzzi and Zambonelli; 2008; Shehory and Sturm; 2001; Lin et al.; 2007; Cossentino et al.; 2009). In the introduction of this thesis, we previously argued that agent technology still faces many challenges in being adopted by industry and possibly taking over from objects technology as the dominant software development technology. We, furthermore, presented several obstacles, we consider important to solve in order to provide mature development methodologies for agent-based systems. In order to evaluate existing approaches and compare them with DSML4MAS, we therefore propose a comparison framework that is built upon the obstacles we have identified and properties derived from a number of related surveys on comparing AOSE methodologies. These obstacles and evaluation properties are as follows:

Main Concepts Agent-based systems and MASs in particular differ from normal object-oriented languages as agents are considered as autonomous, proactive, reactive and offer a certain degree of social ability (cf. Definition 2.1.5). Consequently, the vocabulary of any agent-based design approach should provide mechanisms to express these characteristics and hence approve the following questions:

- Does the design approach allow modeling of agents?
- Does the design approach allow modeling of organizations or other social structures?
- Does the design approach provide techniques for modeling goals?
- Does the design approach provide representations detailing plans and their bodies to describe how an agent accomplishes its goals or responds to external events?
- Does the design approach support the modeling of the agents' environment?
- Does the design approach provide techniques to define interaction protocols?

Methodology The vocabulary of the modeling language is certainly considered as a crucial part of any software method. However, in order to design MASs, certain functionalities are necessary to provide. In particular, if the language contains concepts that are normally used on different abstraction layers. To assist the developers in efficiently designing the MAS, usually a process is offered that guides the developer through the different development phases. Quality characteristics of a good and adequate methodology process are as follows:

- Does the process model cover the whole life-cycle?
- Does the process support the deployment?
- Does the methodology allow modeling of the MAS from different viewpoints?
- Does the methodology support a seamless transfer from more abstract phases to more concrete ones?
- Does the process allow automatically transferring one phase into another?
- Can the modeling technique be used from scratch, without having to walk through the entire phases?

Tool Support Apart from the language's vocabulary and its methodology, adequate tool support in terms of a good graphical modeling notation, consistency checks as well as integrated code generators eases the complex tasks of requirement analysis, design, and implementation. However, as pointed out in (Sudeikat et al.; 2004), evaluating tool support is somehow a difficult task as the their usability is influenced by many aspects. To evaluate the tool support provided by the chosen modeling approaches in a proper manner, we measure the kind of support with the help of the following questions.

- Does the modeling tool allow the design of the MAS from different viewpoints?
- Does the modeling tool support the validation of the design?
- Does the modeling tool provide mechanisms for automatic code generation?
- Is the modeling tool integrated in a generic modeling environment?

Formal Semantics Existing agent-based modeling approaches mainly focus on the process guiding the developer through the different development phases. However, in order to ensure that the design is consistent across the different phases, the precise language's semantics is needed that go beyond the information part of the metamodel or UML profile. In particular, we consider both important, the static semantics as well as the dynamic semantics. This is also reflected by the following properties.

- Does the language offer a static semantic?
- Does the language offer a dynamic semantic?

		Weight	Scale
Main Concepts	Agents	3	0 = No Support, 1 = Low, 2 = Medium, 3 = Large
	Organizations	2	0 = No Support, 1 = Low, 2 = Medium, 3 = Large
	Goals	2	0 = No Support, 1 = Low, 2 = Medium, 3 = Large
	Plans	2	0 = No Support, 1 = Low, 2 = Medium, 3 = Large
	Environment	2	0 = No Support, 1 = Low, 2 = Medium, 3 = Large
	AIP	2	0 = No Support, 1 = Low, 2 = Medium, 3 = Large
Methodology	Whole Lifecycle	2	0 = No Support, 1 = Partial, 2 = Full
	Deployment	1	0 = No Support, 1 = Support
	Model-driven	1	0 = No Support, 1 = Support
	Modeling Language	2	0 = No Support, 1 = Support
Tool Support	Viewpoints	1	0 = No Support, 1 = Support
	Validation	2	0 = No Support, 1 = Support
	Code Generation	2	0 = No Support, 1 = Support
	Generic Modeling Environment	1	0 = No Support, 1 = Support
Semantics	Static Semantics	3	0 = No Support, 1 = Support
	Dynamic Semantics	2	0 = No Support, 1 = Support
Interoperability	Executable Implementation	3	0 = No Support, 1 = Manual, 2 = Skeleton, 3 = Full
	Platform Support	2	0 = No Support, 1 = One Platform, 2 = Various Platforms
	Interoperability	2	0 = No Support, 1 = Support

Tab. 10.1: A summary on the requirements of our evaluation framework.

Code Generation and Interoperability In these days, the importance of automatically generating code is increasing. Especially in the area of AOSE, the implementation is often too complex to understand for non experts. This issue certainly hampers the break through of agent-based computing in industry, which could be compensated by automatic code generators defined in accordance to MDD. Moreover, (modeling) languages do not exist in pure isolation, instead and in particular in industrial settings, there are plenty of more business-oriented languages available that need to be integrated with the MAS and the corresponding language when aiming at bringing MASs to industry. Based on these facts, we consider the following features necessary to be offered by agent-oriented design approaches.

- Does the approach allow the automatically generation of executable code?
- Does the approach provide code generators for different agent-based platforms?
- Does the approach provide mechanisms to combine with other languages?

The criteria just presented are summarized in Table 10.1. As some of the criteria are more important than others, we also establish a weight of each requirement, as well as a certain scale in terms of

measured characteristics. The weight is a number between 1 (lowest importance) and 3 (highest importance) expressing if the requirement is mandatory or optional. Scale, on the other hand, explains the different levels of support. It is important that the scale is precisely defined to easily evaluate the target language. The importance level indicated by the weight is subjective, however, these weights are critical to ensure that evaluated languages are ranked higher if they fulfill the most important requirements.

10.2 Agent-Based Modeling Techniques

AOSE design approaches mainly try to suggest a clean and disciplined approach to analyze, design, and develop MASs, using specific methods and techniques. They typically start from a metamodel or UML profile that identifies the basic abstractions to be exploited in agent-based development. On the base of this, they exploit and organize these abstractions to define guidelines on how to model correctly. These design approaches are either agent-based modeling languages (e.g. Agent Modeling Language (AMOLA, (Spanoudakis and Moraitis; 2008a))) or methodologies (e.g. AALAADIN (Ferber and Gutknecht; 1998), MESSAGE (Caire et al.; 2002), Nemo (Huget; 2002a), ROADMAP (Juan and Sterling; 2003), ASEME (Spanoudakis and Moraitis; 2007), or SODA (Omicini; 2001)).

In the remainder of this section, the most prominent agent-based modeling and design approaches are presented and discussed in terms of their main concepts, provided tool support, specified semantics, as well as code generation facilities.

10.2.1 Agent UML

Agent UML (AUML) (Odell et al.; 2000; Bauer et al.; 2001; Huget and Odell; 2004) is an extension of UML to overcome the limitations of UML with respect to the development of MASs. AUML results from the cooperation between the OMG and FIPA) aiming at increasing acceptance of agent technology in industry. AUML and in particular the AUML interaction diagram is considered as de facto standard to model agent interactions. It is used as add-on by various agent-oriented methodologies like for instance Prometheus (see Section 10.2.10), Tropos (see Section 10.2.7), or INGENIAS (see Section 10.2.9).

Main Concepts The main parts of the AUML's modeling language are the agent class diagram and the agent interaction protocol diagram that extend UML's state and sequence diagrams. In the agent class diagram, agents are specified by extending UML classes with an agent stereotype. A role thereby defines the behavior of an agent within the society, where an agent can have multiple roles or it can change its role(s) during the execution. Agents evolving in MASs, finally, belong to one or several organizations that define the agents' roles and the relationships between them. Furthermore, classes exist that model capabilities, services and protocols. The protocol diagram combines the notation of the sequence diagram and the state diagram for the specification of AIPs. In UML, the concept of a role is an instance focused term, whereas in AUML, a role defines a set of agents satisfying distinguished properties, interfaces, service descriptions or having a distinguished behavior. Agents can perform various roles within one interaction protocol.

Even if the interaction part of AUML is applied by many other agent methodologies, its expressiveness is limited. The main reason for this is that the AIPs of AUML strongly base

on UML sequence diagrams, which does not allow to describe the interaction between one-to-many entities like this is needed for modeling complex interactions (e.g. CNP). Likewise, AUML does not allow representing the agent's knowledge about the environment's status (Juneidi and Vouros; 2004), which makes the agent adapt to changes in the dynamic environment difficult.

Methodology AUML originally serves as a pure modeling language. An extension of AUML has been presented in (de Cerqueira Gatti et al.; 2007) in the sense of a methodology for AUML called AUML-BP. An AUML-BP iteration includes the phases *requirements*, *analysis and design*, and *implementation and tests*. However, the implementation seems to be done manually as, for instance, generic model transformations between AUML and JADE do not exist.

Tool Support AUML does not provide any graphical tool support (Peres and Bergmann; 2005). However, as AUML is based upon UML through profiles, any UML-capable tool can be used for this purpose. In (Winikoff; 2005), Winikoff presented a textual notation for AUML interaction protocols, which takes AUML protocols in a textual format and produces a graphical representation.

		Weight	Scale
Main Concepts	Agents	2	2
	Organizations	2	1.33
	Goals	0	0
	Plans	1	0.66
	Environment	0	0
	AIP	2	1.33
Methodology	Whole Lifecycle	2	2
	Deployment	0	0
	Model-driven	0	0
	Modeling Language	1	2
Tool Support	Viewpoints	1	1
	Validation	0	0
	Code Generation	0	0
	Generic Modeling Environment	0	0
Semantics	Static Semantics	0.5	1.5
	Dynamic Semantics	0.5	1
Interoperability	Executable Implementation	0	0
	Platform Support	0	0
	Interoperability	0	0
		12	12.83

Tab. 10.2: A summary on AUML's characteristics.

Formal Semantics Huget (2002d) validated AUML protocol diagrams using Promela[1] (del Mar Gallardo and Merino; 1999) and Spin[2] (Ben-Ari; 2008) and proposed a description

[1] PROMELA is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and borrowing the notation for I/O operations from Hoare's CSP language.

[2] Spin is a model checker used for the formal verification of distributed software systems.

language called AXF, which allows to represent textually protocol diagrams (Huget; 2002c). Similarly, Mokhati et al. (2007) propose the verification of AUML's interaction protocols using Maude's model checker (Eker et al.; 2002), whereas Cabac and Moldt (2004) describe the formal semantics of AUML's interaction protocols using Petri nets. Hence, a rich set of formal specifications is available for AUML, however, these specifications mainly focus on the interaction part, the semantics of the agent class diagrams are often not considered. This limitation certainly hampers the usefulness of the formal specification.

Interoperability In (Ehrler and Cranefield; 2004), the Plug-in for AUML Linking (PAUL) is presented that allows the automatic interpretation of AUML interaction protocols. This is mainly achieved by building an AUML interpreter and integrating application-specific code using Java. However, in its current version, PAUL only supports agent interaction protocols.

Further code generation approaches are described in (Quenum et al.; 2006; Huget; 2002b; Dinkloh and Nimis; 2003; Doi et al.; 2005). Like in the case of the formal semantics, the presented approaches mainly deal with AIPs. Consequently, only skeleton code can be generated as the remaining design is missing (e.g. agents, organization, etc.) in order to produce fully executable code.

Table 10.2 summarizes the core characteristics of AUML in accordance to our evaluation framework. AUML is one of the first approaches taking UML as base and extending it for modeling agents and MASs. Even if the main concepts of agent-based system development are covered in a manner that precisely allows designing MASs, the semantics of the initial version is—like for any UML profile—not formally defined. AUML's formal semantics only focus on AIPs. In terms of methodology support, originally, AUML was developed as pure modeling language. A methodology process was later on built on top of AUML covering all important phases including an implementation phase, which unfortunately does not allow to automatically produce code. The overall score of AUML is 12.83.

10.2.2 Agent-Object-Relationship Modeling Language

The Agent-Object-Relationship Modeling Language (AORML) (Wagner; 2003, 2002) is a modeling language for MASs that is mainly inspired by the Agent-Oriented Programming proposal of (Shoham; 1993). A UML profile for AORML is defined in (Wagner; 2002).

Main Concepts The main concept of AORML is the *AgentType* which is in the AORML sense an entity like an event, an action, a claim, a commitment, or an ordinary object. Agents and objects, respectively, form the active and passive entities, while actions and events are the dynamic entities of the system model. Commitments and claims establish a special type of relationship between agents. These concepts are fundamental components of social interaction processes and can explicitly used to achieve coherent behavior. Beside the simple agents, AORML also models institutional agents, which are usually composed of other (institutional) agents that act on its behalf. The interaction between agents is, like in the case of AUML, restricted to two agents. This means that sending message to multiple agent instances is, like in the AUML case, not supported. Moreover, in its current version, AORML does neither include concepts to model mental concepts like goals nor proactive behaviors of agents (Wagner; 2003). Additionally, as pointed out in (Xiao and Greer; 2005; da Silva et al.; 2004), the manner in which agents, objects and rules interplay together is not adequately described.

		Weight	Scale
Main Concepts	Agents	2	2
	Organizations	2	1.33
	Goals	0	0
	Plans	0	0
	Environment	1	0.66
	AIP	2	1.33
Methodology	Whole Lifecycle	1	1
	Deployment	0	0
	Model-driven	0	0
	Modeling Language	1	2
Tool Support	Viewpoints	1	1
	Validation	0	0
	Code Generation	0	0
	Generic Modeling Environment	0	0
Semantics	Static Semantics	0	0
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	1	1
	Platform Support	1	1
	Interoperability	1	2
		13	13.33

Tab. 10.3: A summary on AORML's characteristics.

Methodology AORML is rather a modeling language than a methodology. Though some methodological directions on how to use AORML for software development have been identified in (Wagner and Taveter; 2004). In the *analysis phase* of the corresponding RAP/AOR methodology, the interaction between agents is specified that is then transformed to the agents' internal perspectives supporting the specified requirements. These internal AOR models are afterward refined into an implementation model of the particular target language (e.g. Java). An external AOR model may comprise (i) the agent diagram focusing on the agents of the domain, certain relevant objects, and the relationships among them, (ii) the interaction frame diagram, which depicts the action event types and commitment/claim types that determine the feasible interactions between two agents, (iii) the interaction sequence diagram focuses on instances of interaction processes and finally, (iv) the interaction pattern diagram depicts the interaction patterns expressed by defining reaction rules illustrating an interaction process type.

Tool Support A Microsoft Visio template exist for AOR modeling (Wagner; 2003), providing the specific graphical shapes of the AOR modeling elements.

Formal Semantics -

Interoperability In (Taveter and Wagner; 2008), a conceptual mapping between AORML and JADE is presented, however, model transformations automatically generating code are not given. To leverage the advantages of both agent-oriented design techniques, a manual mapping between Tropos and AORML is discussed in (Guizzardi-Silva Souza et al.; 2003).

Table 10.3 summarizes the core characteristics of AORML. The main advantage of AORML is certainly the provided modeling capabilities. However, the generated design cannot be automatically transferred into an agent implementation. In addition, the provided tool support is rather weak, a formal semantics is lacking, too. The overall score of AORML is 13.33.

10.2.3 Agent Modeling Language

The Agent Modeling Language (AML) (Cervenka et al.; 2004; Ivan et al.; 2006) was designed to address the specific qualities offered by MASs that are difficult, or impossible, to model with object-oriented modeling languages like UML. Aspects provided by AML to model MAS include constructs for modeling (i) architectural aspects of MASs like, for instance, ontologies and social aspects (i.e. organizational units), (ii) global aspects like MAS behaviors in terms of communicative interactions modeled within protocols, and (iii) mental aspects of agents like beliefs, goals, and plans. For an adequate representation of AML's concepts, a UML profile was developed, concepts further extending the vocabulary of UML are provided as metamodel.

Main Concepts The agent type is used to specify the type of agents, i.e. self-contained entities that are capable of interactions, observations and autonomous behavior within their environment. The environmental type is used to model the type of a system's inner environment. Moreover, the resource type is used to model the type of resources within the system, which are either physical or informational entities. To accommodate the special needs for modeling social structures, social behavior and social attitudes, AML provides the following entities: The organizational units are used to model social entities that can evolve in the system. The role type is used to model capabilities, behaviors, observations, relationships, participation in interactions, and offered services. To support modeling of interactions, AML provides different kinds of UML extensions. These can be subdivided into (i) generic extensions to UML interactions, (ii) speech act based extensions to UML interactions, (iii) observations and effecting interactions, and (iv) services (see (Trencansky and Cervenka; 2005)). AML furthermore extends the capacity concept of UML to abstract and decompose behavior by two modeling elements: capability and behavior fragments. The concept of capability is defined as abstract specification of a behavior, which allows reasoning about and operations on that specification. The behavior fragment is a specialized behaviored semi-entity type used to model a coherent re-usable fragment of behavior and related structural and behavioral features. It enables the decomposition of a complex behavior into simpler and (possibly) concurrently executable fragments. Finally, for modeling mental concepts, AML provides, among others, the concepts of goals and plans. The former is a specialized UML class used to model conditions or states of affairs, the latter is a specialized UML activity used to model either predefined plans, or fragments of behavior from which plans can be composed.

Methodology -

Tool Support AML provides a rich set of modeling constructs that allow to model the most important aspects of MASs like social relationships, organizational units or interactions. As AML bases on UML through profiles, the notation is geared to the notation used by UML 2.0. Hence, any UML-capable tool is able to import the AML design.

Formal Semantics The static semantics of AML have been described in (Cervenka and Trencansky; 2004) using OCL. Danč 2008, moreover, demonstrated the use of Object-Z to formally

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	2	1.33
	Goals	3	2
	Plans	2	1.33
	Environment	2	1.33
	AIP	2	1.33
Methodology	Whole Lifecycle	0	0
	Deployment	0	0
	Model-driven	0	0
	Modeling Language	1	2
Tool Support	Viewpoints	1	1
	Validation	0 (?)	0
	Code Generation	2	2
	Generic Modeling Environment	1	1
Semantics	Static Semantics	1	3
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	2	2
	Platform Support	1	1
	Interoperability	0	0
		23	22.33

Tab. 10.4: A summary on AML's characteristics.

define the semantics. However, the operational semantics has not been dealt with, since the authors consider the dynamic semantics as a dependency of the specific execution platform (cf. (Cervenka et al.; 2004)).

Interoperability In (Cervenka et al.; 2006), code generators were defined based on a CASE modeling tool that translate the AML specification into code of the Living Systems Technology Suite (LS/TS) (Rimassa et al.; 2006), which is a development environment for producing applications based on software agent technology based on Java.

Table 10.4 summarizes the core characteristics of AML in accordance to our evaluation framework. All in all, the AML language offers important features to develop MASs, like (i) a static semantics based on Object-Z and (ii) a code generator for LS/TS. The only real meaningful shortcoming is the insufficient tool support to integrate the just mentioned features. The overall score of AML is 22.33.

10.2.4 Generic Architecture for Information Availability

The Generic Architecture for Information Availability (Gaia) (Zambonelli et al.; 2003; Wooldridge et al.; 2000) has been designed to explicitly model and represent the social aspects of open agent systems, with particular attention to social goals, social tasks or organizational rules. In the course of time, Gaia has further been improved, the ROADMAP methodology is only one representative for this permanent modernization.

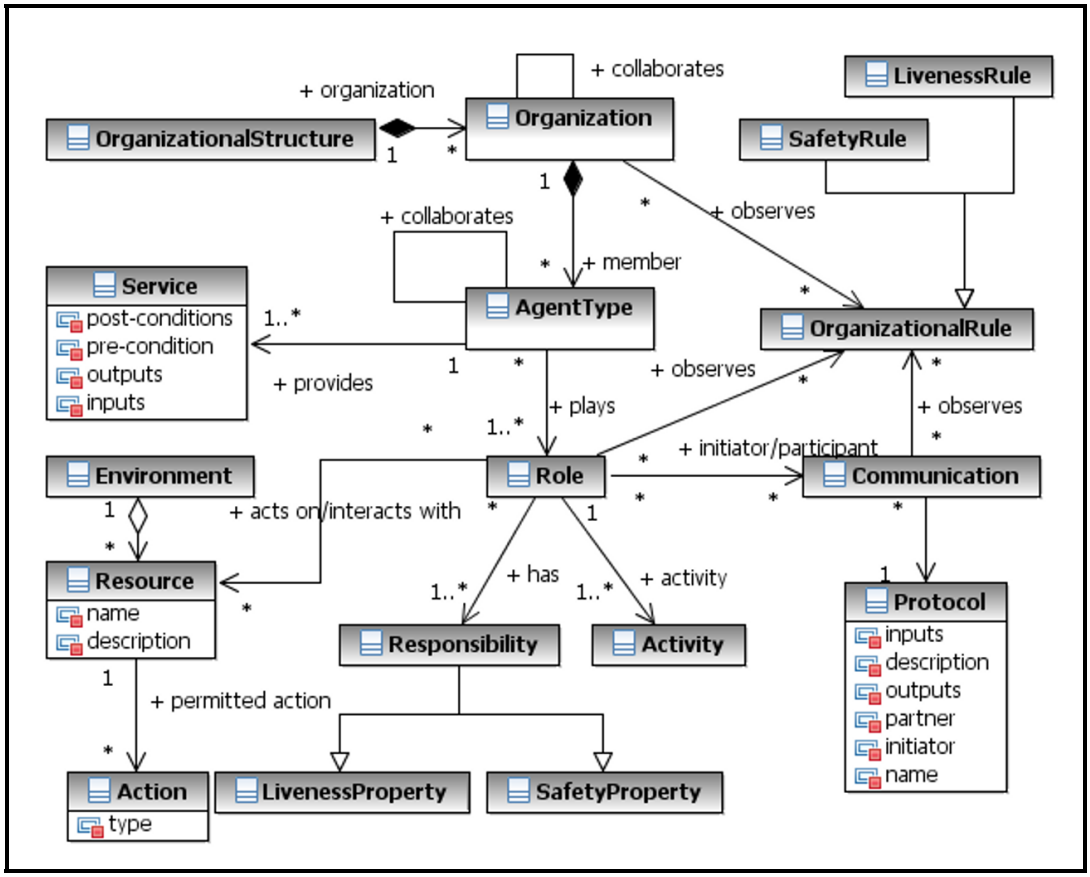


Fig. 10.1: The Gaia metamodel, in accordance to (Bernon et al.; 2005b).

Main Concepts The metamodel of Gaia is depicted in Fig. 10.1. The main concept of Gaia is *AgentType*, which is part of an *Organization*, collaborates with other *AgentTypes*, provides *Services*—having post-conditions, pre-condition, outputs and inputs—and plays several *Roles*. The concept of Role refers to certain *Activities*, has *Responsibilities* and are used within *Communications* that specify *Protocols*. The *Environment* contains *Resources* that interact with *Roles*. The ROADMAP (Juan and Sterling; 2003) metamodel extends Gaia in terms of a formal environment, knowledge models, as well as a dynamic role hierarchy to constrain the behavior of agents in organizations.

Methodology The development process of Gaia mainly consists of two phases. In the *analysis phase*, the characteristics of the MAS are analyzed to identify the necessary roles and their interactions. The generated work products include role and interaction models, as well as, a model representing social laws. In the *design phase*, the agent and service models are detailed, as well as, rough behavior descriptions for the cooperation between agents are produced.

Tool Support -

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	2	1.33
	Goals	0	0
	Plans	1	0.66
	Environment	2	1.33
	AIP	1	0.66
Methodology	Whole Lifecycle	1	1
	Deployment	0	0
	Model-driven	0	0
	Modeling Language	0	0
Tool Support	Viewpoints	1	1
	Validation	0	0
	Code Generation	0	0
	Generic Modeling Environment	0	0
Semantics	Static Semantics	1	3
	Dynamic Semantics	1	2
Interoperability	Executable Implementation	2	2
	Platform Support	1	1
	Interoperability	0	0
		16	17

Tab. 10.5: A summary on Gaia's characteristics.

Formal Semantics Miller and McBurney presented a formal semantics for Gaia liveness expressions and liveness rules, and discussed a sound and complete axiom system for them.

Interoperability In (Moraitis and Spanoudakis; 2004), conceptual mappings between Gaia and JADE have been presented. However, the model mappings have not been implemented and hence the MAS developers still need to manually convert the design made with Gaia into code. The *Gaia2Jade* process (Moraitis and Spanoudakis; 2006) proposes to perform the *implementation phase* in four stages: communication protocol definition, activities refinement, JADE behavior creation, and agent classes construction. One relevant detail in the behavior creation is that Gaia roles are transformed to high level JADE behaviors. In (Spanoudakis and Moraitis; 2009), the concrete model transformations were presented.

Table 10.5 summarizes the core characteristics of Gaia in accordance to the evaluation framework. Gaia is one of the first agent-based methodologies that has been further improved over the years. Recently, a model transformation in accordance to MDD has been presented, which provides an automatic link to JADE as execution platform. The main shortcoming of Gaia is the non-existing tool support, which limits Gaia's usability in complex scenarios. The overall score of Gaia is 17.

10.2.5 Process for Agent Societies Specification and Implementation

The Process for Agent Societies Specification and Implementation (PASSI) (Cossentino; 2005) is an agent-based methodology to design MASs. PASSI conciliates classical software engineering concepts like problem and solution domain with the potentiality of the agent-based approach

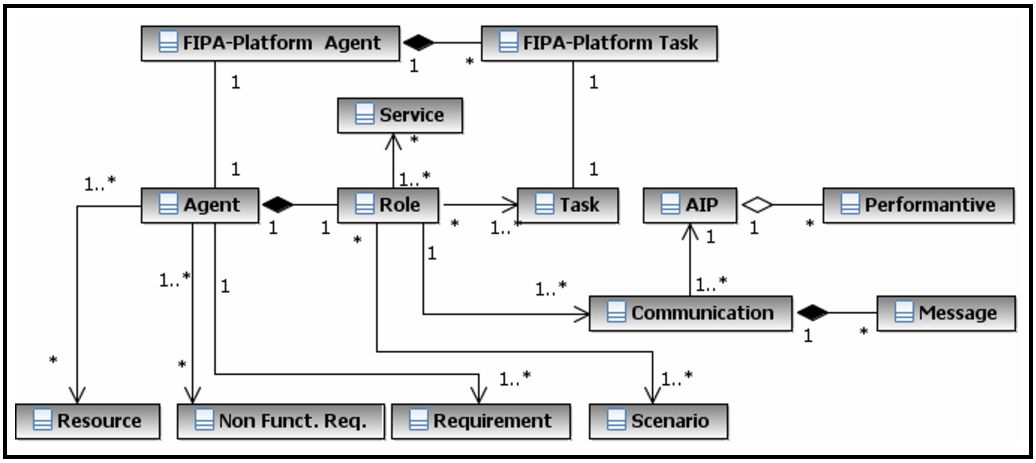


Fig. 10.2: The PASSI metamodel, in accordance to (Bernon et al.; 2005b).

by integrating design models and concepts from both OOSE and AOSE using UML notation. The convergence between agents and traditional issues of software engineering is obtained by introducing an abstraction layer called agency domain. The communication and implementation are FIPA-based. An extension to PASSI called Agile PASSI has been presented in (Chella et al.; 2004b) that leads to a faster development process mainly oriented toward the code generation. Recently, HoloPASSI (Cossentino et al.; 2007) has been proposed extending PASSI with the capability of modeling holonic MASs.

Main Concepts The PASSI metamodel covers the *FIPA-Platform Agent* concept that contains several *FIPA-Platform Tasks*. Any *Agent* is represented by a *FIPA-Platform Agent* and performs a set of *Roles* providing *Services* to solve *Tasks*. In addition, the *Role* concept is represented in *Communications* to establish *AIPs* and specify *Messages*. In addition to performing *Roles*, an *Agent* refers to *Resource*, *Non Functional Aspects* and *Requirements*. A plan in PASSI can only be represented as algorithm, no graphical support to visualize the workflow is provided. The environment agents are situated can neither be modeled.

Methodology The software development process of PASSI bases on five phases (Chella et al.; 2004a). In the *system requirements phase*, a use-case based description of the functionalities and an initial decomposition of them is produced. The *agent society phase* is used to compose a model of domain ontology, social interactions and dependencies among the agents. In the *agent implementation phase*, the solution architecture is modeled in terms of agents required, classes and methods. It includes a structural definition, as well as, behavior descriptions of the whole system. The *code phase* aims to model a solution at the code level. It is largely supported by patterns reuse and automatic code generation. In the *deployment phase*, the distribution of the system parts across a distributed platform is modeled. Furthermore, facilities are provided for testing single agents, as well as, the whole society of the MAS after deployment.

Tool Support The PASSI Tool Kit (PTK) (Cossentino and Potts; 2002) is a graphical agent-based CASE (Computer-Aided Software Engineering) tool that bases on IBM's Rational Rose [3]. The

[3] http://www-01.ibm.com/software/awdtools/developer/rose/

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	1	0.66
	Goals	1	0.66
	Plans	1	0.66
	Environment	2	1.33
	AIP	2	1.33
Methodology	Whole Lifecycle	2	2
	Deployment	1	1
	Model-driven	0	0
	Modeling Language	0	0
Tool Support	Viewpoints	1	1
	Validation	0	0
	Code Generation	1	2
	Generic Modeling Environment	1	1
Semantics	Static Semantics	0	0
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	2	2
	Platform Support	2	2
	Interoperability	0	0
		19	18.66

Tab. 10.6: A summary on PASSI's characteristics.

diagrams offered by PTK are either totally dependent by the designer, some are automatically built by the tool and others are partially composed by the tool and then completed by the designer (Chella et al.; 2004a).

Formal Semantics —

Interoperability The AgentFactory (Collier et al.; 2004) tool allows the automatically generation of pattern code in accordance to PASSI. The generated code for the target platforms FIPA-OS and JADE needs final completion.

Table 10.6 summarizes the main features of the PASSI methodology in accordance to the proposed evaluation framework. PASSI does not originally allow modeling of organizations (only the modeling of roles is supported), goals can only be expressed through tasks. Adequate support for modeling plans is missing, which is compensated by the service concept used to wrap basic, simple operations. The main advantage of PASSI is the model-driven support for various platforms even if only code skeletons are produced. The generation of the remaining code parts (i.e. the inner parts of the methods) is delegated to a repository including several patterns of agents and behaviors. The main drawback of PASSI is the missing formal semantics. The overall score of PASSI is 18.66.

10.2.6 Atelier de Développement de Logiciels à Fonctionnalité Emergente

The Atelier de Développement de Logiciels à Fonctionnalité Emergente (ADELFE) (Bernon et al.; 2003, 2005a; Picard and Gleizes; 2004) specifies a methodology to develop adaptive MASs by

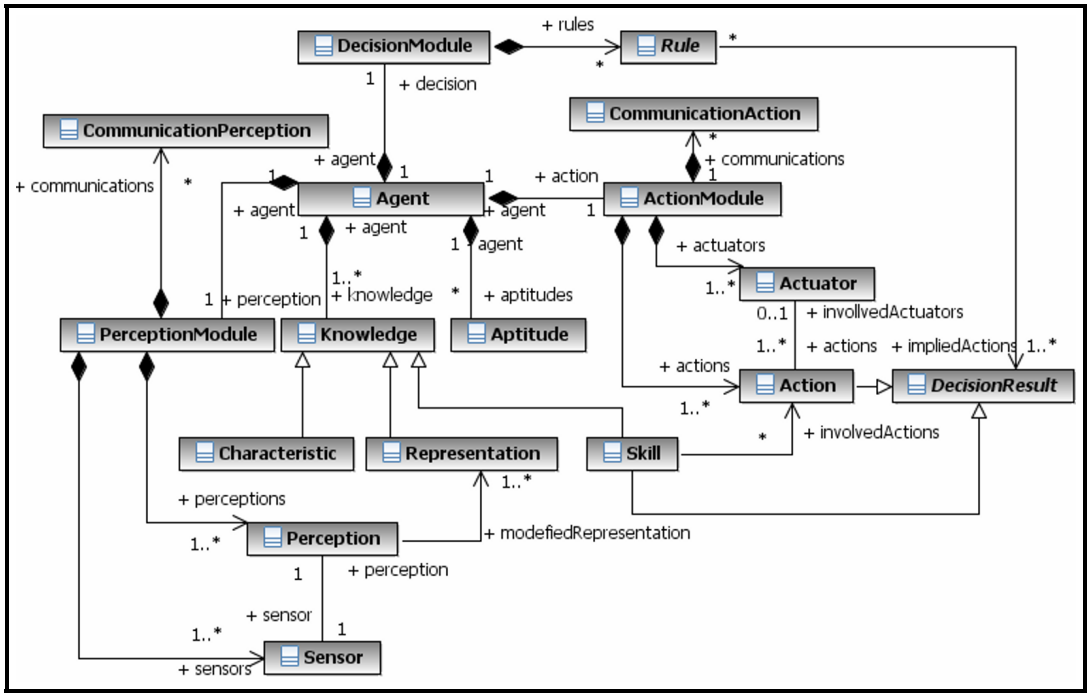


Fig. 10.3: The ADELFE metamodel.

concentrating on cooperative behavior. ADELFE is based on adaptive MAS (AMAS) and therefore a great effort is done in order to study all feasible situations that could enable or inhibit the cooperation among agents. The cognitive and behavioral representations of an agent are given in terms of its attitudes, skills and characteristics. The interaction among the intelligent entities is either realized through direct communication or via the environment. In version 2 of ADELFE, the developers take the ideas of MDD into consideration. The respective methodology is built upon a domain-specific modeling language for Adaptive Multi-Agent Systems called (AMAS-ML) as PIM and μADL (micro-Architecture Description Language) as PSM.

Main Concepts The metamodel of AMAS-ML is depicted in Fig. 10.3. The metamodel is centered around the concept of *Agent* that has perceptions (through the PerceptionModule) in forms of *CommunicationPerception*, *Sensor* and *Perception*. Moreover, the *Agent* has certain *Knowledge* available, which is either of the kind *Characteristics*, *Representations* and *Skills*, or *Aptitudes*. In order to behave in a reactive, social and proactive manner, the *Agent* uses *ActionModules* that are either *CommunicationActions* or simple *Actions*.

Methodology The development process of ADELFE is divided into five phases: In the *preliminary requirements phase*, the user requirements are defined and validated. The *final requirements phase* characterizes the environment and determines the use case. Furthermore, the developed prototype is elaborated and validated. In the *analysis phase*, the domain is analyzed, the agents are identified, and the interaction between the involved entities is studied. Followed by the *design phase* in which the detailed architecture and the MAS model is studied. Finally, the *implementation phase* deals with the model-driven transfer of the design into code.

		Weight	Scale
Main Concepts	Agents	2	2
	Organizations	0	0
	Goals	0	0
	Plans	1	0.66
	Environment	1	0.66
	AIP	2	1.33
Methodology	Whole Lifecycle	2	2
	Deployment	0	0
	Model-driven	0	0
	Modeling Language	0	0
Tool Support	Viewpoints	1	1
	Validation	0	0
	Code Generation	0	0
	Generic Modeling Environment	1	1
Semantics	Static Semantics	0	0
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	2	2
	Platform Support	1	1
	Interoperability	0	0
		13	11.66

Tab. 10.7: A summary on ADELFE's characteristics.

Tool Support UML diagrams are used by ALDELFE for visualization purposes, i.e., UML sequence diagrams and AUML are used, for instance, to model the agent interaction. Other diagrams of ADELFE, like the agent diagram or the behavioral rules diagrams, are oriented toward UML class diagrams.

Formal Semantics -

Interoperability A model-driven approach is presented in (Rougemaille et al.; 2008) that transforms the design made with ADELFE v.2 methodology into the platform-specific μADL. The architectural style of the micro-component assembly and a special tool called MAY (Make your Agents Yourself)—a kind of abstract agent machine—is generated. Both can be used by the developer as an abstract layer to implement the behavior of the generated agents. However, from the behavioral rules expressed in the design phase with AMAS-ML, only code skeleton is automatically produced, which needs considerable manual effort to produce an executable implementation.

Table 10.7 summarizes the core characteristics and features of ADELFE in accordance to the evaluation framework proposed in Section 10.1. The main benefit of ADELFE is the automatic code generation, even if only skeletons are produced for agents' behaviors. The main drawbacks are (i) the missing semantics, (ii) the weak tool support, and (iii) the missing design patterns for modeling organizations. The overall score of ADELFE amounts to 11.66.

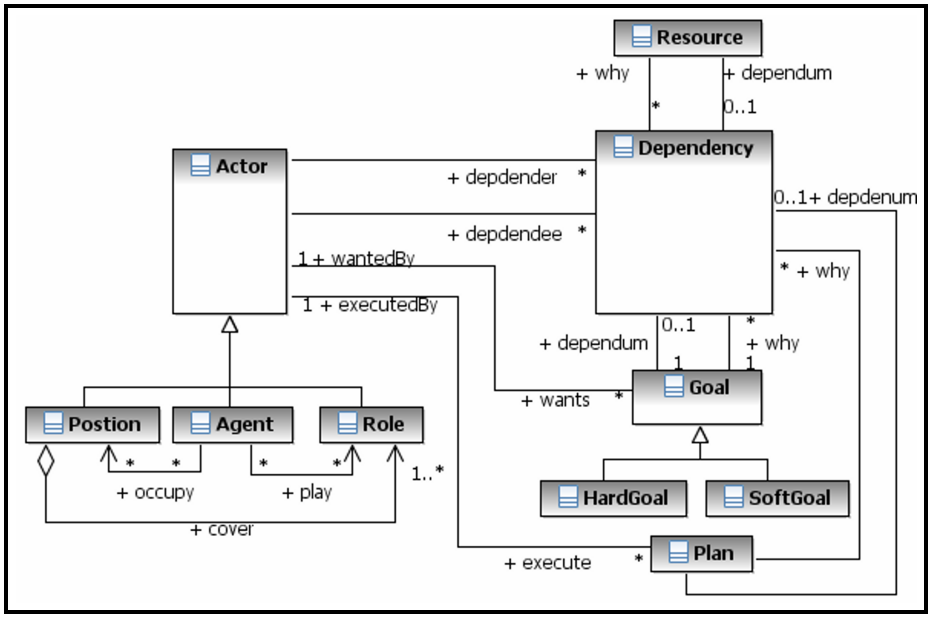


Fig. 10.4: The Tropos metamodel related to the actor diagram, in accordance to (Susi et al.; 2005).

10.2.7 Tropos

Tropos (Bresciani et al.; 2004) is an agent-oriented methodology based on the concepts of actor and goal. It bases on the i* framework (Yu; 1995), which was originally developed for modeling and reasoning about organizational environments and their information systems.

Main Concepts The concepts related to the Tropos actor diagram are depicted in Fig. 10.4. The core concept of the related metamodel is the concept of *Actor* and its specializations *Position*, *Agent* and *Role*. A *Position* covers at least one *Role* and an *Agent* may play *Roles* and may occupy *Positions*. An *Agent* may in addition want *Goals* to achieve that are either *HardGoals* or *SoftGoals*. A *Dependency* defines a relationship between a depender (i.e. *Actor*), dependee (i.e. *Actor*) and a dependum (i.e. *Goal*, *Plan* or *Resource*).

The concepts related to the Tropos goal diagram are depicted in Fig. 10.6. The core concept of the related metamodel is the concept of a *Goal* that can be analyzed by an *Actor* using *Means-End Analysis*, *Boolean Decomposition* and *Contribution*. The *Means-End Analysis* defines a relationship among the *Actor* (whose point of view is presented in the analysis), a *Goal* (the end) and feasible *Plans* and *Resources* (the means). The *Contribution* defines a relationship among the *Actor* (whose point of view is presented in the analysis) and two *Goals* to identify *Goals* that can positively or negatively affect other goals. *Decomposition* defines a relationship for a generic boolean decomposition of a root *Goal* into subgoals, either in an AND- or OR manner through the attribute type in a *Boolean Decomposition*.

The modeling of social groupings like organizations nor adequate concepts to describe the agent's environment are neither provided by the actor nor goal diagram.

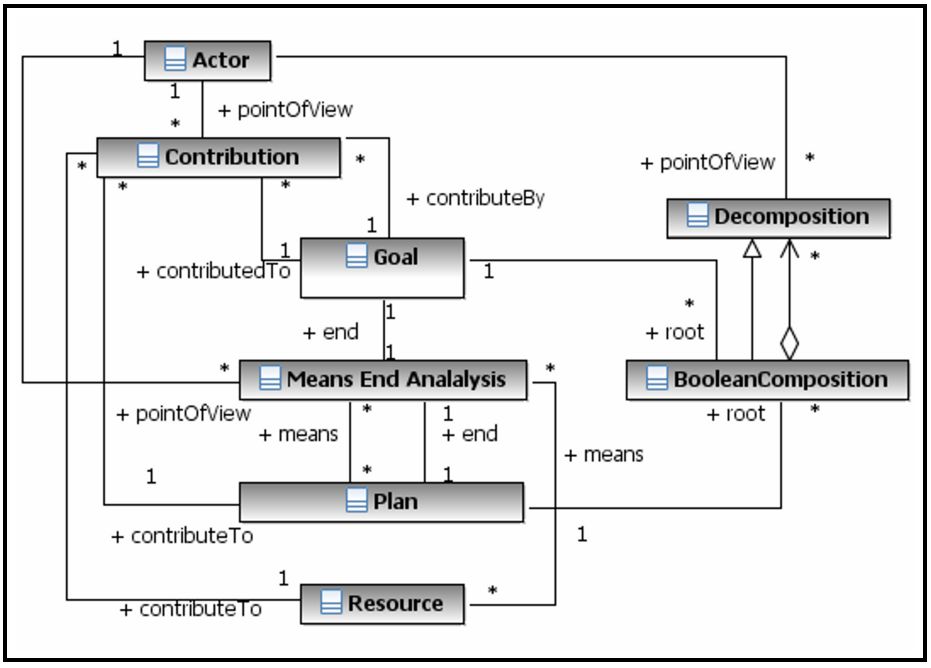


Fig. 10.5: The Tropos metamodel related to the goal diagram, in accordance to (Susi et al.; 2005).

Methodology The Tropos methodology consists of five phases (Giorgini et al.; 2001). The *early requirements phase* deals with the problem understanding by studying the existing organizational setting. The output of this phase is an organizational model that defines relevant actors and their respective dependencies. The *late requirements phase* deals with describing how the system should be within its operational environment, along with relevant functions and qualities. The output of this phase is a (small) number of actors, which have a number of dependencies with actors in their environment. The *architectural design phase* deals with the definition of the system's global architecture, i.e. subsystems, interconnected through data and control flows. The *detailed design phase* deals with the definition of each architectural component in further detail in terms of inputs, outputs, control, and other relevant information. Tropos thereby adopts elements from AUML (see Section 10.2.1). Finally, the *implementation phase* deals with the implementation of the system mainly done through JADE.

Tool Support The modeling tool of the Tropos methodology is called TAOM4E[4]. It bases on EMF and GEF, however, only a fragment of the Tropos metamodel is implemented by TAOM4E. As an example, in TAOM4E, only the concept of Actor is implemented. Its specialization of Position, Agent and Role and their relationships cannot be applied to the design. In this respect, one of the main advantages of GMF in contrast to GEF is the model-driven creation of the graphical editor based on the underlying metamodel. Hence, the concepts of the metamodel do not have to be manually realized, rather they are implemented in an automatic manner based on the metamodel.

[4] http://sra.itc.it/tools/taom4e/

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	1 (roles)	0.66
	Goals	3	2
	Plans	2	1.33
	Environment	0	0
	AIP	2	1.33
Methodology	Whole Lifecycle	2	2
	Deployment	0	0
	Model-driven	1	1
	Modeling Language	0	0
Tool Support	Viewpoints	1	1
	Validation	1	2
	Code Generation	1	2
	Generic Modeling Environment	0.5	0.5
Semantics	Static Semantics	1	3
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	2	2
	Platform Support	2	2
	Interoperability	0	0
		22.5	24.83

Tab. 10.8: A summary on Tropos's characteristics.

Formal Semantics Tropos provides tool support for goal analysis (GR-Tool, see (Giorgini et al.; 2005)) and model checking (T-Tool, see (Fuxman et al.; 2001)). The eCAT tool (Nguyen et al.; 2008) generates test case skeletons from goal analysis diagrams produced using TAOM4E.

Interoperability For the purpose of code generation, TAOM4E includes a suite of code generators in accordance to the MDA philosophy (cf. (Perini and Susi; 2005)). These generators are UML2JADE, t2x and Tropos2UML that produce code skeletons for either JADE or Jadex agents. However, in contrast to the DSML4MAS approach, Tropos utilizes UML as middle layer between Tropos itself and the execution engines of JADE and Jadex. The model transformation approach (Penserini et al.; 2007) consequently consists of two transformations, i.e. (i) Tropos2UML transformation that generates UML activities diagrams from Tropos goal models and (ii) UML2JADE transformation that generates JADE agent code skeletons from UML activity and sequence diagrams. However, as previously mentioned, UML lacks expressiveness to adequately model agent systems. Consequently, the expressiveness is lost during the two model transformations.

Table 10.8 summarizes the core characteristics of Tropos. It provides a clear process model covering all necessary steps to develop MASs in combination with code generators producing a skeleton JADE implementation. The design made with Tropos can be evaluated by means of a static semantics and testing facilities. The main drawback of Tropos is the lack of support regarding organizational modeling. The overall score of Tropos amounts to 24.83.

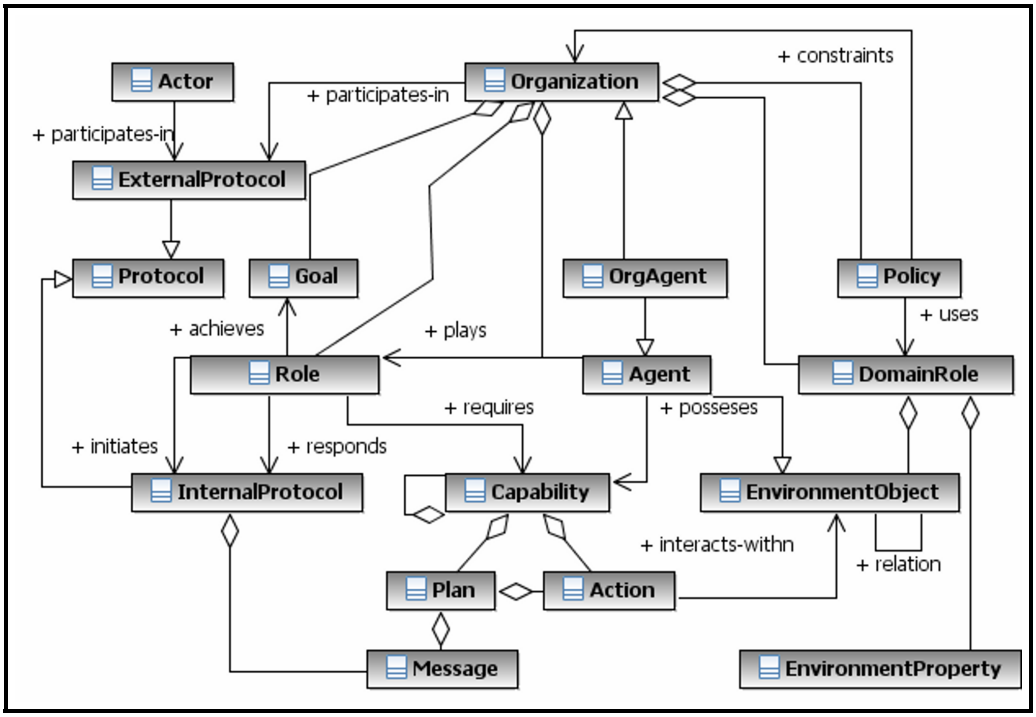


Fig. 10.6: The O-MaSE metamodel, in accordance to (DeLoach and Valenzuela; 2006).

10.2.8 Organization-based Multiagent System Engineering

The Organization-based Multiagent System Engineering (O-MaSE, (García-Ojeda et al.; 2007)) methodology was originally developed to improve the MaSE (Multiagent Systems Engineering, (Wood and DeLoach; 2001)) methodology and its limitations. O-MaSE aims at allowing MAS designers to construct agent-oriented methodologies based on method fragments conforming to a common metamodel.

Main Concepts The metamodel of O-MaSE is centered around the concept of *Organization*, which is composed of *Goals*, *Roles*, *Agents*, *Domain Model*, and *Policies*. A *Goal* defines the overall function of the *Organization* and a *Role* defines a position within an *Organization* whose behavior is expected to achieve particular *Goal(s)*. An *Agent* owns certain *Capabilities* to act in the environment and plays *Roles* inside *Organizations*. *Capabilities* are algorithms or plans that capture algorithms used by *Agents* to carry out specific tasks, while *Actions* are used to interact with environmental objects. The environment is modeled in terms of a *Domain Model*, which defines the types of objects in the environment and the relations between them. *Policies* normally constraint *Organizations* to behave in certain situations in a pre-defined manner. Finally, *Protocols* define how to interact either internally or externally.

Methodology The O-MaSE methodology consists of three phases: In the *requirement engineering phase* the goal hierarchies are defined, the *analysis phase* includes modeling the (i) organizational interfaces, (ii) roles, and (iii) domain using UML. Finally, in the *design phase*, the system designers specifies the agent, protocol, plan, policies, capabilities, action and service models. O-MaSE assumes an iterative cycle across all three phases with the intent

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	2	1.33
	Goals	2	1.33
	Plans	2	1.33
	Environment	2	1.33
	AIP	2	1.33
Methodology	Whole Lifecycle	2	2
	Deployment	0	0
	Model-driven	0	0
	Modeling Language	0	0
Tool Support	Viewpoints	1	1
	Validation	1	2
	Code Generation	1	2
	Generic Modeling Environment	1	1
Semantics	Static Semantics	1	3
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	3	3
	Platform Support	1	1
	Interoperability	0	0
		26	24.66

Tab. 10.9: A summary on O-MaSE's characteristics.

that successive iterations will add details to the models until a complete design is produced (García-Ojeda et al.; 2007).

Tool Support The O-MaSE methodology is supported by the agentTool III (aT^3) development environment, which is based on the first two versions of agentTool supporting the original MaSE methodology. agentTool III was developed using Java and is provided as an Eclipse plug-in.

Formal Semantics As part of aT^3, the designer has the opportunity to apply a verification framework to check the consistency between the different models defined (cf. (Garcia-Ojeda et al.; 2009)). The verification framework consists of a set of static rules that can be turned on and off. However, in its current version, aT^3 does not support automatic verification of the correct implementation of protocols within plans (DeLoach et al.; 2009).

Interoperability Like the verification framework, aT^3 also includes code generators for one agent-based execution platform. In the aT^3 case, the authors of (Garcia-Ojeda et al.; 2009) claim to have a 100% code generation of JADE code. In (Radziah et al.; 2006), conceptual mappings between MaSE and Jadex have been explored based on concetual mappings.

Table 10.9 summarizes the evaluation of O-MaSE in terms of our evaluation framework. O-MaSE offers a (i) comprehensive vocabulary to model MASs, (ii) static semantics integrated in the provided modeling tool, and (iii) complete automatic model transformation to JADE. The overall score of O-MaSE is 24.66.

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	2	1.33
	Goals	3	2
	Plans	0	0
	Environment	2	1.33
	AIP	2	1.33
Methodology	Whole Lifecycle	2	2
	Deployment	1	1
	Model-driven	0	0
	Modeling Language	1	2
Tool Support	Viewpoints	1	1
	Validation	0	0
	Code Generation	1	2
	Generic Modeling Environment	0.5	0.5
Semantics	Static Semantics	0	0
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	3	3
	Platform Support	1	1
	Interoperability	0	0
		22.5	21.5

Tab. 10.10: A summary on INGENIAS's characteristics.

10.2.9 INGENIAS

INGENIAS (Pavón and Jorge; 2003) is a methodology for specifying MASs on a platform independent level. It builds upon the MESSAGE methodology (Caire et al.; 2002) by proposing a notation for the specification of MAS by extending UML with agent-related concepts such as agent, organization, role, goals and tasks.

Main Concepts The metamodel of INGENIAS (Pavón et al.; 2005), which is based on the Ecore meta-metamodel is split up in several metamodels each covering important aspects to model MASs. In particular, concepts for modeling agents, organizations of agents, the environment in which agents and organizations are situated, goals and tasks and, finally, interactions in which two or more agents are interacting. The agent viewpoint describes the agentŠs behavior in terms of the agent's mental state, a set of goals and beliefs, as well as, the roles the agent is able to play. However, a vocabulary for describing how an agent achieves a certain goal is not given. The goals and tasks viewpoint describes the relationship between goals and task, i.e., goals for agents could be further refined into simpler goals up to a level where it is possible to identify specific tasks to satisfy them. For modeling interactions, AUML is utilized.

Methodology The INGENIAS methodology includes the core phases from *requirements* to *testing*. In (García-Magariño et al.; 2009), a similar approach to DSML4MAS is presented, i.e. the authors present a model-driven development process that allows to automatically generate role definitions, interactions from the agent workflow, and agent deployments.

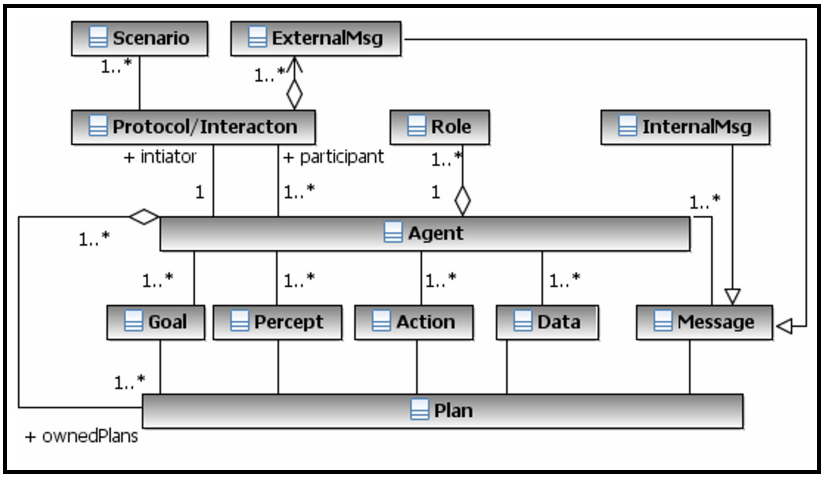


Fig. 10.7: The Prometheus metamodel (part 2), in accordance to (Dam et al.; 2006). For association ends that do not have a multiplicity, it should be interpreted that their multiplicity is zero or many (0..*).

Tool Support The INGENIAS methodology is supported by the INGENIAS Development Kit (IDK) (Gomez-Sanz et al.; 2008a), which is a graphical modeling tool based on Java. It facilitates means to transform the design into code, however, IDK does not provide validation and verification mechanisms.

Formal Semantics -

Interoperability INGENIAS adopts the ideas of MDD and, thereby, provides a general process for transforming the design into code. In particular, the code generation is targeting JADE. Moreover, the INGENIAS Code Uploader extension supports refactoring of JADE code. In contrast to DSML4MAS, INGENIAS only provide model transformations to a single agent platform. In (Gascue and Fernández-Caballero; 2009), the authors describe how to combine and integrate INGENIAS and Prometheus based on conceptual mappings.

Table 10.10 depicts an overview on the basic characteristics of INGENIAS. Like DSML4MAS, INGENIAS provides a model transformation to JADE that allows generating most of the code fragments. The major shortcoming of INGENIAS is the lack of formal semantics. The overall score of INGENIAS is 21.5.

10.2.10 Prometheus

In accordance to Padgham and Winikoff (2002a), Prometheus is an AOSE methodology that is detailed and complete in the sense of covering all activities required in developing intelligent agent systems. Furthermore, the authors argue that it is a generic modeling language that can be used for any MAS architecture and environment, even if it has mainly been used for designing BDI agents.

Main Concepts The metamodel of Prometheus is divided into two parts. Fig. 10.8 depicts the concepts needed within the system specification phase, whereas Fig. 10.7 depicts the concepts

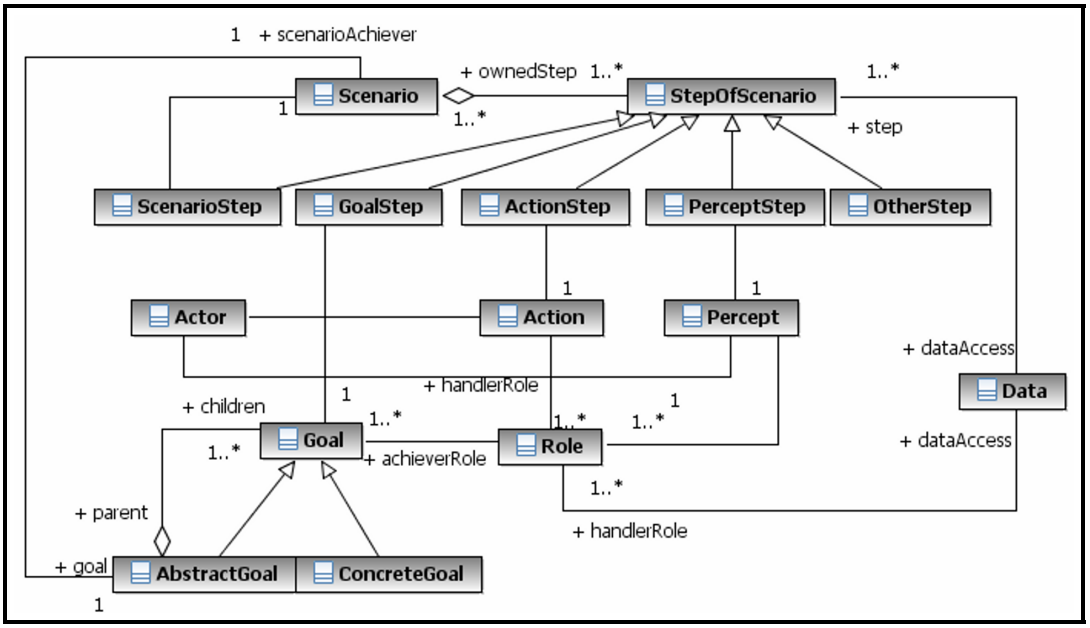


Fig. 10.8: The Prometheus metamodel (part 1), in accordance to (Dam et al.; 2006). For association ends that do not have a multiplicity, it should be interpreted that their multiplicity is zero or many (0..*).

needed within the detailed design phase. Prometheus distinguishes between two sorts of Goals, namely AbstractGoal and ConcreteGoal (see Fig. 10.8). The main difference between both is that the AbstractGoal has children, which are again Goals. A Scenario consists of a set of concepts called StepOfScenario, which is the parent class for the specializations ScenarioStep, GoalStep, ActionStep and PerceptStep. A Role could achieve Goals, has access to Data and has to handle Percepts. The core concept of the detailed design phase is the concept of an Agent (see Fig. 10.7). An Agent may have access to a set of Capabilities, owns a set of Plans, and plays Roles. Additionally, the Agent may either be participant or initiator of a Protocol/Interaction, which again refers to ExternalMsg that is like the InternalMsg a specialization of Message. Interaction Protocols are realized through AUML. Prometheus does not support the modeling of organizational structures.

Methodology The Prometheus methodology consists of three phases (Padgham and Winikoff; 2002b). The *system specification phase* is used for specifying goals and scenarios. The system's interface to its environment is described in terms of actions, percepts and external data. Furthermore, functionalities are defined in this phase. The *architectural design phase* is used for identifying agent types. The system's overall structure is captured in a system overview diagram. Scenarios are developed into interaction protocols. Finally, the *detailed design phase* is used for developing the details of each agent's internals that are defined in terms of capabilities, data and events.

Tool Support The Prometheus Design Tool (PDT[5]) (Thangarajah et al.; 2005) offers diagrams for the high-level analysis of a system, the refinement with interaction diagrams with AUML,

[5] http://www.cs.rmit.edu.au/agents/pdt/

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	1	0.66
	Goals	3	2
	Plans	1	0.66
	Environment	1	0.66
	AIP	2	1.33
Methodology	Whole Lifecycle	2	2
	Deployment	0	0
	Model-driven	0	0
	Modeling Language	2	2
Tool Support	Viewpoints	1	1
	Validation	1	2
	Code Generation	1	2
	Generic Modeling Environment	0	0
Semantics	Static Semantics	1	3
	Dynamic Semantics	0	0
Interoperability	Executable Implementation	2	2
	Platform Support	1	1
	Interoperability	0	0
		22	23.33

Tab. 10.11: A summary on Prometheus's characteristics.

and the specification of processes. PDT contains a cross checking tool that detects problems like inconsistency checking, identification of dangling model elements, type checking, etc. Moreover, PDT provides code generation for JACK. It seems that PDT was implemented as a usual Java Swing application. There also exists a plug-in for the Eclipse platform, but the integration seems to be rather weak.

Formal Semantics As aforementioned, PDT includes tools for verifying and validating the design made based on the metamodel. These consistency rules include (i) reference to non-existent entities, (ii) internal design consistency, (iii) errors such as incorrect relationships between entity types, and (iv) violations of interface declarations (Padgham et al.; 2007b). Similar to our approach, OCL has been used to specify these consistency rules (see (Dam et al.; 2006) for details).

Interoperability PDT provides a code generation feature generating skeleton code that can be imported into the JACK IDE. In (Jayatilleke et al.; 2006), the CAFnE toolkit is presented, which extends Prometheus in terms of automatically producing an executable implementation.

Table 10.11 summarizes the characteristics of Prometheus. It provides a rich set of concepts to model MAS and integrates these into the graphical editing framework. The static semantics are used to validate the models at design time. The code generation is based on MDD and allows producing JACK code that needs to be manually completed. The overall score of Prometheus is 23.33.

10.3 DSML4MAS and State of the Art

In this section, we directly compare DSML4MAS with the modeling techniques presented in the previous section. The evaluation framework proposed in Section 10.1 gives us again the base for performing this evaluation.

10.3.1 Evaluation of DSML4MAS

To directly compare DSML4MAS with the state of the art on AOSE design frameworks, we firstly need to examine how well DSML4MAS supports the main criteria of our evaluation framework, i.e. *main concepts*, *methodology*, *tool support*, *semantics*, and *interoperability*.

Main Concepts DSML4MAS provides a detailed vocabulary to design MASs in an abstract (i.e. platform-independent) manner. In particular, the modeling of the following core concepts is supported:

- agents in terms of (i) knowledge (i.e. beliefs) used for reasoning purposes, (ii) plans used to act in a proactive, reactive, autonomous and social manner, and (iii) capabilities.
- organizations in terms of (i) interactions used to delegated members, and (ii) collaborations to define in which manner domain roles and actors are bound.
- interactions and protocols in terms of how abstract entities (i.e. actors), potentially representing more the one concrete agent instance, are exchanging messages.
- environments in terms of objects and service located outside and/or inside the MAS.
- plans in terms of complex workflow-like process structures.

In the current version of PIM4AGENTS goals can only be indirectly expressed through the concept of task. However, the next version of PIM4AGENTS will include a more precise goal modeling aspect as further viewpoint.

Methodology DSML4MAS covers the whole lifecycle from analysis to deployment and implementation. Parts of the phases can be semi-automatically produced. In particular, the model transformation from agent interaction protocols to internal behaviors is part of this model-driven methodology. Apart from the methodology, DSML4MAS can be used as simple modeling language.

Tool Support The DSML4MAS Development Environment is based on GMF and integrates the model transformations as well as the statics semantics of DSML4MAS expressed with OCL. Mechanisms for testing have not been yet integrated. The static and dynamic semantics of DSML4MAS is defined using Object-Z, the static semantics have been manually transformed to OCL.

Interoperability The model transformations from PIM4AGENTS to the metamodels of JACK and JADE produce nearly executable code. In complex scenarios, the system designer needs to manually add and refine the generated code. The model transformation between SoaML and PIM4AGENTS furthers the integration of agents into service-oriented environments and thus enables interoperability between both paradigms.

Table 10.12 depicts an overview on the basic characteristics of DSML4MAS. The overall score of DSML4MAS is 35.

		Weight	Scale
Main Concepts	Agents	3	3
	Organizations	3	2
	Goals	1	0.66
	Plans	3	2
	Environment	2	1.33
	AIP	3	2
Methodology	Whole Lifecycle	2	2
	Deployment	1	1
	Model-driven	1	1
	Modeling Language	1	2
Tool Support	Viewpoints	1	1
	Validation	1	2
	Code Generation	1	2
	Generic Modeling Environment	1	1
Semantics	Static Semantics	1	3
	Dynamic Semantics	1	2
Interoperability	Executable Implementation	3	3
	Platform Support	2	2
	Interoperability	2	2
		33	35

Tab. 10.12: A summary on DSML4MAS's characteristics.

10.3.2 Comparison with State of the Art

Based on the evaluation in the previous section, we now directly compare DSML4MAS and the modeling approaches discussed in Section 10.2. The comparison is structured in accordance to the main criteria of the evaluation framework.

10.3.2.1 Main Concepts

As depicted in Fig. 10.9, with respect to the main agent-based concepts, DSML4MAS scores best, followed by the modeling language of AML and the methodologies O-MaSE and INGENIAS. The fewest core concepts are provided by PASSI, AUML, and AORML. The high score of DSML4MAS mainly results from the fact that agents, organizational structures, plans and interaction protocols are well supported. Only the support of goals needs to be further improved.

10.3.2.2 Methodology

As depicted in Fig. 10.9, DSML4MAS offers the best methodology support. Apart from the model-driven support of the whole lifecycle, the option to either apply the DSML4MAS methodology process or to use it as pure modeling language has certainly advantages over other approach that exclusively support the one or the other option. The next best approaches are INGENIAS and Prometheus.

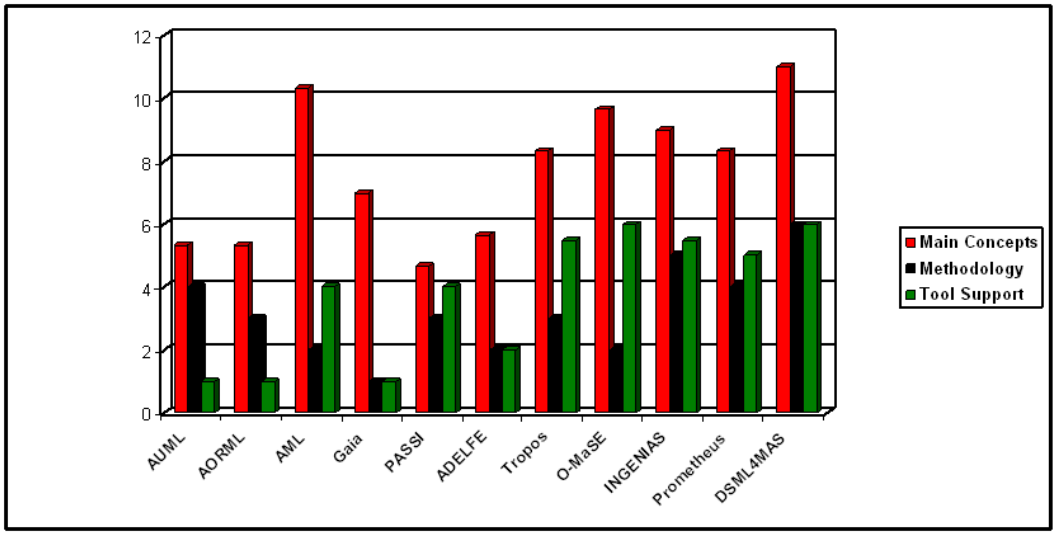


Fig. 10.9: Bar chart on the criteria *main concept*, *methodology*, and *tool support* of the different AOSE approaches.

10.3.2.3 Tool Support

The support of adequate tools differs among the evaluated approaches (cf. Fig. 10.9). Some of them nearly offer no support at all (e.g. AUML, AORML, Gaia), while others allow the graphical editing and validation of the design. Together with O-MaSE, DSML4MAS scores best, followed by INGENIAS, Tropos, and Prometheus. In the case of DSML4MAS, the generic development environment based on GMF thereby allows to make use of any editing capability of Eclipse and supports the integration of any add-on (i.e. model transformation, code generation), which is based on Ecore. In contrast, other approaches use standard GUI development languages like Java, where any of these editing capabilities needs to be manually implemented.

10.3.2.4 Semantics

Only few AOSE design approaches support the full range of a semantic specification (cf. Fig. 10.10). Approaches like Tropos, Prometheus, and O-MaSE only provide a static semantics, which is integrated into the graphical IDE to validate the correctness of the created model at design time. In contrast, Gaia and DSML4MAS provide a dynamic semantic, which is the base for performing model checking of the design.

10.3.2.5 Interoperability

The main differences between DSML4MAS and the other agent-based design approaches can be determined in the area of *interoperability* (see Fig. 10.10). The reason is that DSML4MAS offers an executable implementation for two standard AOPLs. The code generation is thereby automatic with only minor human intervention. Other approaches (e.g. Tropos, Prometheus) only support a

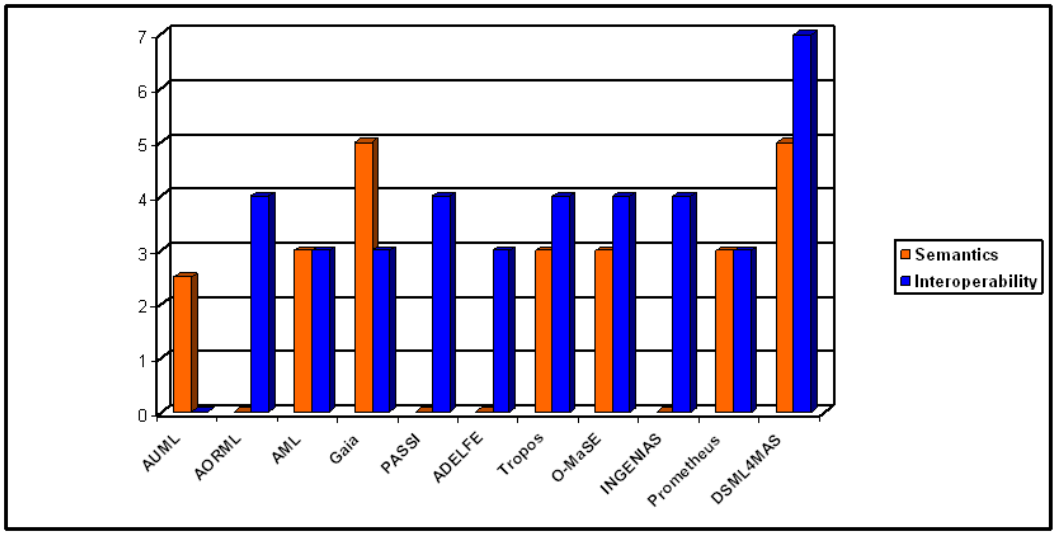


Fig. 10.10: Bar chart on the criteria *semantics* and *interoperability* of the different AOSE approaches.

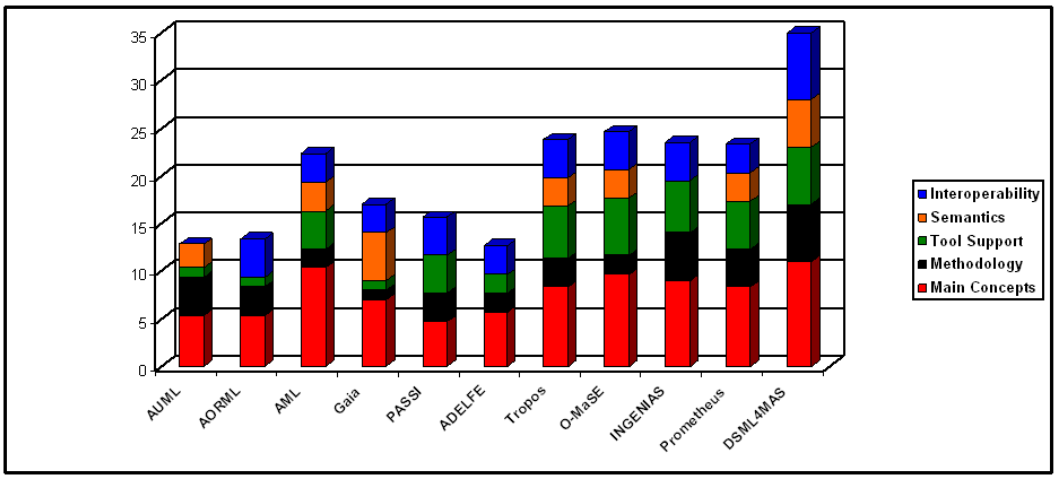


Fig. 10.11: Bar chart on the overall support of the different AOSE approaches.

single execution platform and the underlying model transformations only produce skeleton code that needs to be manually completed.

Beside code generation issues, only AORML supports the integration of other, perhaps more business-oriented, languages into their agent-based framework. The other approaches assume that the complete design is done with the own language, which is especially not the case in more business-oriented scenarios.

10.3.2.6 Overall

Fig. 10.11 presents the overall result when combining the scores of each criteria of our evaluation framework. As DSML4MAS performs best in each single category, it is not astonishing that DSML4MAS performs also best in the overall evaluation. The DSML4MAS framework achieves 95% of the maximal scores to obtain (i.e. 37). The second best AOSE design approach is Tropos followed by O-MaSE. The worst performing approaches are ADELFE, AUML and AORML.

10.4 Bottom Line

In this chapter, the DSML4MAS framework is compared with the state of the art on AOSE design approaches, i.e., pure modeling languages and methodologies. For this purpose, we defined an evaluation framework focusing on the key areas of *main concepts*, *methodology*, *tool support*, *formal semantics* and *interoperability*. The most well-known agent-based design approaches and DSML4MAS are then evaluated in accordance to these criteria to have a common foundation for comparison.

This comparison showed that DSML4MAS performs best in each category of the evaluation framework. Consequently, DSML4MAS offers the richest set of modeling concepts, a complete methodology process, the best agent-based tool support, static and dynamic semantics, and a well explored MDD process to enhance the interoperability between DSML4MAS and other languages. A second interesting result is that there is no single second best design approach. AML and O-MaSE, for instance, both provide a rich set of modeling concepts, their methodology support is however rather limited. In contrast, INGENIAS supports all phases of the development process, but the support of modeling concepts lacks.

Part V

Conclusion & Further Work

11. Conclusion

In recent years, agent-oriented software engineering (AOSE) as a new software engineering discipline has emerged. The overall objective of AOSE is to provide approaches for the development of open, distributed, robust and intelligent systems. A particular interest is thereby the development of agent-based modeling frameworks that provide the software engineer with adequate means to support the development of MASs. This thesis and the presented DSML4MAS language contributes to the development of agent-based modeling approaches.

Structure of this Chapter In Section 11.1, this final chapter discusses the main contributions of this dissertation project and points areas for future research in Section 11.2.

11.1 Contributions

This dissertation focused on the development of a platform independent modeling language for the domain of MASs called DSML4MAS. The motivation for this domain-specific modeling language came forth from the need of AOSE to:

- specify a unified metamodel to define basic constructs to model agent, organization, interaction, behavior, environment and mental concepts
- provide code generators that allow transferring the abstract design into an executable implementation
- define a formal semantics and concrete syntax
- define a mature methodology
- enhance the interoperability with other software engineering approaches

The approach presented in this dissertation addresses these needs through a platform-independent modeling language for the domain of MASs. The remainder of this section lists the main scientific contributions of this thesis.

11.1.1 Core Contributions

Platform-Independent Vocabulary for Designing Multiagent Systems In order to define MASs in a platform independent manner, the core concepts of MASs were identified to define agent-based systems in an abstract manner. These concepts were defined through the so-called platform-independent metamodel for MAS (PIM4AGENTS). The concepts are, on the one hand, sufficient enough to generate executable code and, on the other hand, platform-independent as code generators for two platforms exist. To separate concerns, different viewpoints were indicated, each of them focus on a particular aspect of MASs. These viewpoints comprises the multiagent system, agent, organization, role, interaction, behavior, environment, and deployment viewpoint.

Formal Semantics Apart from the syntax formed through the PIM4AGENTS metamodel, the semantics of DSML4MAS are formalized through the specification language of Object-Z. For this purpose, for each concepts, the static and dynamic (if applicable) semantics are characterized through invariants. The static semantics have been further manually translated to OCL, which allow the integration into the graphical development environment. This gives the application developers the opportunity to test the produced models at design time and to correct errors if needed before code generators are applied.

Generic Code Generation The code generation within DSML4MAS is achieved through model transformations in accordance to the principles of MDD. Based on code generation templates, the designed MAS is transformed to executable code that may optionally be merged with manually written code. We developed model transformations to the agent-based execution platforms of JACK and JADE. The implementation of model-to-model transformations is done using ATL that transfers the PIM4AGENTS models to a model conforming to the metamodel of JACK (i.e. JackMM) or JADE (i.e. JadeMM). On this models, a model-to-text transformation is applied based on MOFScript that finally transfers models conforming to either JackMM or JadeMM to code that can be compiled and executed. The two code generators allow the application developer to (i) define the MASs on an abstract level by utilizing the vocabulary provided by PIM4AGENTS and (ii) transform it either to JACK or JADE depending on the requirements he/she has.

Model-Driven Integration of Service-Oriented Architectures into Multiagent Systems As MASs do not exist in pure isolation, it is important to provide means for the integration with existing standard software engineering approaches. In the DSML4MAS architecture, this integration has been exemplarily realized by linking MASs and service-oriented architectures (SOAs), which are nowadays one of the favorite approaches to realize distributed software landscapes. The integration has been done in a model-driven manner by implementing a model transformation between the service-oriented architecture modeling language (SoaML) and PIM4AGENTS. SoaML was proposed by the OMG as standard modeling language for SOAs and is thus a good candidate to establish this model-driven relationship as industry will more and more base their software systems on this standard. An automatic transformation to DSML4MAS might further increase the acceptance of MASs in industry.

Integrated Development Environment The DSML4MAS Development Environment is a model-driven framework for the development of MASs. It bases on the abstract syntax of DSML4MAS, which is given by PIM4AGENTS. The functionality of DDE encompasses (i) the platform independent specification of MASs, (ii) model validation, (iii) model transformations and code generation, and (iv) execution of generated source code. The code generation facilities support the agent execution environments JACK and JADE. The features of DDE include (i) reduction of complexity by separation into diagrams, (ii) model validation by integrating the static semantics described by OCL, (iii) reusable components, and (iv) extensibility.

Model-Driven Methodology This thesis defines a (semi-) automatic model-driven methodology process including endogenous—on the PIM level—and vertical transformations between the PIM and PSM levels. The endogenous transformation supports the (semi-) automatic transfer of agent-based interaction protocols to (i) internal behaviors used by agents to execute the global interaction descriptions and (ii) MAS and role views. The overall process has been formalized using the Eclipse Process Framework. This specification guides the

MAS developer through the different phases, beginning with the analysis phase, to the architectural specification and detailed design phase up to the deployment and implementation phase on the PSM level. Apart from the model-driven methodology process, experienced users may apply DSML4MAS as pure modeling language, without using the endogenous transformation.

Theoretical Evaluation In order to evaluate DSML4MAS from a scientific perspective, a comprehensive evaluation framework has been proposed. Ten of the most known agent design approaches and frameworks have been presented and evaluated in accordance to the proposed evaluation framework. The comparison between them and DSML4MAS demonstrates that DSML4MAS offers a rich set of modeling concepts, as well as, a formal semantics. Furthermore, tool support is provided that integrates validation facilities and code generators. A model-driven methodology process, as well as, the integration with standard software approaches like SOAs is given. Compared to other agent-based design approaches, DSML4MAS scores best in accordance to our evaluation framework.

Practical Evaluation DSML4MAS has firstly been evaluated in a number of European (e.g. SHAPE[1], Coin[2]) and national (e.g. MODEST[3]) projects. Secondly, a list of examples like the conference management system have been defined to evaluate the usefulness of DSML4MAS. Finally, DSML4MAS has been applied to two industrial use cases to evaluate its practicability under real-word conditions in industrial settings.

Standardization Activities around DSML4MAS An important means of disseminating the ideas of DSML4MAS in the AOSE community was the participation in international standardizations activities. Our active involvement in the standardization activities around the OMG's open source projects Service-oriented architecture Modeling Language (SoaML) and Agent Metamodel and Profile (AMP) allows us to (i) bring the DSML4MAS ideas into standardized approaches for SOA and MASs and (ii) gather valuable feedback that was taken into account when iteratively improving DSML4MAS and its features.

Integration of Semantic Web services to express the dynamics of the system How to express the dynamics of the MAS at design time is still considered as open issue. For DSML4MAS, two mechanisms are provided to solve this issue. Firstly, tasks in a plan are defined that allow to specify the dynamic binding of agent instance to domain roles, where the actual binding is done on the PSM level. Secondly, by integrating Semantic Web services and match making facilities, we enable the search for feasible services at run-time in accordance to certain pre- and post-conditions specified at design time. This allows the dynamic model-driven integration of services. For details on the integration of Semantic Web services, we refer to (Hahn et al.; 2008a,b).

11.1.2 Core Characteristics of DSML4MAS

Several requirements are necessary to make a DSL successful in terms of usage for the particular domain. Basically, the requirements and principles for DSLs do not differ to requirements for designing general purpose languages (Deursen and Klint; 1997) or programming languages (Hoare; 1973), however, the weighting might be different. In the following, we review the basic

[1] http://www.shape-project.eu/
[2] http://www.coin-ip.eu/
[3] http://www-ags.dfki.uni-sb.de/ klusch/modest/index.html

requirements of DSLs proposed by Kolovos et al. in (Kolovos et al.; 2006) and related them to DSML4MAS.

Conformity The language constructs of any DSL must correspond to important domain concepts. The abstract syntax of DSML4MAS is defined through a metamodel called PIM4AGENTS that provides abstract agent-based constructs that correspond to the core building blocks of MAS (cf. Section 2.1.1). This allows modeling MASs in a comfortable manner, also for non experts on agent theory.

Orthogonality Each construct in DSML4MAS is used to represent exactly one distinct concept in the domain. The notation (i.e. concrete syntax), moreover, is defined in an unambiguous manner.

Supportability DDE provides tool support for typical model and program management, e.g., creating, deleting, editing, debugging, transforming. This is enabled through basing DDE on GMF, which naturally supports the domain expert with model management facilities like editing and deleting. Furthermore, as part of the IDE, model transformations to agent-based execution platforms are given.

Integrability The DSML4MAS language, in our case, can be used in the concert with other approaches and corresponding languages for software engineering. Semantic Web services and SOAs are two examples that are integrated into DSML4MAS in a model-driven manner, which makes the whole language applicable in other domains.

Longevity Any DSL should be used and useful for a non-trivial period of time in order to ensure tool support, and to make it possible to quantify to the DSL stakeholders the payoff obtained from using the DSL. There is, of course, an assumption with this requirement (and with the use of DSLs in general) that the domain under consideration persists for a sufficiently lengthy period of time to justify the cost of building a DSL and supporting tools. DSML4MAS provides constructs for the domain of MASs, which has grown over years and is thus a very stable area. If needed, DSML4MAS can easily be extended to support new concepts and views. This could easily be achieved by extending the PIM4AGENTS metamodel and its transformations to the supported execution environments.

Simplicity A language should, in general, be as simple as possible in order to express the concepts of interest and to support its users and stakeholders in their preferred ways of working. Through basing the development environment of DSML4MAS on Eclipse, which is one of the most used IDEs for developing software, most potential users and stakeholders need less time to get in touch with DSML4MAS and to use it in an efficient manner. This also allows non agent-specialists to use DSML4MAS in an intuitive manner. The static semantics allow then to validate the design made.

Scalability A language should provide constructs to help managing large-scale descriptions. This is of particular importance when building complex potentially distributed architectures. DSML4MAS and its constructs and the provided IDE support scalability. On the language side, this is mainly achieved through various abstraction levels, like the agent-based design and deployment. On the IDE side, features like zoom-in or zoom-out and the various diagrams that allow building the MAS from different viewpoints provide adequate mechanisms to build complex MASs.

Quality Any language in accordance to LDD shall provide general mechanisms for building quality systems. This may include (but is not limited to) language constructs for improving reliability (e.g., pre- and postconditions), security, safety, etc. The formally specified semantics support the stakeholders and users building qualitative software as the software's characteristics could easily be validated and checked to ensure that the generated design behaves in accordance to the requirements specified on more abstract levels.

11.2 Open Issues & Future Work

In future work, we would naturally like to extend DSML4MAS with respect to the following list of open issues.

Goal Modeling To achieve full pro-active behavior, the agents need to have goals that they can pursuit. In the current version of PIM4AGENTS, goals are represented implicitly inside the agent's plan. In order to provide full support, a new viewpoint has to be introduced, which allows both the abstract and concrete goal modeling. However, this viewpoint should be considered as optional, as not all of the supported agent-based platforms provide mechanisms to deliberate on goals. In (Madrigal-Mora et al.; 2008), an extension of PIM4AGENTS has been presented that allows goal modeling in DDE.

Norm Modeling Norms and institutions are considered as an important mechanism of the environment to regulate the overall behavior of the MASs. Previous research (e.g. (Hahn et al.; 2007a, 2006a, 2005)) in the Socionics initiative funded by the Deutsche Forschungsgemeinschaft lay the foundations for future research on how to bring norms and institutions into PIM4AGENTS. This enables restricting the run-time behavior of the autonomous entities in DSML4MAS at design time.

Model-Driven Model Checking The formal semantics of DSML4MAS in principle allows a model-checking on the PIM level. However, it would be very interesting to investigate a model-driven approach to transfer the created PIM4AGENTS models, as well as, the formal semantics to, for instance, timed automata. This would enable the model checking on the design in order to detect dead locks, etc.

Architecture-Driven Modernization To realize round-trip engineering, a future version of DSML4MAS will implement reverse engineering mechanisms in accordance to the Architecture-Driven Modernization (ADM) initiative (Newcomb; 2005). In particular, model transformations will be developed that transfer models on the PSM level including the agent-based execution platforms of JACK and JADE to PIM4AGENTS. These transformations offer developers a full round-trip engineering approach from DSML4MAS to JACK and JADE to DSML4MAS.

Bibliography

Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*, Cambridge University Press, New York, NY, USA.

Agostini, A. (2003). Cooperation = coordination + solvability, *Technical report*, ITC-IRST Trento.

Albayrak, S. and Wieczcorek, D. (1999). JIAC - a toolkit for telecommunication applications, *Proceedings of the Third International Workshop on Intelligent Agents for Telecommunication Applications (IATA '99)*, Springer Verlag, London, pp. 1–18.

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2003). *Web Services: Concepts, Architectures and Applications*, Springer Verlag, Berlin et al.

Amor, M., Fuentes, L. and Vallecillo, A. (2004). Bridging the gap between agent-oriented design and implementation using MDA, *in* J. Odell, P. Giorgini and J. P. Müller (eds), *Agent-Oriented Software Engineering (AOSE-2004)*, Vol. 3382 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 93–108.

Anastasakis, K., Bordbar, B., Georg, G. and Ray, I. (2007). UML2Alloy: A challenging model transformation, *in* G. Engels, B. Opdyke, D. C. Schmidt and F. Weil (eds), *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, Proceedings*, Vol. 4735 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 436–450.

Ashbacher, C. (2008). IBM rational unified process reference and certification guide solution designer, *Journal of Object Technology* **7**(6): 53–54.

Bachem, A., Hochstättler, W. and Malich, M. (1992). Simulated trading: A new approach for solving vehicle routing problems, *Technical Report 92.125*, Mathematisches Institut der Universität zu Köln.

Bachem, A., Hochstättler, W. and Malich, M. (1993). The simulated trading heuristic for solving vehicle routing problems, *Technical Report 93.139*, Mathematisches Institut der Universität zu Köln.

Barros, A., Dumas, M. and Oaks, P. (2005a). A critical overview of the web services choreography description language (WS-CDL), *BPTrends Newsletter 3*.

Barros, A. P., Dumas, M. and ter Hofstede, A. H. M. (2005b). Service interaction patterns, *in* W. M. P. van der Aalst, B. Benatallah, F. Casati and F. Curbera (eds), *Proceedings of the 3rd International Conference on Business Process Management, 5-8 September 2005, Nancy, France*, Vol. 3649, Springer Verlag, New York, pp. 302–318.

Bauer, B. and Odell, J. (2002). UML 2.0 and agents: How to build agent-based systems with the new UML standard, *Journal of Engineering Applications of Artificial Intelligence* **18**(2): 141–157.

Bauer, B. and Stiener, D. (1998). *MECCA-System Reference Model*, Siemens, Munich.

Bauer, B., Müller, J. P. and Odell, J. (2001). Agent UML: A formalism for specifying multiagent software systems, *in* P. Ciancarini and M. Wooldridge (eds), *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Limerick, Ireland, June 10, 2000, Revised Papers*, Vol. 1957 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 91–103.

Bell, M. (2008). *Service-Oriented Modeling: Service Analysis, Design, and Architecture*, Wiley Publishing.

Bellifemine, F. and Rimassa, G. (2001). Developing multi-agent systems with a FIPA-compliant agent framework, *Software Practical Experience* **31**(2): 103–128.

Ben-Ari, M. (2008). *Principles of the Spin Model Checker*, Springer Verlag, Berlin et al.

Benguria, G., Larrucea, X., Elvesæter, B., Neple, T., Beardsmore, A. and Friess, M. (2007). A platform independent model for service oriented architectures, *in* G. Doumeingts, J. Müller, G. Morel and B. Vallespir (eds), *Enterprise Interoperability New Challenges and Approaches*, Springer Verlag, London, pp. 23–32.

Bennett, K., Gold, N., Munro, M., Xu, J., Layzell, P., Nehandjiev, N., Budgen, D. and Brereton, P. (2002). Prototype implementations of an architectural model for service-based flexible software, *Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS-35), 7-10 January 2002*, IEEE Computer Society, Washington, DC, USA, p. 76.2.

Bentley, J. L. (1984). Programming perls, *Communications of the ACM* **27**(1): 12–13.

Bernardi, S., Donatelli, S. and Merseguer, J. (2002). From UML sequence diagrams and statecharts to analysable petri net models, *Proceedings of the 3rd international Workshop on Software and Performance (WOSP '02)*, ACM Press, New York, NY, USA, pp. 35–45.

Bernon, C., Camps, V., Gleizes, M.-P. and Picard, G. (2005a). Engineering adaptive multi-agent systems: The ADELFE methodology, *in* B. Henderson-Sellers and P. Giorgini (eds), *Agent-Oriented Methodologies*, Idea Group Pub, NY, USA, pp. 172–202.

Bernon, C., Cossentino, M., Gleizes, M.-P., Turci, P. and Zambonelli, F. (2005b). A study of some multi-agent meta-models, *in* J. Odell, P. Giorgini and J. Müller (eds), *Proceedings of the 5th International Workshop on Agent-Oriented Software Engineering (AOSE 2004)*, number 3382 in *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 62–77.

Bernon, C., Gleizes, M. P., Peyruqueou, S. and Picard, G. (2003). ADELFE: A methodology for adaptive multi-agent systems engineering, *Engineering Societies in the Agents World III (ESAW 2002)*, Vol. 2257 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 156–169.

Berre, A. J. (2008). UPMS - UML profile and metamodel for services - an emerging standard, *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC '08)*, IEEE Computer Society, Washington, DC, USA, p. xxx.

Bertolini, D., Delpero, L., Mylopoulos, J., Novikau, A., Orler, A., Penserini, L., Perini, A., Susi, A. and Tomasi, B. (2006). A Tropos model-driven development environment, *in* N. Boudjlida, D. Cheng and N. Guelfi (eds), *Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE '06), Forum Proceedings, Theme: Trusted Information Systems, Luxembourg, June 5-9, 2006*, Vol. 231 of *CEUR Workshop Proceedings*, CEUR-WS.org.

Beydoun, G., Gonzalez-Perez, C., Low, G. and Henderson-Sellers, B. (2005). Synthesis of a generic MAS metamodel, *Proceedings of the fourth international Workshop on Software Engineering for large-scale Multi-agent Systems (SELMAS '05)*, ACM Press, New York, NY, USA, pp. 1–5.

Bézivin, J. and Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework, *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*, IEEE Computer Society, Washington, DC, USA, p. 273.

Blake, M. B. and Gomaa, H. (2005). Agent-oriented compositional approaches to services-based cross-organizational workflow, *Decision Support System* **40**(1): 31–50.

Blanke, K., Krafzig, D. and Slama, D. (2004). *Enterprise SOA: Service Oriented Architecture Best Practices*, Prentice Hall, Upper Saddle River, NJ, USA.

Blum, B. I. (1994). A taxonomy of software development methods, *Communications of the ACM* **37**(11): 82–94.

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

Bond, A. H. and Gasser, L. (1988). *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann, San Mateo, CA, USA.

Booch, G. (1995). *Object solutions: managing the object-oriented project*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Booth, D., Haas, H., Mccabe, F., Newcomer, E., Champion, M., Ferris, C. and Orchard, D. (2004). Web Services Architecture, *Technical report*, World Wide Web Consortium.

Bordini, R., Braubach, L., Dastani, M., Seghrouchni, A. E. F., Gomez-Sanz, J., Leite, J., O'Hare, G., Pokahr, A. and Ricci, A. (2006). A survey of programming languages and platforms for multi-agent systems, *Informatica* **30**: 33–44.

Bordini, R. H., Dastani, M. and Winikoff, M. (2007a). Current issues in multi-agent systems development (invited paper), *in* G. M. P. O'Hare, A. Ricci, M. J. O'Grady and O. Dikenelli (eds), *Engineering Societies in the Agents World VII, 7th International Workshop, ESAW 2006 Dublin, Ireland, September 6-8, 2006 Revised Selected and Invited Papers*, Vol. 4457 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Berlin et al., pp. 38–61.

Bordini, R. H., Dastani, M., Dix, J. and Seghrouchni, A. E. F. (2009). *Multi-Agent Programming: Languages, Tools and Applications*, Springer Verlag, Berlin et al.

Bordini, R. H., Wooldridge, M. and Hübner, J. F. (2007b). *Programming Multi-Agent Systems in AgentSpeak using Jason*, John Wiley & Sons, Inc.

Boydens, J. and Steegmans, E. (2004). Model Driven Architecture: The next abstraction level in programming, *in* L. De Backer (ed.), *Proceedings of the First European Conference on the Use of Modern Information and Communication Technologies*, pp. 97–104.

Brandão, A., Alencar, P. S. C. and de Lucena, C. J. P. (2004). AgentZ: Extending Object-Z for multi-agent systems specification, *in* P. Bresciani, P. Giorgini, B. Henderson-Sellers, G. Low and M. Winikoff (eds), *Agent-Oriented Information Systems II, 6th International Bi-Conference Workshop, AOIS 2004, Riga, Latvia, June 8, 2004 and New York, NY, USA, July 20, 2004, Revised Selected Papers*, Vol. 3508 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 125–139.

Bratman, M. E. (1987). *Intentions, Plans, and Practical Reason*, Harvard University Press, Cambridge, MA.

Braubach, L. and Pokahr, A. (2007). Goal-oriented interaction protocols, *in* P. Petta, J. P. Müller, M. Klusch and M. P. Georgeff (eds), *Proceedings of the Fifth German Conference on Multi-Agent System TEchnologieS (MATES-2007)*, Vol. 4687 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin and Heidelberg, pp. 85–97.

Braubach, L., Pokahr, A. and Lamersdorf, W. (2004). Jadex: A short overview, *Main Conference Net.ObjectDays 2004*, pp. 195–207.

Braubach, L., Pokahr, A. and Lamersdorf, W. (2005). Jadex: A BDI-agent system combining middleware and reasoning, *in* M. Calisti, M. Walliser, S. Brantschen, M. Herbstritt, R. Unland, M. Calisti and M. Klusch (eds), *Software Agent-Based Applications, Platforms and Development Kits*, Whitestein Series in Software Agent Technologies and Autonomic Computing, Birkhäuser Basel, pp. 143–168. Book chapter.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. and Mylopoulos, J. (2004). Tropos: An Agent-Oriented Software Development Methodology, *International Journal on Autonomous Agents and Multiagent Systems (JAAMAS)* **8**(3): 203–236.

Brooks, R. (1991). Intelligence without reason, *Proceedings of the 12th International Joint Conference on Artifical Intelligence, ICAI-91*, Morgan Kaufmann Publishers, San Mateo, CA, USA, pp. 569–595.

Brooks, R. A. (1986). A robust layered control system for a mobile robot, *IEEE Journal Of Robotics And Automation* **RA-2**: 14–23.

Brooks, R. A. (1990). The behavior language: User's guide, *Technical Report AIM-1227*, MIT Artificial Intelligence Laboratory.

Brown, A. W. (2004). Model driven architecture: Principles and practice, *Software and System Modeling* **3**(4): 314–327.

Cabac, L. and Moldt, D. (2004). Formal semantics for AUML agent interaction protocol diagrams, *in* J. Odell, P. Giorgini and J. P. Müller (eds), *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, Vol. 3382 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 47–61.

Cabri, G., Leonardi, L. and Puviani, M. (2007). Service-oriented agent methodologies, *Proceedings of the 16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '07)*, IEEE Computer Society, Washington, DC, USA, pp. 24–29.

Caire, G., Coulier, W., Garijo, F. J., Gomez, J., Pavón, J., Leal, F., Chainho, P., Kearney, P. E., Stark, J., Evans, R. and Massonet, P. (2002). Agent oriented analysis using Message/UML, *Revised Papers and Invited Contributions from the Second International Workshop on Agent-Oriented Software Engineering II (AOSE '01)*, Vol. 2222 of *Lecture Notes in Computer Science*, Springer Verlag, London, UK, pp. 119–135.

Casati, F. and Shan, M. (2001). Dynamic and adaptive composition of e-services, *Information Systems* **26**: 143–163.

Castelfranchi, C. (1995). Guarantees for autonomy in cognitive agent architecture, *in* M. Wooldridge and N. Jennings (eds), *Working Notes of the IJCAI-95 Workshop on Agent Theories, Architectures, and Languages*, Vol. 890 of *Lecture Nodes in Artificial Intelligence*, Springer Verlag, Berlin, Heidelberg, New York, pp. 56–70.

Cernuzzi, L. and Zambonelli, F. (2004). Experiencing AUML in the GAIA methodology, *Proceedings of the 6th International Conference on Enterprise Information Systems, Porto, Portugal, April 14-17, 2004*, pp. 283–288.

Cernuzzi, L. and Zambonelli, F. (2008). Profile based comparative analysis for AOSE methodologies evaluation, *Proceedings of the 2008 ACM symposium on Applied computing (SAC '08)*, ACM Press, New York, NY, USA, pp. 60–65.

Cervenka, R. and Trencansky, I. (2004). Agent modeling language specification, version 0.9, *Technical report*, Whitestein Technologies AG.

Cervenka, R., Greenwood, D. and Trencansky, I. (2006). The AML approach to modeling autonomic systems, *Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS '06)*, IEEE Computer Society, Washington, DC, USA, p. 29.

Cervenka, R., Trencanský, I., Calisti, M. and Greenwood, D. A. P. (2004). AML: Agent modeling language toward industry-grade agent-based modeling, *in* J. Odell, P. Giorgini and J. P. Müller (eds), *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004. Revised Selected Papers*, Vol. 3382 of *Lecture Notes in Computer Science 3382*, Springer Verlag, Berlin et al., pp. 31–46.

Chapman, D. (1987). Planning for conjunctive goals, *Artificial Intelligence* **32**(3): 333–337.

Chella, A., Cossentino, M. and Sabatucci, L. (2004a). Tools and patterns in designing multi-agent systems with PASSI, **3**(1): 352–358.

Chella, A., Cossentino, M., Sabatucci, L. and Seidita, V. (2004b). From PASSI to Agile PASSI: Tailoring a design process to meet new needs, *IAT '04: Proceedings of the Intelligent Agent Technology, IEEE/WIC/ACM International Conference*, IEEE Computer Society, Washington, DC, USA, pp. 471–474.

Chen, Y. and Miao, H. (2004). From an abstract Object-Z specification to UML diagram, *Journal of Information & Computational Science* **1**(2): 319–324.

Cheong, C. and Winikoff, M. (2005a). Hermes: A methodology for goal oriented agent interactions., *International Conference on Autonomous Agents and Mulitagent Systems (AAMAS-05)*, ACM Press, New York, NY, USA, pp. 1121–1122.

Cheong, C. and Winikoff, M. (2005b). Hermes: Implementing goal-oriented agent interactions, *in* R. H. Bordini, M. Dastani, J. Dix and A. E. Fallah-Seghrouchni (eds), *Programming Multi-Agent Systems, Third International Workshop, ProMAS 2005, Utrecht, The Netherlands, July 26, 2005, Revised and Invited Papers*, Vol. 3862 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 168–183.

Ciancarini, P. and Wooldridge, M. J. (eds) (2001). *Agent-Oriented Software Engineering. First International Workshop, AOSE-2000, Limerick, Ireland, June 10, 2000*, Vol. 1957 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Clark, T., Evans, A., Sammut, P. and Willans, J. (2004a). An eXecutable metamodelling facility for domain specific language design, *Proceedings of he 4th OOPSLA Workshop on Domain-Specific Modeling*.

Clark, T., Evans, A., Smmut, P. and Willans, J. (2004b). *Applied Metamodelling: A Foundation for Language Driven Development*, Xactium.

Cohen, P. R. and Levesque, H. J. (1979). Elements of a plan based theory of speech acts, *Cognitive Science* **3**: 177–212.

Collier, R. W., O'Hare, G. M. P. and Rooney, C. (2004). A uml-based software engineering methodology for agent factory, *in* F. Maurer and G. Ruhe (eds), *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada, June 20-24, 2004*, pp. 25–30.

Cook, S. (2004). Domain-specific modeling and model-driven architecture, *The MDA Journal: Model Driven Architecture Straight from the Masters, Chap. 3, Electronic journal available at http://www.bptrends.com/*.

Cook, S., Jones, G., Kent, S. and Wills, A. C. (2007). *Domain-Specific Development with Visual Studio DSL Tools*, Addison-Wesley Professional; 1 edition.

Cossentino, M. (2005). From requirements to code with the PASSI methodology, *in* B. Henderson-Sellers and P. Giorgini (eds), *Agent-Oriented Methodologies*, Idea Group Inc., Hershey, PA, USA.

Cossentino, M. and Potts, C. (2002). A case tool supported methodology for the design of multi-agent systems, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*.

Cossentino, M., Gaud, N., Galland, S., Hilaire, V. and Koukam, A. (2007). A holonic metamodel for agent-oriented analysis and design, *Proceedings of the 3rd International Conference on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS '07)*, Springer Verlag, Berlin, Heidelberg, pp. 237–246.

Cossentino, M., Gaud, N., Hilaire, V., Galland, S. and Koukam, A. (2009). ASPECS: an agent-oriented software process for engineering complex systems, *International Journal on Autonomous Agents and Multiagent Systems (JAAMAS)* **20**(2): 260–302.

Cranefiled, S., Purvis, M., Nowostawski, M. and Hwang, P. (2002). Ontologies for interaction protocols, *Proceedings of the Workshop on Ontologies in Agent Systems, 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy*.

Cuesta, P., Gómez, A., González, J. and Rodríguez, F. A. (2003). Framework for evaluation of agent oriented methodologies, *Actas del Taller de Agentes Inteligentes en el tercer milenio (CAEPIA'2003), San Sebastián (Spain), November 2003*.

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches, *in* J. Bettin, G. van Emde Boas, A. Agrawal, E. Willink and J. Bezivin (eds), *Proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches, *IBM Systems Journal* **45**(3): 621–645.

da Silva, V. T., Choren, R. and de Lucena, C. J. P. (2004). A UML based approach for modeling and implementing multi-agent systems, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '04)*, IEEE Computer Society, Washington, DC, USA, pp. 914–921.

Dalpiaz, F., Molesini, A., Puviani, M. and Seidita, V. (2008). Towards filling the gap between AOSE methodologies and infrastructures: requirements and meta-model, *Proceedings of WOA08, Evolution of Agent Development: Methodologies, Tools, Platforms and Languages. WOA - Nona Edizione. Palermo, Italy. 17-18 September*, pp. 115–121.

Dam, K. H. and Winikoff, M. (2004). Comparing agent-oriented methodologies, *Proceedings of 5th International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS), Melbourne, Australia, 2003*, Vol. 3030 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 78–93.

Dam, K. H., Winikoff, M. and Padgham, L. (2006). An agent-oriented approach to change propagation in software evolution, *Proceedings of the 17th Australian Software Engineering Conference (ASWEC 2006), 18-21 April 2006, Sydney, Australia*, IEEE Computer Society, Washington, DC, USA, pp. 309–318.

Danč, J. (2008). *Formal specification of AML*, Master's thesis, Department of Computer Science Faculty of Mathematics, Physics and Informatics, Comenius University.

Dastani, M., de Boer, F. S., Dignum, F. and Meyer, J.-J. C. (2003). Programming agent deliberation: an approach illustrated using the 3APL language, *Proceeding of the Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia*, ACM Press, New York, NY, USA, pp. 97–104.

Davidsson, P. (2001). Categories of artificial societies, *Engineering Societies in the Agents World II*, Vol. 2203 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 1–9.

Davis, J. (2003). GME: the generic modeling environment, *Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, ACM Press, New York, NY, USA, pp. 82–83.

de Cerqueira Gatti, M. A., von Staa, A. and de Lucena, C. J. P. (2007). AUML-BP: A basic agent oriented software development process model using AUML, *Technical Report 21/07*, Laboratório de Engenharia de Software–LES.

Decker, G. and Puhlmann, F. (2007). Extending BPMN for modeling complex choreographies, *in* R. Meersman and Z. Tari (eds), *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, Vol. 4803 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 24–40.

Decker, G., Kopp, O., Leymann, F. and Weske, M. (2007). BPEL4Chor: extending BPEL for modeling choreographies, *Proceedings of the International Conference on Web Services (ICWS 2007)*, IEEE Computer Society, Washington, DC, USA, pp. 296–303.

Decreus, K. and Poels, G. (2009). Mapping semantically enriched formal Tropos to business process models, *Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09)*, ACM Press, New York, NY, USA, pp. 371–376.

del Mar Gallardo, M. and Merino, P. (1999). A framework for automatic construction of abstract Promela models, *in* D. Dams, R. Gerth, S. Leue and M. Massink (eds), *Proceedings of Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999*, Vol. 1680 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 184–199.

DeLoach, S. A. (2001). Analysis and design using MaSE and agentTool, *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS)*.

DeLoach, S. A. (2005). Multiagent systems engineering of organization-based multiagent systems, *SIGSOFT Software Engineering Notes* **30**(4): 1–7.

DeLoach, S. A. (2007). Developing a multiagent conference management system using the O-MaSE process framework, *in* M. Luck and L. Padgham (eds), *Proceedings of the 8th International Workshop on Agent-Oriented Software Engineering VIII, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, Vol. 4951 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 168–181.

DeLoach, S. A. and Valenzuela, J. L. (2006). An agent-environment interaction model, *in* L. Padgham and F. Zambonelli (eds), *Agent-Oriented Software Engineering VII, 7th International Workshop, AOSE 2006, Hakodate, Japan, May 8, 2006, Revised and Invited Papers*, Vol. 4405 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 1–18.

DeLoach, S. A., Padgham, L., Perini, A., Susi, A. and Thangarajah, J. (2009). Using three AOSE toolkits to develop a sample design, *International Journal on Agent-Oriented Software Engineering (IJAOSE)* **3**(4): 416–476.

DeLoach, S. A., Wood, M. F. and Sparkman, C. H. (2001). Multiagent systems engineering, *The International Journal of Software Engineering and Knowledge Engineering* **11**(3): 231–258.

Demuth, B. (2004). The Dresden OCL toolkit and its role in information systems development, *13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice Conference, Advances in Theory, Practice and Education (ISD'2004), Vilnius, Lithuania, 9-11 September*.

Demuth, B., Hussmann, H. and Konermann, A. (2005). Generation of an OCL 2.0 parser, *in* T. Baar (ed.), *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*, Technical Report LGL-REPORT-2005-001, EPFL, pp. 38–52.

Deursen, A. V. and Klint, P. (1997). Little languages: Little maintenance?, *Journal of Software Maintenance* **10**: 10–75.

Dickinson, I. and Wooldridge, M. (2003). Towards practical reasoning agents for the Semantic Web, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '03)*, ACM Press, New York, NY, USA, pp. 827–834.

Dickinson, I. and Wooldridge, M. (2005). Agents are not (just) Web services: Considering BDI agents and Web services, *Proceedings of the Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE '2005), Utrecht, The Netherlands, July 2005*.

Dignum, V. and Dignum, F. (2007). Coordinating tasks in agent organizations, *Coordination, Organizations, Institutions, and Norms in Agent Systems II: AAMAS 2006 and ECAI 2006 International Workshops, COIN 2006 Hakodate, Japan, May 9, 2006 Riva del Garda, Italy, August 28, 2006. Revised Selected Papers*, Vol. 4386/2007 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Berlin, Heidelberg, pp. 32–47.

Dinkloh, M. and Nimis, J. (2003). A tool for integrated design and implementation of conversations in multiagent systems, *in* M. Dastani, J. Dix and A. E. Fallah-Seghrouchni (eds), *Programming Multi-Agent Systems, First International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003, Selected Revised and Invited Papers*, Vol. 3067 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 187–200.

d'Inverno, M. and Luck, M. (2001a). Formal agent development: Framework to system, *Formal Approaches to Agent-Based Systems: First International Workshop, FAABS 2000* pp. 133–147.

d'Inverno, M. and Luck, M. (2001b). *Understanding agent systems*, Springer Verlag, New York.

d'Inverno, M., Luck, M., Georgeff, M., Kinny, D. and Wooldridge, M. (2004). The dMARS architecture: A specification of the distributed multi-agent reasoning system, *International Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* **9**(1-2): 5–53.

Doi, T., Tahara, Y. and Honiden, S. (2005). IOM/T: an interaction description language for multi-agent systems, *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '05)*, ACM Press, New York, NY, USA, pp. 778–785.

Doyle, D., Geers, H., Graaf, B. and van Deursen, A. (2007). Migrating a domain-specific modeling infrastructure to MDA technology, *in* J.-M. Favre, D. GaŽevic, R. Lämmel and A. Winter (eds), *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies*, Vol. 4364 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 52–55.

D'Souza, D. (2001). Model-driven architecture and integration - opportunities and challenges, Version 1.1, Kineticum.

Durfee, E. H. (1999). Distributed problem solving and planning, *in* G. Weiss (ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, MIT Press, Cambridge, MA, USA, pp. 121–164.

Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D. and Gyapay, S. V. (2005). Model transformation by graph transformation: A comparative study, *Proceedings of the International Workshop on Model Transformations in Practice (MTiP '05) at MoDELS Conference, Montego Bay, Jamaica*.

Ehrler, L. and Cranefield, S. (2004). Executing Agent UML diagrams, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 04)*, Vol. 2, IEEE Computer Society, Washington, DC, USA, pp. 906–913.

Eijk, P. V. and Diaz, M. (eds) (1989). *Formal Description Technique Lotos: Results of the Esprit Sedos Project*, Elsevier Science Inc., New York, NY, USA.

Eker, S., Meseguer, J. and Sridharanarayanan, A. (2002). The Maude LTL model checker, *in* F. Gadducci and U. Montanari (eds), *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, Vol. 71 of *Electronic Notes in Theoretical Computer Science*, Elsevier, p. 27.

Elvesæeter, B., Panfilenko, D., Jacobi, S. and Hahn, C. (2010). Aligning business and IT models in service-oriented architectures using BPMN and SoaML, *Proceedings of the First International Workshop on Model-Drive Interoperability*, MDI '10, ACM, New York, NY, USA, pp. 61–68.

Endert, H., Hirsch, B., Küster, T. and Albayrak, S. (2007). Towards a mapping from BPMN to agents, *in* J. Huang, R. Kowalczyk, Z. Maamar, D. L. Martin, I. Müller, S. Stoutenburg and K. P. Sycara (eds), *Proceedings on the Internal Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE 2007), AAMAS 2007 Honolulu, HI, USA, May 14, 2007*, Vol. 4504 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 92–106.

Erl, T. (2005). *Service-Oriented Architecture : Concepts, Technology, and Design*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Estefan, J. A. (2007). Survey of model-based systems engineering (MBSE) methodologies, *Technical report*, INCOSE MBSE Focus Group.

Evans, A. (2006). Domain specific languages and MDA, *Technical report*, Xactium Limited.

Falleri, J., Huchard, M. and Nebut, C. (2006). Towards a traceability framework for model transformations in kermeta, *in* J. Aagedal, T. Neple and J. Oldevik (eds), *Proceedings of the ECMDA Traceability Workshop (ECMDA-TW'06) Bilbao, Spain*, pp. 31–40.

Favre, J.-M. (2004). Foundations of model (driven) (reverse) engineering - episode i: Story of the fidus papyrus and the solarus, *Post-proceedings of Dagsthul Seminar on Model Driven Reverse Engineering*.

Ferber, J. and Gutknecht, O. (1998). Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems, *Third International Conference on Multi-Agent Systems (ICMAS), Paris, 1998*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 128–135.

Ferber, J., Gutknecht, O. and Michel, F. (2004). From agents to organizations: an organizational view of multi-agent systems, *in* P. Giorgini, J. Müller and J. Odell (eds), *Agent-Oriented Software Engineering (AOSE) IV,*, Vol. 2935 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 214–230.

Finin, T., Fritzson, R., McKay, D. and McEntire, R. (1994a). KQML - a language and protocol for knowledge and information exchange, *Technical Report CS-94-02*, Computer Science Department, University of Maryland and Valley Forge Engineering Center, Unisys Corporation, Computer Science Department, University of Maryland, UMBC Baltimore MD 21228.

Finin, T., Fritzson, R., McKay, D. and McEntire, R. (1994b). KQML as an agent communication language, *Proceedings of the Third International Conference on Information and Knowledge Management*, ACM Press, New York, NY, USA, pp. 456–463.

Firby, J. (1989). *Adaptive Execution in Complex Dynamic Domains*, PhD thesis, Yale University.

Firby, J. R. (1994). Task networks for controlling continuous processes, *Proceedings of the Second International Conference on AI Planning Systems*, Chicago, Illinois, pp. 49–54.

Firby, J. R. (1995). The RAP Language Manual, *Technical Report Animate Agent Project Working Note APP-6*, University of Chicago.

Fischer, K., Elvesæter, B., Berre, A.-J., Hahn, C., Madrigal-Mora, C. and Zinnikus, I. (2006). Model-driven design of interoperable agents, *Proceedings of the 2nd Workshop on Web Service Interoperability (WSI 2006), Bordeaux, France, 2006, Interoperability for Enterprise Software and Applications, ISTE Ltd.*, pp. 119–130.

Fischer, K., Florian, M. and Malsch, T. (eds) (2005). *Socionics - Scalability of Complex Social Systems*, Vol. 3413 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Fischer, K., Hahn, C. and Madrigal-Mora, C. (2007). Agent-oriented software engineering: a model-driven approach, *International Journal on Agent-oriented Engineering (IJAOSE)* **1**(3/4): 334–369.

Fischer, K., Hahn, C. and Warwas, S. (2009). Modeling teletruck: A case study, *Engineering Societies in the Agents World IX: 9th International Workshop, ESAW 2008, Saint-Etienne, France, September 24-26, 2008, Revised Selected Papers*, Springer Verlag, Berlin, Heidelberg, pp. 1–26.

Foundation for Intelligent Physical Agents (2002). FIPA communicative act library specification, version j, http://www.fipa.org/specs/fipa00037/.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap, *2007 Future of Software Engineering (FOSE '07)*, IEEE Computer Society, Washington, DC, USA, pp. 37–54.

Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M. and Traverso, P. (2004). Specifying and analyzing early requirements in Tropos, *Requirements Engineering* **9**(2): 132–150.

Fuxman, A., Pistore, M., Mylopoulos, J. and Traverso, P. (2001). Model checking early requirements specifications in Tropos, *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, IEEE Computer Society, Washington, DC, USA, pp. 174–181.

García-Magariño, I., Fuentes-Fernández, R. and Gómez-Sanz, J. J. (2009). INGENIAS development process assisted with chains of transformations, *in* J. Cabestany, F. Sandoval, A. Prieto and J. M. Corchado (eds), *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*, Vol. 5517 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 514–521.

Garcia-Ojeda, J. C., DeLoach, S. A. and Robby (2009). agentTool III: from process definition to code generation, *in* C. Sierra, C. Castelfranchi, K. S. Decker and J. S. Sichman (eds), *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1393–1394.

García-Ojeda, J. C., DeLoach, S. A., Robby, Oyenan, W. H. and Valenzuela, J. (2007). O-MaSE: A customizable approach to developing multiagent development processes, *in* M. Luck and L. Padgham (eds), *Agent-Oriented Software Engineering VIII, 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, Vol. 4951 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 1–15.

Gary T. Leavens, G. T., Baker, A. L. and Ruby, C. (1999). JML: A notation for detailed design, *in* H. Kilov, B. Rumpe and I. Simmonds (eds), *Behavioral Specifications of Businesses and Systems*, Kluwer Academic Publishers, Dordrecht, Boston, London, pp. 175–188.

Gascue na, J. M. and Fernández-Caballero, A. (2009). Prometheus and INGENIAS agent methodologies: A complementary approach, *Agent-Oriented Software Engineering IX: 9th International Workshop, AOSE 2008 Estoril, Portugal, May 12-13, 2008 Revised Selected Papers*, Springer Verlag, Berlin, Heidelberg, pp. 131–144.

Gasser, L. (1992). An overview of DAI, *in* L. Gasser and N. Avouris (eds), *Distributed Artificial Intelligence: Theory and Praxis*, Kluwer Academic Publishers, Norwell, MA, USA, pp. 9–30.

Gasser, L. (2001). Perspectives on organizations in multi-agent systems, *Multi-agents systems and applications*, Springer Verlag, New York, NY, USA, pp. 1–16.

Georgeff, M. P., Pell, B., Pollack, M. E., Tambe, M. and Wooldridge, M. (1999). The Belief-Desire-Intention model of agency, *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages (ATAL '98)*, Springer Verlag, London, UK, pp. 1–10.

Ghezzi, C., Jazayeri, M. and Mandrioli, D. (2002). *Fundamentals of Software Engineering*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Giorgini, P., Mylopoulos, J. and Sebastiani, R. (2005). Goal-oriented requirements analysis and reasoning in the Tropos methodology, *Engineering Applications of Artificial Intelligence* **18**(2): 159–171.

Giorgini, P., Perini, A., Mylopoulos, J., Giunchiglia, F. and Bresciani, P. (2001). Agent-oriented software development: A case study, *in* S. Sen, J. P. Müller, E. Andre and C. Frassen (eds), *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2001), Sheraton Buenos Aires Hotel, Buenos Aires, Argentina, June 13-15, 2001*, pp. 283–290.

Giunchiglia, F., Odell, J. and Weiß, G. (eds) (2003). *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, Vol. 2585 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Gogolla, M., Büttner, F. and Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL, *Science of Computer Programming* **69**(1-3): 27–34.

Gomez-Sanz, J. J., Fuentes, R., Pavón, J. and García-Magariño, I. (2008a). INGENIAS development kit: a visual multi-agent system development environment, *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '08)*, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1675–1676.

Gomez-Sanz, J. J., Michel, F., Platon, E. and Ricci, A. (2008b). Towards an agent-oriented paradigm, *in* D. Weyns (ed.), *Position Statement for FOSE-MAS at AAMAS 2008*.

Gracanin, D., Singh, H. L., Hinchey, M. G., Eltoweissy, M. and Bohner, S. A. (2005). A CSP-based agent modeling framework for the Cougaar agent-based architecture, *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS '05)*, IEEE Computer Society, Washington, DC, USA, pp. 255–262.

Gray, J., Lin, Y. and Zhang, J. (2006). Automating change evolution in model-driven engineering, *Computer* **39**(2): 51–58.

Greenfield, J., Short, K., Cook, S. and Kent, S. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley and Sons, Inc., New York, NY, USA.

Gronmo, R., Belaunde, M., Aagedal, J., Engel, K.-D., Faugere, M. and Solheim, I. (2005). Evaluation of the proposed QVTMerge language for model transformations, *in* S. Bevinakoppa, L. F. Pires and S. Hammoudi (eds), *Web Services and Model-Driven Enterprise Information Services, Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Services, WSMDEIS 2005, In conjunction with ICEIS 2005, Miami, USA, May 2005*, INSTICC Press, pp. 65–74.

Gründel, T. (2009). *Eine generische Transformation zur Überbrückung von Design und Implementierung von Multiagenten Systemen*, Master's thesis, Universität des Saarlandes.

Guessoum, Z. (2005). MAS Meta-Models and MDA, AgentLink III AOSE TFG2. Online at: http://www.pa.icar.cnr.it/c̆ossentino/al3tf2/docs/zahia_slovenia.pdf.

Guessoum, Z. and Briot, J. (1999). From active object to autonomous agents, *IEEE Concurrency* **7**(3): 68–78.

Guizzardi-Silva Souza, R., Perini, A. and Dignum, V. (2003). Using intentional analysis to model knowledge management requirements in communities of practice, *Technical Report TR-CTIT-03-53*, Centre for Telematics and Information Technology, University of Twente, Enschede.

Hahn, C. (2004). *A detailed analysis of holonic multiagent systems*, Master's thesis, Universität des Saarlandes.

Hahn, C. (2008). A domain specific modeling language for multiagent systems, *in* L. Padgham, D. C. Parkes, J. Müller and S. Parsons (eds), *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008*, Vol. 1, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 233–240.

Hahn, C. and Fischer, K. (2007). Service composition in holonic multiagent systems: Model-driven choreography and orchestration, *in* V. Mavrík, V. Vyatkin and A. W. Colombo (eds), *Proceedings of the Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS 2007), Regensburg, Germany, September 3-5, 2007, Proceedings*, Vol. 4659 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 47–58.

Hahn, C. and Fischer, K. (2008a). The dynamic semantics of the domain specific modeling language for multiagent systems, *Proceedings of the Agent-based Technologies and Applications for Enterprise InterOPerability (ATOP 2008). Workshop at AAMAS'08, Estoril, Portugal, 13.5.2008*, pp. 25–38.

Hahn, C. and Fischer, K. (2008b). The formal semantics of the domain specific modeling language for multiagent systems, *in* M. Luck and J. J. Gómez-Sanz (eds), *Proceedings of the 9th International Workshop Agent-Oriented Software Engineering IX (AOSE '08), Estoril, Portugal, May 12-13, 2008, Revised Selected Papers*, Vol. 5386 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 145–158.

Hahn, C. and Slomic, I. (2008). Agent-based extensions for the UML profile and metamodel for service-oriented architectures, *Proceedings of the 12th Enterprise Distributed Object Computing Conference (EDOCW '08), Third International Workshop on Modeling, Design, and Analysis for Service-oriented Architectures (2008)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 309–316.

Hahn, C. and Zinnikus, I. (2008). Modeling and executing service interactions using an agent-oriented modeling language, *in* Z. Bellahsene, C. Woo, E. Hunt, X. Franch and R. Coletta (eds), *Proceedings of the Forum at the CAiSE'08 Conference, Montpellier, France, June 18-20, 2008*, Vol. 344 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 37–40.

Hahn, C., Dmytro, P. and Fischer, K. (2010a). A model-driven approach to close the gap between business requirements and agent-based execution, *Proceedings of the 4th Workshop on Agent-based Technologies and Applications for Enterprise Interoperability (ATOP 2010) held in conjunction with AAMAS 2010, Toronto, Canada, June, 10th*.

Hahn, C., Fley, B. and Florian, M. (2005). A framework for the design of self-regulation of open agent-based electronic marketplaces, *Proceedings of the Symposium on Normative Multi-Agent Systems, NORMAS 2005, part of the SSAISB 2005 Convention, University of Hertfordshire, Hatfield, UK, 12-15 April 2005*, pp. 8–23.

Hahn, C., Fley, B. and Florian, M. (2006a). Self-regulation through social institutions: A framework for the design of open agent-based electronic marketplaces, *Computational and Mathematical Organization Theory* **12**(2-3): 181–204.

Hahn, C., Fley, B., Florian, M., Spresny, D. and Fischer, K. (2007a). Social reputation: a mechanism for flexible self-regulation of multiagent systems, *Journal of Artificial Societies and Social Simulation* **10**(1): 2.

Hahn, C., Jacobi, S. and Raber, D. (2010b). Enhancing the interoperability between multiagent systems and service-oriented architectures through a model-driven approach, *in* J. Dix and C. Witteveen (eds), *Multiagent System Technologies, 8th German Conference, MATES 2010, Leipzig, Germany, September 27-29, 2010. Proceedings*, Vol. 6251 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 88–99.

Hahn, C., Jacobi, S. and Raber, D. (2010c). Enhancing the interoperability between multiagent systems and service-oriented architectures through a model-driven approach, *in* J. X. Huang, A. A. Ghorbani, M.-S. Hacid and T. Yamaguchi (eds), *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2010, Toronto, Canada, August 31 - September 3, 2010*, IEEE Computer Society Press, Washington, DC, USA, pp. 415–422.

Hahn, C., Madrigal-Mora, C. and Fischer, K. (2007b). Interoperability through a platform-independent model for agents, *in* R. J. Goncalves, J. Müller, K. Mertins and M. Zelm (eds), *Enterprise Interoperability II - New Challenges and Approaches*, Springer Verlag, Berlin et al., pp. 195–206.

Hahn, C., Madrigal-Mora, C. and Fischer, K. (2007c). Interoperability through a platform-independent model for agents, *in* K. M. Ricardo J. Gonçalves, Jörg P. Müller and M. Zelm (eds), *Enterprise Interoperability II, New Challenges and Approaches, Proceedings of the third International Conference on Interoperability for Enterprise Software and Applications (I-ESA)*, Springer Verlag, London, pp. 195–206.

Hahn, C., Madrigal-Mora, C. and Fischer, K. (2007d). A platform-independent model for agents, *Technical Report RR-07-01*, German Research Center for Artificial Intelligence.

Hahn, C., Madrigal-Mora, C. and Fischer, K. (2009a). A platform-independent metamodel for multiagent systems, *International Journal on Autonomous Agents and Multi-Agent Systems (JAAMAS)* **18**(2): 239–266.

Hahn, C., Madrigal-Mora, C., Fischer, K., Elvesæter, B., Berre, A.-J. and Zinnikus, I. (2006b). Meta-models, models, and model transformations: Towards interoperable agents, *in* K. Fischer, I. J. Timm, E. André and N. Zhong (eds), *Proceedings of the 4th German Conference on Multiagent System Technologies (MATES) 2006, Erfurt, Germany, September 19-20, 2006*, Vol. 4196 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 123–134.

Hahn, C., Neple, T. and Limyr, A. (2006c). Comparing model transformation approaches, *Proceedings of the 7th Working Conference on Virtual Enterprises (PRO-VE'06), Helsinki, Finland, 25-27 September 2006*.

Hahn, C., Nesbigall, S., Warwas, S., Fischer, K. and Klusch, M. (2008a). Model-driven approach to the integration of multiagent systems and semantic Web services, *Proceedings of the 12th Enterprise Distributed Object Computing Conference (EDOCW'08), Third International Workshop on Modeling, Design, and Analysis for Service-oriented Architectures (1008)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 317–324.

Hahn, C., Nesbigall, S., Warwas, S., Zinnikus, I., Fischer, K. and Klusch, M. (2008b). Integration of multiagent systems and Semantic Web Services on a platform independent level, *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Sydney, NSW, Australia, December 9-12, 2008*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 200–206.

Hahn, C., Panfilenko, D. and Fischer, K. (2010d). A model-driven approach to close the gap between business requirements and agent-based execution, *4th Workshop on Agent-based Technologies and applications for enterprise interoperability. International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-10), May 10-14, Toronto, Canada*, ASR, pp. 13–24.

Hahn, C., Zinnikus, I., Warwas, S. and Fischer, K. (2009b). From agent interaction protocols to executable code: a model-driven approach, *in* C. Sierra, C. Castelfranchi, K. S. Decker and J. S. Sichman (eds), *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009*, Vol. 2, IFAAMAS, Richland, SC, pp. 1199–1200.

Hahn, C., Zinnikus, I., Warwas, S. and Fischer, K. (2011). Automatic generation of executable behavior: A protocol-driven approach, *in* M.-P. Gleizes and J. J. Gomez-Sanz (eds), *Agent-Oriented Software Engineering X - 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers*, Vol. 6038 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin et al., pp. 110–124.

Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"?, *Computer* **37**(10): 64–72.

Haugen, O. (2008). Challenges to UML 2 to describe FIPA Agent Protocol, *Proceedings of Agent-based Technologies and Applications for Enterprise interOPerability (ATOP 2008). Workshop held at the Seventh International Joint Conference on Autonomous Agents & Multiagent Systems*, pp. 37–46.

Hayes-Roth, B. (1995). An architecture for adaptive intelligent systems, *Artifical Intelligence: Special Issue in Agent and Interactivity* **72**(1-2): 329–365.

Hilaire, v., Koukam, A., Gruer, P. and Müller, J.-P. (2000). Formal specification and prototyping of MAS, *in* A. Omicini, R. Tolksdorf and F. Zambonelli (eds), *Engineering Societies in the Agents World*, Vol. 1972 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Berlin et al., pp. 114–127.

Hilaire, V., Simonin, O., Koukam, A. and Ferber, J. (2004). A formal approach to design and reuse agent and multiagent models, *in* J. Odell, P. Giorgini and J. P. Müller (eds), *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, Vol. 3382 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 142–157.

Hindriks, K. V., de Boer, F. S., van der Hoek, W. and Meyer, J.-J. C. (1999). Agent programming in 3APL, *International Journal on Autonomous Agents and Multi-Agent Systems (JAAMAS)* **2**(4): 357–401.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming, *Communication of the ACM* **12**(10): 576–580.

Hoare, C. A. R. (1973). Hints on programming language design, *Technical report*, Stanford University, Stanford, CA, USA.

Hoare, C. A. R. (1978). Communicating sequential processes, *Communication of the ACM* **21**(8): 666–677.

Huber, M. J. (1999). JAM: a BDI-theoretic mobile agent architecture, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, ACM Press, New York, NY, USA, pp. 236–243.

Huget, M. (2002a). Nemo: An agent-oriented software engineering methodology, *in* J. Debenham, B. Henderson-Sellers, N. Jennings and J. Odell (eds), *Proceedings of the Agent Oriented Methodologies Workshop at OOPSLA 2002, Seattle - USA, November 2002*, p. 41Ű53.

Huget, M.-P. (2002b). Generating code for Agent UML sequence diagrams, *Technical Report ULCS-02-020*, Department of computer science, University of Liverpool.

Huget, M.-P. (2002c). A language for exchanging agent UML protocol diagrams, *Technical Report ULCS-02-009*, Department of Computer Science, University of Liverpool.

Huget, M.-P. (2002d). Model checking Agent UML protocol diagrams, *Technical Report ULCS-02-012*, Department of Computer Science, University of Liverpool.

Huget, M.-P. and Odell, J. (2004). Representing agent interaction protocols with Agent UML, *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, IEEE Computer Society, Washington, DC, USA, pp. 1244–1245.

Humphrey, W. S. (1989). The software engineering process: definition and scope, *SIGSOFT Software Engineering Notes* **14**(4): 82–83.

Hyacinth, S. N., Ndumu, D. T., Lee, L. C., Collis, J. C. and Re, I. I. (1999). ZEUS: A tool-kit for building distributed multi-agent systems, *Applied Artificial Intelligence Journal* **13**: 129–186.

IEEE STD 610.12 (1990). Standard glossary of software engineering terminology, IEEE, May, ISBN: 155937067x.

Iseger, M. (2005). Domain-specific modeling for generative software development, *ITarchitect*.

Ivan, T., Cervenka, R. and Greenwood, D. (2006). Applying a UML-based agent modeling language to the autonomic computing domain, *Conference on Object Oriented Programming Systems Languages and Applications*, ACM Press, New York, NY, USA, pp. 521–529.

JACK Intelligent Agents (2005). User Manual Release 5, Agent Oriented Software Pty. Ltd, June 2005, http://www.agent-software.com/shared/demosNdocs/Agent ManualWEB/index.html.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, New York, NY, USA.

Jacobi, S., Hahn, C. and Raber, D. (2009). MasDISPO_xt - process optimisation by use of integrated, agentbased manufacturing execution systems inside the supply chain of steel production, *in* J. Cordeiro and J. Filipe (eds), *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS 2009), Volume AIDSS, Milan, Italy, May 6-10, 2009*, pp. 347–350.

Jacobi, S., Hahn, C. and Raber, D. (2010). Integration of multiagent systems and service oriented architectures in the steel industry, *in* J. X. Huang, A. A. Ghorbani, M.-S. Hacid and T. Yamaguchi (eds), *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2010, Toronto, Canada, August 31 - September 3, 2010*, IEEE Computer Society Press, Washington, DC, USA, pp. 479–482.

Jacobi, S., León-Soto, E., Madrigal-Mora, C. and Fischer, K. (2007). MasDISPO: a multiagent decision support system for steel production and control, *Proceedings of the 19th national conference on Innovative applications of artificial intelligence (IAAI'07)*, AAAI Press, pp. 1707–1714.

Jayatilleke, G., Thangarajah, J., Padgham, L. and Winikoff, M. (2006). Component agent framework for domain-experts (CAFnE) toolkit, *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, ACM Press, New York, NY, USA, pp. 1465–1466.

Jennings, N. (2000). On agent-based software engineering, *Artificial Intelligence* **177**(2): 277–296.

Jennings, N. (2001). An agent-based approach for building complex software systems, *Communications of the ACM* **44**(4): 35–41.

Jennings, N. R. (1999). Agent-based computing: Promise and perils, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Mateo, CA, USA, pp. 1429–1439.

Jennings, N. R., Faratin, P., Norman, T. J., O'Brien, P., Odgers, B. and Alty, J. L. (2000). Implementing a business process management system using ADEPT: A real-world case study, *International Journal of Applied Artificial Intelligence* **14**(5): 421–465.

Jennings, N. R., Mamdani, E. H., Laresgoiti, I., Perez, J. and Corera, J. (1992). GRATE: A general framework for cooperative problem solving, *IEE-BCS Journal of Intelligent Systems Engineering* **1**(2): 102–114.

Jennings, N. R., Sycara, K. and Wooldridge, M. (1998). A roadmap of agent research and development, *International Journal on Autonomous Agents and Multi-Agent Systems (JAAMAS)* **1**(1): 7–38.

Johnston, S. (2006). UML 2.0 profile for software services, *Technical report*, OMG. submitted to OMG ABSIG on SOA at 4/15 meeting in St. Louis.

Jouault, F., Allilaire, F., Bézivin, J. and Kurtev, I. (2008). ATL: A model transformation tool, *Science of Computer Programming* **72**(1-2): 31–39.

Juan, T. and Sterling, L. (2003). A meta-model for intelligent adaptive multi-agent systems in open environments, *Proceedings of the second international joint conference on Autonomous agents and multiagent systems (AAMAS '03)*, ACM Press, New York, NY, USA, pp. 1024–1025.

Juan, T., Pearce, A. and Sterling, L. (2002). ROADMAP: Extending the Gaia methodology for complex open systems, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, ACM Press, New York, NY, USA, pp. 3–10.

Juneidi, S. J. and Vouros, G. A. (2004). Evaluation of agent oriented software engineering main approaches, *Proceedings of the IASTED International Conference on Software Engineering (SE 2004), Innsbruck, Austria*.

Kahl, T., Zinnikus, I., Roser, S., Hahn, C., Ziemann, J., Müller, J. and Fischer, K. (2007). Architecture for the design and agent-based implementation of cross-organizational business processes, *in* R. J. Goncalves, J. Müller, K. Mertins and M. Zelm (eds), *Enterprise Interoperability II - New Challenges and Approaches*, Springer Verlag, Berlin et al., pp. 207–218.

Kardas, G., Ekinci, E. E., Afsar, B., Dikenelli, O. and Topaloglu, Y. N. (2009a). Modeling tools for platform specific design of multi-agent systems, *Proceedings of the 7th German Conference, MATES 2009, Hamburg, Germany, September 9-11, 2009*, Lecture Notes in Computer Science, Springer Verlag, Berlin et al., pp. 202–207.

Kardas, G., Goknil, A., Dikenelli, O. and Topaloglu, N. Y. (2009b). Model driven development of Semantic Web enabled multi-agent systems, *International Journal on Cooperative Information Systems* **18**(2): 261–308.

Kavantzas, N., Burdett, D., Ritzinger, G. and Lafon, Y. (2005). Web services choreography description language version 1.0, W3C candidate recommendation, november 2005. http://www.w3.org/tr/ws-cdl-10.

Kelly, S., Lyytinen, K. and Rossi, M. (1996). MetaEdit+: A fully configurable multi-user and multi-tool CASE environment, *Proceedings of the 8th International Conference on Advanced Information Systems Engineering (CAiSE'96)*, Vol. 1080 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 1–21.

Kim, S.-K. and Carrington, D. (2002). A formal model of the UML metamodel: The UML state machine and its integrity constraints, *Proceedings of the 2nd International Conference of B and Z Users Grenoble, France, January 23-25, 2002*, Vol. 2272/2002 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 101–114.

Kim, S.-K., Carrington, D. and Duke, R. (2001). A metamodel-based transformation between UML and Object-Z, *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, IEEE Computer Society, Washington, DC, USA, p. 112.

Kinny, D. (1999). The Agentis agent interaction model, *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages (ATAL '98)*, Springer Verlag, London, UK, pp. 331–344.

Kleppe, A., Warmer, J. and Bast, W. (2003). *MDA Explained, The Model-Driven Architecture: Practice and Promise*, Addison Wesley.

Knabe, T., Schillo, M. and Fischer, K. (2002). Improvements to the FIPA contract net protocol for performance increase and cascading applications, *International Workshop for Multi-Agent Interoperability at the German Conference on AI (KI-2002)*.

Koehler, J. and Srivastava, B. (2003). Web service composition: Current solutions and open problems, *ICAPS 2003 Workshop on Planning for Web Services*.

Koestler, A. (1967). *The Ghost in the Machine*, Hutchinson, London.

Kolovos, D. S., Paige, R. F., Kelly, T. and Polack, F. A. (2006). Requirements for domain-specific languages, *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006), Nantes, France, July 2006*.

Kuan, P. P., Rarunasekera, S. and Sterling, L. (2005). Improving goal and role oriented analysis for agent based systems, *Proceedings of the 2005 Australian conference on Software Engineering (ASWEC '05)*, IEEE Computer Society, Washington, DC, USA, pp. 40–47.

Kwon, G. (2000). Rewrite rules and operational semantics for model checking UML state charts, *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 2000)*, Vol. 1939 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 528–540.

Labrou, Y. K. and Finin, T. (2000). History, state of the art and challenges for Agent Communication Languages, *Informatik/Informatique*.

Labrou, Y. K., Finin, T. and Peng, Y. (1999). The current landscape of agent communication languages, *IEEE Intelligent Systems*.

Langlois, B., Jitia, C. E. and Jouenne, E. (2007). DSL classification, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, pp. 28–38.

Lapouchnian, A. and Lespérance, Y. (2006). Modeling mental states in the analysis of multiagent systems requirements, *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, ACM Press, New York, NY, USA, pp. 241–243.

Laukkanen, M., Tarkoma, S. and Leinonen, J. (2002). FIPA-OS agent platform for small-footprint devices, *Revised Papers from the 8th International Workshop on Intelligent Agents VIII (ATAL '01)*, Springer Verlag, London, UK, pp. 447–460.

Lawley, M. and Steel, J. (2005). Practical declarative model transformation with tefkat, *in* J.-M. Bruel (ed.), *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, Vol. 3844 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 139–150.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J. and Volgyesi, P. (2001). The Generic Modeling Environment, *Workshop on Intelligent Signal Processing, Budapest, Hungary*, Vol. 17.

Leon-Soto, E., Madrigal-Mora, C., Hahn, C., Warwas, S. and Fischer, K. (2009). A modular protocol model for PIM4Agents, *Proceedings of the Agent-based Technologies and Applications for Enterprise InterOPerability (ATOP 2009) at AAMAS 2009*.

Leszczyna, R. (2004). Evaluation of agent platforms, *Technical report*, European Commission, Joint Research Centre, Institute for the Protection and security of the citizen, Ispra, Italy.

Lilius, J. and Porres Paltor, I. (1999a). Formalising UML state machines for model checking, *Proceedings of the Unified Modeling Language (UML'99)*, Vol. 1723 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 430–445.

Lilius, J. and Porres Paltor, I. (1999b). vUML: A tool for verifying UML models, *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, IEEE Computer Society, pp. 255–258.

Lin, C.-E., Kavi, K. M., Sheldon, F. T., Daley, K. M. and Abercrombie, R. K. (2007). A methodology to evaluate agent oriented software engineering techniques, *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS '07)*, IEEE Computer Society, Washington, DC, USA, p. 60.

Lind, J. (2001). *Iterative Software Engineering for Multiagent Systems - The Massive Method*, Vol. 1994 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Louridas, P. (2008). Orchestrating Web services with BPEL, *IEEE Software* **25**: 85–87.

Luck, M. and Gómez-Sanz, J. J. (eds) (2009). *Agent-Oriented Software Engineering IX, 9th International Workshop, AOSE 2008, Estoril, Portugal, May 12-13, 2008, Revised Selected Papers*, Vol. 5386 of *Lecture Notes in Computer Science*, Springer Verlag.

Luck, M. and Padgham, L. (eds) (2008). *Agent-Oriented Software Engineering VIII, 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, Vol. 4951 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Luck, M., McBurney, P. and Gonzalez-Palacios, J. (2006). Agent-based computing and programming of agent systems, *Proceedings of Programming Multi-Agent Systems, Third International Workshop, ProMAS 2005, Utrecht, The Netherlands, July 26, 2005, Revised and Invited Papers*, Vol. 3862 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 23–37.

Luck, M., P, M., Shehory, O. and Willmott, S. (2005). *Agent Technology Roadmap: A Roadmap for Agent Based Computing*, AgentLink. Electronically available, http://www.agentlink.org/roadmap/al3rm.pdf.

Madrigal-Mora, C., León-Soto, E. and Fischer, K. (2008). Implementing organisations in JADE, *in* R. Bergmann, G. Lindemann, S. Kirn and M. Pechoucek (eds), *In Proceedings of Multiagent System Technologies, 6th German Conference, MATES 2008, Kaiserslautern, Germany, September 23-26, 2008*, Vol. 5244 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 135–146.

Maes, P. (1989). The dynamics of action selection, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers, San Mateo, CA, USA, pp. 991–997.

Maes, P. (1995). Artificial life meets entertainment: Life like autonomous agents, *Communication of the ACM* **38**(38): 108–114.

Malone, T. and Crowston, K. (1994). The interdisciplinary study of coordination, *ACM Computing Surveys* **26**(1): 87–119.

Mann, S. and Klar, M. (1988). A metamodel for object-oriented statecharts, *Proceedings of the 2nd Workshop on Rigorous Object-Oriented Methods, ROOM 2, University of Bradford, May 1998*.

Marschall, F. and Braun, P. (2003). The bi-directional object-oriented transformation language, *Technical Report TUM-I0307*, Technische Universität München, München, Germany.

Mayer, P., Schroeder, A. and Koch, N. (2008). MDD4SOA: Model-driven service orchestration, *Enterprise Distributed Object Computing Conference, IEEE International*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 203–212.

McBurney, P., Parsons, S. and Wooldridge, M. (2002). Desiderata for agent argumentation protocols, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '02)*, ACM Press, New York, NY, USA, pp. 402–409.

McIlraith, S. and Son, T. (2002). Adopting golog for composition of semantic web services, *Proceedings of the International Conference on knowledge representation and Reasoning (KR2002)*, pp. 482–493.

Mendling, J. (2009). Event-driven process chains (epc), *Metrics for Process Models*, Vol. 6 of *Lecture Notes in Business Information Processing*, Springer Verlag, Heidelberg, pp. 17–57.

Meneguzzi, F. R. and Luck, M. (2008). Leveraging new plans in agentspeak(pl), *in* M. Baldoni, T. C. Son, M. B. van Riemsdijk and M. Winikoff (eds), *Declarative Agent Languages and Technologies VI, 6th International Workshop, DALT 2008, Estoril, Portugal, May 12, 2008, Revised Selected and Invited Papers*, Vol. 5397 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 111–127.

Mens, T., Czarnecki, K. and Gorp, P. V. (2005). A taxonomy of model transformations, *in* J. Bezivin and R. Heckel (eds), *Language Engineering for Model-Driven Software Development*, number 04101 in *Dagstuhl Seminar Proceedings*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

Miles, R. H. (1980). *Macro Organizational Behavior*, Goodyear Publishing, Santa Monica, CA.

Miller, T. and McBurney, P. (2007). A formal semantics for Gaia liveness rules and expressions, *International Journal on Agent-Oriented Software Engineering (IJAOSE)* **1**(3/4): 435–476.

Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press, Cambridge, England.

Mizuta, S. and Huang, R. (2005). Automation of grid service code generation with AndroMDA for GT3, *Proceedings of the 19th International Conference on Advanced Information Networking and Applications (AINA '05)*, IEEE Computer Society, Washington, DC, USA, pp. 417–420.

Mokhati, F., Boudiaf, N., Badri, M. and Badri, L. (2007). Translating AUML diagrams into Maude specifications: A formal verification of agents interaction protocols, *Journal of Object Technology* **6**(4): 77–102.

Molesini, A., Denti, E. and Omicini, A. (2005). MAS meta-models on test: UML vs. OPM in the SODA case study, *in* M. Puechouucek, P. Petta and L. Z. Varga (eds), *Proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'05), Budapest, Hungary, 15–17*, Vol. 3690 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Berlin et al., pp. 163–172.

Molesini, A., Denti, E. and Omicini, A. (2008). From AO methodologies to MAS infrastructures: The SODA case study, *Engineering Societies in the Agents World VIII: 8th International Workshop, ESAW 2007, Athens, Greece, October 22-24, 2007, Revised Selected Papers*, Springer Verlag, Berlin, Heidelberg, pp. 300–317.

Molesini, A., Denti, E., Nardini, E. and Omicini, A. (2009). Situated process engineering for integrating processes from methodologies to infrastructures, *Proceedings of the 2009 ACM symposium on Applied Computing (SAC '09)*, ACM Press, New York, NY, USA, pp. 699–706.

Moraitis, P. and Spanoudakis, N. (2004). Combining Gaia and JADE for multi-agent systems development, *Proceedings of the 17th European Meeting on Cybernetics and Systems Research (EMCSR 2004), Vienna, Austria, April 13 - 16*.

Moraitis, P. and Spanoudakis, N. I. (2006). The Gaia2Jade Process for Multi-Agent Systems Development., *Applied Artificial Intelligence* **20**(2-4): 251–273.

Morandini, M., Nguyen, D. C., Perini, A., Siena, A. and Susi, A. (2007). Tool-supported development with Tropos: The conference management system case study, *in* M. Luck and L. Padgham (eds), *Proceedings of the 8th International Workshop on Agent-Oriented Software Engineering VIII, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, Vol. 4951 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 182–196.

Müller, J. (1996). *The Design of Intelligent Agents: A Layered Approach*, Vol. 1177 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Berlin, Heidelberg, New York.

Müller, J. and Pischel, M. (1993). The Agent Architecture InteRRaP: Concept and Application, *Technical Report RR-93-26*, DFKI Saarbrücken.

Müller, J. P. and Zambonelli, F. (eds) (2006). *Agent-Oriented Software Engineering VI, 6th International Workshop, AOSE 2005, Utrecht, The Netherlands, July 25, 2005. Revised and Invited Papers*, Vol. 3950 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Newcomb, P. (2005). Architecture-driven modernization (ADM), *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE '05)*, IEEE Computer Society, Washington, DC, USA, p. 237.

Newell, A. and Simon, H. (1976). Computer science as empirical enquiry, *Communications of the ACM* **19**(3): 112–126.

Nguyen, C. D., Perini, A. and Tonella, P. (2008). ecat: a tool for automating test cases generation and execution in testing multi-agent systems, *In Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems (AAMAS '08)*, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1669–1670.

Nguyen, X. T. and Kowalczyk, R. (2006). WS2JADE: Integrating Web Service with Jade agents, *Technical Report SUTICT-TR2005.03*, Faculty of Information and Communication Technologies Centre for Intelligent Agents and Multi-Agent Systems.

Nissen, M. E. (2000). Supply chain process and agent design for e-commerce, *33rd Hawaii International Conference on System Sciences* **6**: 6021.

Nunes, I., Cirilo, E., de Lucena, C. J. P., Sudeikat, J., Hahn, C. and Gomez-Sanz, J. J. (2011). A survey on the translation from agent oriented specifications to code, *in* M.-P. Gleizes and J. J. Gomez-Sanz (eds), *Agent-Oriented Software Engineering X: 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers*, Vol. 6038 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin et al., pp. 169–179.

Nunes, I., Kulesza, U., Nunes, C. and de Lucena, C. J. P. (2009). A domain engineering process for developing multi-agent systems product lines, *in* C. Sierra, C. Castelfranchi, K. S. Decker and J. S. Sichman (eds), *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*, IFAAMAS, Richland, SC, pp. 1339–1340.

Object Management Group (1999). Requirements for UML Profiles, 1.0 edn.

Object Management Group (2003a). MDA Guide Version 1.0.1, Document omg/03-06-01, June 2003, http://www.omg.org/docs/omg/03-06-01.pdf.

Object Management Group (2003b). UML 2.0 Superstructure Specification, Document ptc/03-08-02, August 2003, http://www.omg.org/docs/ptc/03-08-02.pdf.

Object Management Group (2004). Meta Object Facility (MOF) 2.0 Core Specification, Document ptc/04-10-15, October 2004, http://www.omg.org/docs/ptc/04-10-15.pdf.

Object Management Group (2005). *Ontology Definition Metamodel Third Revised Submission to OMG/ RFP ad/2003-03-40*, Object Modeling Group.

Object Management Group (2006). Business process modeling notation specification, OMG final adopted specification.

Object Management Group (2007). Software & systems process engineering meta-model, v2.0 (spem 2.0), *Technical report*, Object Management Group (OMG).

Object Management Group (2008a). Meta Object Facility (MOF) 2.0 Query/View/Transformation, *Specification Version 1.0*, Object Management Group.

Object Management Group (2008b). Service oriented architecture Modeling Language (SoaML)–Specification for the UML Profile and Metamodel for Services (UPMS). Revised Submission, Finalization Task Force beta 2 document, OMG document ad/2008-11-01; available at: http://www.omg.org/spec/SoaML/20091101.

Object Management Group (2008c). Software & systems process engineering meta-model specification, version 2.0 (SPEM 2.0). OMG document number formal/2008-04-01, available at: http://www.omg.org/spec/spem/2.0/pdf.

Object Management Group (2009a). Agent Metamodel and Profile (AMP), OMG Initial Submission, OMG document: ad/2009-08-04.

Object Management Group (2009b). Service oriented architecture Modeling Language (SoaML)–Specification for the UML Profile and Metamodel for Services (UPMS). OMG Adopted Specification, Finalization Task Force beta 2 document, OMG document number: ptc/2009-12-09; available at: http://www.omg.org/spec/SoaML/20091101.

Odell, J. (2002). Objects and agents compared, *Journal of Object Technology* **1**(2): 41–53.

Odell, J. (2007). Agents: A necessary ingredient in today's highly collaborative world, *Business Technology Trends & Impacts, Council Opinion*.

Odell, J., Giorgini, P. and Müller, J. P. (eds) (2004). *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, Vol. 3382 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Odell, J., Nodine, M. H. and Levy, R. (2005). A metamodel for agents, roles, and groups, *in* J. Odell, P. Giorgini and J. P. Müller (eds), *Proceedings of the 5th International Workshop on Agent-Oriented Software Engineering (AOSE '04)*, Vol. 3382 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 78–92.

Odell, J., Parunak, H. and Bauer, B. (2000). Extending UML for agents, *in* G. Wagner, I. Lesperance and E. Yu (eds), *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pp. 3–17.

Odell, J., Parunak, H. V. D. and Fleischer, M. (2002). The role of roles in designing effective agent organizations, *in* A. F. Garcia, C. J. P. de Lucena, F. Zambonelli, A. Omicini and J. Castro (eds), *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications*, pp. 27–38.

Oldevik, J. (2006). MOFScript Eclipse plug-in: Metamodel-based code generation, *Eclipse Technology eXchange workshop (eTX) at the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*.

Oldevik, J., Neple, T., Gronmo, R., Aagedal, J. O. and Berre, A.-J. (2005). Toward standardised model to text transformations, *in* A. Hartman and D. Kreische (eds), *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings*, Vol. 3748 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 239–253.

Oluyomi, A. (2006). *Patterns and protocols for agent oriented software development*, PhD thesis, The University of Melbourne.

Omicini, A. (2001). SODA: societies and infrastructures in the analysis and design of agent-based systems, *First International Workshop on Agent-oriented Software Engineering (AOSE '00)*, Springer Verlag, Secaucus, NJ, USA, pp. 185–193.

Padgham, L. and Liu, W. (2007). Internet collaboration and service composition as a loose form of teamwork, *Journal of Network and Computer Applications* **30**(3): 1116–1135.

Padgham, L. and Luck, M. (2007). Introduction to AOSE tools for the conference management system, *in* M. Luck and L. Padgham (eds), *Proceedings of the 8th International Workshop on Agent-Oriented Software Engineering VIII, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, Vol. 4951 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 164–167.

Padgham, L. and Winikoff, M. (2002a). Prometheus: a methodology for developing intelligent agents, *Proceedings of the First International Joint Conference on Autonomous Agents & Multi-agent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy,*, ACM Press, New York, NY, USA, pp. 37–38.

Padgham, L. and Winikoff, M. (2002b). Prometheus: a pragmatic methodology for engineering intelligent agents, *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pp. 97–108.

Padgham, L. and Zambonelli, F. (eds) (2007). *Agent-Oriented Software Engineering VII, 7th International Workshop, AOSE 2006, Hakodate, Japan, May 8, 2006, Revised and Invited Papers*, Vol. 4405 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Padgham, L., Thangarajah, J. and Winikoff, M. (2007a). AUML protocols and code generation in the Prometheus design tool, *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '07)*, ACM Press, New York, pp. 1–2.

Padgham, L., Thangarajah, J. and Winikoff, M. (2007b). The Prometheus design tool - a conference management system case study, *in* M. Luck and L. Padgham (eds), *Proceedings of the 8th International Workshop on Agent-Oriented Software Engineering VIII, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*, Vol. 4951 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 197–211.

Padgham, L., Winikoff, M., DeLoach, S. and Cossentino, M. (2008). A unified graphical notation for AOSE, *in* M. Luck and J. Gomez-Sanz (eds), *Proceedings of the Ninth International Workshop on Agent-Oriented Software Engineering (AOSE-2008) at the Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2008), Estoril, Portugal. May, 2008*, pp. 116–130.

Panzarasa, P. and Jennings, N. (2001). The organization of sociality: A manifest for a new science of multi-agent systems, *Proceedings of the 10th Workshop on Multi-agent Systems (MAAMAW'01), Annecy, France*, Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin et al.

Papasimeon, M. and Heinze, C. (2001). Extending the UML for designing JACK agents, *Proceedings of the 13th Australian Software Engineering Conference (ASWEC 01)*, IEEE Computer Society, Washington, DC, USA, p. 89.

Papazoglou, M. and Georgakopoulos, D. (2003). Service-oriented computing, *Communications of the ACM* **46**(10): 25–28.

Parunak, H. and Odell, J. (2002). Representing social structure in UML, *Agent-Oriented Software Engineering II*, Vol. 2222/2002 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 1–16.

Parunak, H. V. D. (1997). Go to the ant: Engineering principles from natural agent systems, *Annals of Operations Research* **75**: 69–101.

Patrascoiu, O. (2004). YATL: Yet another transformation language, *Proceedings of First European Workshop MDA (MDA-IA), University of Twente, the Nederlands, January 2004*, pp. 83–90.

Pavón, J., Gómez-Sanz, J. J. and Fuentes-Fernández, R. (2005). The INGENIAS methodology and tools, *in* B. Henderson-Sellers and P. Giorgini (eds), *Agent-Oriented Methodologies*, Idea Group Publishing, article IX, pp. 236–276.

Pavón, J. and Jorge (2003). Agent oriented software engineering with INGENIAS, *in* V. Marik, J. Müller and M. Pechoucek (eds), *Multi-Agent Systems and Applications III*, Vol. 2691 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 394–403.

Pavón, J., Gómez-Sanz, J. J. and Fuentes, R. (2006). Model driven development of multi-agent systems, *in* A. Rensink and J. Warmer (eds), *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, Vol. 4066 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 284–298.

Payne, T. R. (2008). Web Services from an agent perspective, *IEEE Intelligent Systems* **23**(2): 12–14.

Pe na, J., Hinchey, M. G., Resinas, M., Sterritt, R. and Rash, J. L. (2007). Designing and managing evolving systems using a MAS product line approach, *Sci. Comput. Program.* **66**(1): 71–86.

Penserini, L., Perini, A., Susi, A. and Mylopoulos, J. (2006). From stakeholder intentions to software agent implementations, *in* E. Dubois and K. Pohl (eds), *Proceedings of the Advanced Information Systems Engineering, 18th International Conference (CAiSE'06)*, Vol. 4001 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 465–479.

Penserini, L., Perini, A., Susi, A. and Mylopoulos, J. (2007). High variability design for software agents: Extending Tropos, *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* **2**(4): 16.

Peres, J. and Bergmann, U. (2005). Experiencing AUML for MAS modeling: A critical view, *Proceedings of Software Engineering for Agent-Oriented Systems (SEAS)*, pp. 11–20.

Perini, A. and Susi, A. (2005). Automating Model Transformations in Agent-Oriented modelling, *Agent-Oriented Software Engineering (AOSE-2005)*, Springer Verlag, Berlin et al., pp. 167–178.

Peyravi, F. and Taghyareh, F. (2007). Applying MAS-CommonKADS methodology in knowledge management problem in call centers, *Proceedings of the 25th conference on IASTED International Multi-Conference (SE'07)*, ACTA Press, Anaheim, CA, USA, pp. 99–104.

Pfeffer, J. and Salancik, G. (2003). *The External Control of Organizations: A Resource Dependence Perspective*, new edition edn, Stanford Business Books, Stanford, CA.

Picard, G. and Gleizes, M.-P. (2004). The ADELFE methodology, *in* F. Bergenti, M.-P. Gleizes and F. Zambonelli (eds), *Methodologies and Software Engineering for Agent Systems*, Kluwer Academic Press, pp. 157–176.

Plotkin, G. (2004). The origins of structural operational semantics, *Journal of Logic and Algebraic Programming* **60–61**: 3–15.

Pokahr, A., Braubach, L. and Lamersdorf, W. (2005a). A BDI architecture for goal deliberation, *in* F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh and Wooldridge (eds), *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, ACM Press, New York, NY, USA, pp. 1295–1296.

Pokahr, A., Braubach, L. and Lamersdorf, W. (2005b). Jadex: A BDI reasoning engine, *in* R. Bordini, M. Dastani, D. J. and A. El Fallah Seghrouchni (eds), *Multi-Agent Programming: Languages, Platforms and Applications*, Vol. 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Springer Verlag, Berlin, pp. 149–174.

Poole, J. and Mellor, D. (2001). *Common Warehouse Metamodel: An Introduction to the Standard for Data Warehouse Integration*, John Wiley & Sons, Inc., New York, NY, USA.

Poslad, S., Buckle, P. and Hadingham, R. (2000). The FIPA-OS agent platform: Open source for open standards, *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, pp. 355–368.

Quatrani, T. (2000). *Visual modeling with Rational Rose 2000 and UML (2nd ed.)*, Addison-Wesley Longman Ltd., Essex, UK, UK.

Quenum, J. G., Aknine, S., Briot, J.-P. and Honiden, S. (2006). A modeling framework for generic agent interaction protocols, *Declarative Agent Languages and Technologies IV*, Vol. 4327 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 207–224.

Raber, D. (2009). *A model-driven approach for the integration of multiagent systems and service-oriented architectures in the steel industry*, Master's thesis, Universität des Saarlandes.

Radziah, M., Safaai, D. and Hany, A. H. (2006). MaSE2Jadex: a roadmap to engineer JADEX agents from MaSE methodology, *International Journal of Intelligent Systems and Technologies* **1**(3): 245–251.

Rahwan, I., Juan, T. and Sterling, L. (2006). Integrating social modelling and agent interaction through goal-oriented analysis, *Computer Systems Science Engineering*.

Rahwan, I., Sonenburg, L., Jennings, N. and McBurney, P. (2007). Stratum: A methodology for designing heuristic agent negotiation strategies., *International Journal of Applied Artificial Intelligence* **21**(6): 489–527.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language, *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '96)*, Lecture Notes in Physics, Springer Verlag, Secaucus, NJ, USA, pp. 42–55.

Rao, A. S. and Georgeff, M. P. (1991). Modeling agents within a BDI-architecture, *in* R. Fikes and E. Sandewall (eds), *KR'91*, Morgan Kaufmann, Cambridge, Mass., pp. 473–484.

Rao, A. S. and Georgeff, M. P. (1995). BDI-agents: from theory to practice, *in* V. Lesser (ed.), *Proceedings of the First Intl. Conference on Multiagent Systems*, AAAI Press/The MIT Press, San Francisco, pp. 312–319.

Rimassa, G., Greenwood, D. and Kernland, M. E. (2006). The Living Systems Technology Suite: An autonomous middleware for autonomic computing, *Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS '06), 16-21 July 2006, Silicon Valley, California, USA*, IEEE Computer Society, Washington, DC, USA, p. 33.

Roe, D., Broda, K., Russo, A. and Russo, R. (2003). Mapping UML models incorporating OCL constraints into Object-Z, *Technical Report 2003/9*, Imperial College, 180 Queen's Gate, London, 2002.

Rougemaille, S., Arcangeli, J.-P., Gleizes, M.-P. and Migeon, F. (2008). ADELFE design, AMAS-ML in action, *International Workshop on Engineering Societies in the Agents World (ESAW), Saint-Etienne*, Springer Verlag, Berlin et al., p. electronic medium.

Royce, W. W. (1987). Managing the development of large software systems: concepts and techniques, *Proceedings of the 9th international conference on Software Engineering (ICSE '87)*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 328–338.

Rupert, M., Hassas, S., Li, C. and Sherwood, J. (2007). Simulation of online communities using MAS social and spatial organisations, *International Journal of Information Technology* **4**(3): 183–188.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence, A Modern Approach*, 1st edn, Prentice Hall, Englewood Cliffs, New Jersey.

Sadovykh, A., Hahn, C., Panfilenko, D., Shafiq, O. and Limyr, A. (2009). SOA and SHA tools developed in SHAPE project, *in* R. Vogel (ed.), *Fifth European Conference on Model-Driven Architecture Foundations and Applications: Proceedings of the Tools and Consultancy Track. European Conference on Model-Driven Architecture (ECMDA-09), in Conjunction with Fifth European Conference on Model-Driven Architecture Foundations and Applications, June 23-26, Enschede, Netherlands*, Vol. 09-12 of *CTIT Proceedings Series, WP*, University of Twente, Enschede, University of Twente, Enschede, The Netherlands, p. 113.

Savarimuthu, B., Purvis, M., Purvis, M. and Cranefield, S. (2005). Integrating Web services with agent based workflow management system (WfMS), *IEEE/WIC/ACM International Conference on Web Intelligence*, IEEE Computer Society, Washington, DC, USA, pp. 471–474.

Schäfer, T., Knapp, A. and Merz, S. (2001). Model checking UML state machines and collaborations, *Electronic Notes in Theoretical Computer Science* **55**(3): 357–369.

Schillo, M. (2004). *Multiagent Robustness: Autonomy vs. Organisation*, PhD thesis, Universität des Saarlandes.

Schillo, M., Kray, C. and Fischer, K. (2002). The eager bidder problem: a fundamental problem of DAI and selected solutions, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems (AAMAS '02)*, ACM Press, New York, NY, USA, pp. 599–606.

Schmidt, D. (1995). Using design patterns to develop reusable object-oriented communication software, *Communications of the ACM* **38**(10): 65–74.

Searle, J. R. (1969). *Speech Acts: An Essay in the Philosophy of Language*, Cambridge University Press, Cambridge.

Serrano, J. M. and Ossowski, S. (2004). On the impact of Agent Communication Languages on the implementation of agent systems, *Proceedings of the Eight International Workshop CIA 2004 on Cooperative Information Agents*, Vol. 3191 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 92–106.

Shehory, O. and Sturm, A. (2001). Evaluation of modeling techniques for agent-based systems, *Proceedings of the fifth international conference on Autonomous agents (AGENTS '01)*, ACM Press, New York, NY, USA, pp. 624–631.

Shen, W., Compton, K. and Huggins, J. K. (2002). A toolset for supporting UML static and dynamic model checking, *Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC 2002)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 147–152.

Sheng, Q., Benatallah, B., Dumas, M. and Mak, E. (2002). Self-serv: A platform for rapid composition of Web services in a peer-to-peer environment, *Proceedings of the 28th International Conference on Very Large Databases*, pp. 1051–1054.

Shoham, Y. (1993). Agent-oriented programming, *Artificial Intelligence* **60**(1): 51–92.

Shoham, Y. (1997). An overview of agent-oriented programming, *Software agents*, MIT Press, Cambridge, MA, USA, pp. 271–290.

Singh, M. and Huhns, M. (2005). *Service Oriented Architecture: Semantics, Processes, Agents*, Wiley John & Sons, Inc., Chichster, West Sussex, UK.

Smith, D. C., Cypher, A. and Spohrer, J. (1994). KidSim: Programming agents without a programming language, *Communications of the ACM* **37**(3): 55–67.

Smith, G. (2000). *The Object-Z Specification Language*, Vol. 1 of *Advances in Formal Methods*, Kluwer Academic Publishers, Norwell, MA, USA.

Smith, G. and Hayes, I. J. (2000). Structuring real-time Object-Z specifications, *Proceedings of the Second International Conference on Integrated Formal Methods*, Vol. 1945 of *Lecture Notes In Computer Science*, Springer Verlag, London, pp. 97–115.

Smith, R. G. (1988). The contract net protocol: high-level communication and control in a distributed problem solver, *Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, San Mateo, CA, USA, pp. 357–366.

Spanoudakis, N. and Moraitis, P. (2007). The Agent SystEms MEthodology (ASEME): A preliminary report, *Proceedings of the 5th European Workshop on Multi-Agent Systems (EUMAS'07), Hammamet, Tunisia, 2007*.

Spanoudakis, N. and Moraitis, P. (2009). Gaia agents implementation through models transformation, *Principles of Practice in Multi-Agent Systems, 12th International Conference on Principles of Practice in Multi-Agent Systems (PRIMA'09), Nagoya, Japan, 2009*, Vol. 5925/2009 of *Lecture Notes in Computer Science*, Berlin et al., pp. 127–142.

Spanoudakis, N. I. and Moraitis, P. (2008a). The agent modeling language (AMOLA), *in* D. Dochev, M. Pistore and P. Traverso (eds), *Proceedings of the Artificial Intelligence: Methodology, Systems, and Applications, 13th International Conference, AIMSA 2008, Varna, Bulgaria, September 4-6, 2008*, Vol. 5253 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 32–44.

Spanoudakis, N. I. and Moraitis, P. (2008b). An agent modeling language implementing protocols through capabilities, *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Sydney, NSW, Australia, December 9-12, 2008*, IEEE Computer Society, Washington, DC, USA, pp. 578–582.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual*, 2nd edn, Prentice Hall International Series in Computer Science.

Stormer, H. and Knorr, K. (2001). AWA - Eine Architektur eines agenten-basierten Workflow-Systems, *in* H. H. Buhl, A. Huther and B. Reitwiesner (eds), *Tagungsband 5. Internationale Tagung Wirtschaftsinformatik (WI 2001)*, pp. 147–160.

Stoy, J. E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, USA.

Sudeikat, J., Braubach, L., Pokahr, A. and Lamersdorf, W. (2004). Evaluation of agent-oriented software methodologies - examination of the gap between modeling and platform, *in* P. Giorgini, J. P. Müller and J. Odell (eds), *Agent-Oriented Software Engineering V, Fifth International Workshop AOSE 2004*, Springer Verlag, Berlin et al., pp. 126–141.

Susi, A., Perini, A., Giorgini, P. and Mylopoulos, J. (2005). The Tropos metamodel and its use, *Informatica* **29**(4): 401–408.

Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., Wylie, H. and Yusuf, L. (2005). *Patterns: Model-Driven Development Using IBM Rational Software Architect*, IBM Corp., Riverton, NJ, USA.

Sztipanovits, J. and Karsai, G. (1997). Model-integrated computing, *Computer* **30**(4): 110–111.

Taveter, K. A. (2004). *A Multi-Perspective Methodology for Agent-. Oriented Business Modelling and Simulation*, PhD thesis, Tallinn University of Technology.

Taveter, K. and Wagner, G. (2008). Agent-oriented modeling and simulation of distributed manufacturing, *in* J. Rennard (ed.), *Handbook of Research on Nature-Inspired Computing for Economics and Management*, Idea Group Reference, pp. 527–540.

Thangarajah, J., Padgham, L. and Winikoff, M. (2005). Prometheus design tool, *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, ACM Press, New York, NY, USA, pp. 127–128.

Tran, Q.-N. N. and Low, G. C. (2005). Comparison of ten agent-oriented methodologies, *Agent-Oriented Methodologies*, Idea Group, chapter XII, pp. 341–367.

Tratt, L. (2005). Model transformations and tool integration, *Software and System Modeling* **2**(4): 112–122.

Trencansky, I. and Cervenka, R. (2005). Agent modeling language (AML): A comprehensive approach to modeling MAS, *Informatica* **29**(4): 391–400.

Tveit, A. (2001). A survey of agent-oriented software engineering, *Proceedings of the first NTNU Computer Science Graduate Student (CSGS) Conference (http://www.amundt.org)*.

van Deursen, A., Klint, P. and Visser, J. (2000). Domain-specific languages: an annotated bibliography, *SIGPLAN Not.* **35**(6): 26–36.

van Deursen, A., Visser, E. and Warmer, J. (2007). Model-driven software evolution: A research agenda, *Technical Report TUD-SERG-2007-006*, Delft University of Technology, Software Engineering Research Group.

Verhagen, H. (2000). *Norm Autonomous Agents*, PhD thesis, Stockholm University.

Vilkomir, S., Ghose, A. and Krishna, A. (2004). Combining agent-oriented conceptual modeling with formal methods, *Proceedings of the Australian Software Engineering Conference (2004)*, pp. 147–155.

Wagner, G. (2002). A UML profile for external AOR models, *in* F. Giunchiglia, J. Odell and G. Weiß (eds), *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, Vol. 2585 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 138–149.

Wagner, G. (2003). The Agent-Object-Relationship meta-model: Towards a unified view of state and behavior, *Information Systems* **28**(5): 475–504.

Wagner, G. and Taveter, K. (2004). Towards radical agent-oriented software engineering processes based on AOR modeling, *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '04)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 509–512.

Walton, D. N. and Krabbe, E. C. W. (1995). *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*, SUNY Press, Albany NY, USA.

Wang, H. H., Gibbins, N., Payne, T., Saleh, A. and Sun, J. (2007a). A formal semantic model of the Semantic Web Service Ontology (WSMO), *in* J. S. Dong and J. Sun (eds), *Twelfth IEEE International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society Press, Washington, DC, USA, pp. 111–120.

Wang, H. H., Saleh, A., Payne, T. and Gibbins, N. (2007b). Formal specification of OWL-S with Object-Z: the static aspect, *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI '07)*, IEEE Publisher Society, Washington, DC, USA, pp. 431–434.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Warwas, S. (2007). *Model-driven approach for the integration of service-oriented architectures with agent technology*, Master's thesis, Universität des Saarlandes.

Warwas, S. and Hahn, C. (2008). The concrete syntax of the platform independent modeling language for multiagent systems, *Proceedings of the Agent-based Technologies and Applications for Enterprise InterOPerability (ATOP 2008) at AAMAS 2008*.

Warwas, S. and Hahn, C. (2009a). The DSML4MAS development environment, *in* C. Sierra, C. Castelfranchi, K. S. Decker and J. S. Sichman (eds), *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009*, Vol. 2, IFAAMAS, Richland, SC, pp. 1379–1380.

Warwas, S. and Hahn, C. (2009b). The platform independent modeling language for multiagent systems, *Agent-Based Technologies and Applications for Enterprise Interoperability*, Vol. 25 of *Lecture Notes in Business Information Processing*, Springer Verlag, Berlin et al., pp. 129–153.

Warwas, S., Hahn, C. and Fischer, K. (2009). A visual development environment for Jade, *in* C. Sierra, C. Castelfranchi, K. S. Decker and J. S. Sichman (eds), *Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009*, Vol. 2, IFAAMAS, Richland, SC, pp. 1349–1350.

Watt, D. A. (1990). *Programming language concepts and paradigms*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T. and Ferguson, D. F. (2005). *Web Service Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Weiss, G. (ed.) (1999). *Multiagent systems: A modern approach to distributed artificial intelligence*, MIT Press, Cambridge, MA, USA.

Weyns, D., Omicini, A. and Odell, J. (2007). Environment as a first class abstraction in multiagent systems, *International Journal on Autonomous Agents and Multi-Agent Systems (JAAMAS)* **14**(1): 5–30.

White, T. (2000). *SynthECA: A Synthetic Ecology of Chemical Agents*, PhD thesis, Carleton University.

Winikoff, M. (2005). Towards making Agent UML practical: A textual notation and a tool, *Proceedings of the Fifth International Conference on Quality Software (QSIC '05)*, IEEE Computer Society, Washington, DC, USA, pp. 401–412.

Wood, M. F. and DeLoach, S. A. (2001). An overview of the multiagent systems engineering methodology, *First International Workshop on Agent-oriented Software Engineering (AOSE 2000)*, Springer Verlag, Berlin et al., pp. 207–221.

Woodcock, J. and Davies, J. (1996). *Using Z: Specification, Refinement, and Proof*, Prentice-Hall International, Upper Saddle River, NJ, USA.

Wooldridge, M. (1997). Agent-based software engineering, *IEEE Proc Software Engineering* **144(1)**: 26–37.

Wooldridge, M. (2000a). Intelligent agents, *in* G. Weiss (ed.), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, MIT Press, pp. 27–77.

Wooldridge, M. (2000b). *Reasoning about Rational Agents*, MIT Press, Cambridge, Massachusetts.

Wooldridge, M. and Jennings, N. (1995a). Agent theories, architectures, and languages: A survey, *in* M. Wooldridge and N. Jennings (eds), *Intelligent Agents*, Vol. 890 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 1–39.

Wooldridge, M. J. and Jennings, N. R. (1995b). Intelligent Agents: Theory and Practice, *Knowledge Engineering Review* **10**(2): 115–152.

Wooldridge, M. J., Weiss, G. and Ciancarini, P. (eds) (2002). *Agent-Oriented Software Engineering II, Second International Workshop, AOSE 2001, Montreal, Canada, May 29, 2001, Revised Papers and Invited Contributions*, Vol. 2222 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al.

Wooldridge, M., Jennings, N. and Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design, *International Journal on Autonomous Agents and Multi-Agent Systems (JAAMAS)* **3**(3): 285–312.

Worldwide Web Consortium (2004). Standard glossary of software engineering.

Xiao, L. and Greer, D. (2005). Modeling, auto-generation and adaptation of multi-agent systems, *in* T. Halpin, K. Siau and J. Krogstie (eds), *Proceedings of the Workshop on Evaluating Modeling Methods for Systems Analysis and Design (EMMSAD'05), held in conjunction with the 17th Conference on Advanced Information Systems (CAiSE'05), Porto, Portugal, EU*, FEUP, Porto, Portugal, pp. 605–616.

Xiao, L. and Greer, D. (2007). Towards agent-oriented model-driven architecture, *European Journal of Information Systems* **16**(4): 390–406.

Xu, H. and Shatz, S. M. (2003). ADK: An agent development kit based on a formal design model for multi-agent systems, *Automated Software Engineering* **10**(4): 337–365.

Xu, H. and Zhang, X. (2005). A methodology for role-based modeling of open multi-agent software systems, *in* C.-S. Chen, J. Filipe, I. Seruca and J. Cordeiro (eds), *ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, May 25-28, 2005*, pp. 246–253.

Xueguang, C. and Haigang, S. (2004). Further extensions of FIPA contract net protocol: threshold plus DoA, *Proceedings of the 2004 ACM symposium on Applied computing (SAC '04)*, ACM Press, New York, NY, USA, pp. 45–51.

Yan, Q., jun Shan, L., jun Mao, X. and chang Qi, Z. (2003). RoMAS: A role-based modeling method for multi-agent system, *in* J. P. Li, J. Zhao, J. Liu and N. Z. J. Yen (eds), *Proceedings of International Conference on Active Media Technology 2003*, World Scientific Publishing, pp. 156–161.

Yu, E. (1995). *Modeling Strategic Relationships for Process Reengineering*, PhD thesis, Department of Computer Science, University of Toronto.

Zachman, J. A. (1987). A framework for information systems architecture, *IBM Systems Journal* **26**(3): 277–293.

Zaha, J. M., Barros, A. P., Dumas, M. and ter Hofstede, A. H. M. (2006a). Let's Dance: A language for service behavior modeling, *in* R. Meersman and Z. Tari (eds), *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part I*, Vol. 4275 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin et al., pp. 145–162.

Zaha, J. M., Dumas, M., ter Hofstede, A., Barros, A. and Decker, G. (2006b). Service interaction modeling: Bridging global and local views, *Proceedings of the Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16-20 October 2006, Hong Kong, China*, IEEE Computer Society, Washington, DC, USA, pp. 45–55.

Zambonelli, F. and Parunak, H. (2002). From design to intentions: Signs of a revolution, *First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy*, ACM Press, New York, NY, USA, pp. 455–456.

Zambonelli, F., Jennings, N. and Wooldridge, M. (2003). Developing multiagent systems: the Gaia methodology, *ACM Transactions on Software Engineering and Methodology* **12**(3): 417–470.

Zambonelli, F., Jennings, N. R. and Wooldridge, M. (2001). Organisational rules as an abstraction for the analysis and design of multi-agent systems, *International Journal of Software Engineering and Knowledge Engineering* **11**(3): 303–328.

Zeng, L., Ngu, A., Benatallah, B. and O'Dell, M. (2001). An agent-based approach for supporting cross-enterprise workflows, *Proceedings of the 12th Australasian Database Conference (Gold Coast, Queensland, Australia, January 29 - February 01, 2001). ACM International Conference Proceeding Series*, Vol. 10, IEEE Computer Society, Washington, DC, pp. 123–130.

Zinnikus, I., Hahn, C. and Fischer, K. (2008a). A model-driven, agent-based approach for a rapid integration of interoperable services, *in* K. Mertins, R. Ruggaber, K. Popplewell and X. Xu (eds), *Enterprise Interoperability III. New Challenges and Approaches*, Springer Verlag, London, pp. 651–663.

Zinnikus, I., Hahn, C. and Fischer, K. (2008b). A model-driven, agent-based approach for the integration of services into a collaborative business process, *in* L. Padgham, D. C. Parkes, J. Müller and S. Parsons (eds), *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008*, Vol. 1, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 241–248.

Zinnikus, I., Hahn, C. and Fischer, K. (2010). Agent-driven semantic interoperability for cross-organisational business processes, *in* G. Mentzas and A. Friesen (eds), *Semantic Enterprise Application Integration for Business Processes: Service-Oriented Frameworks*, IGI Global, pp. 61–89.

Zinnikus, I., Hahn, C., Klein, M. and Fischer, K. (2007). An agent-based, model-driven approach for enabling interoperability in the area of multi-brand vehicle configuration, *in* B. J. Krämer, K.-J. Lin and P. Narasimhan (eds), *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, Vol. 4749 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, pp. 330–341.

Index

Appendix

A. Remaining Object-Z Specification

A.1 Multiagent View

A.1.1 NamedElement

The concept of *NamedElement* (see Fig. 4.2) includes a single attribute called *name* to define the name of the concepts that inherits from *NamedElement* like *Agent*, *Interactions*, *Cooperation*, *Environment* etc.

The Object-Z class schema of *NamedElement* is given in Schema A.1.1. This class schema is rather simple as it only consists of a single variable called *name*.

NamedElement

$name : String$ [Variables]

Schema A.1.1: Class schema of *NamedElement*.

A.2 Agent View

A.2.1 Knowledge

The abstract syntax of *Knowledge* was given in Definition 4.3.3 in Section 4.3.3. The semantics of *Knowledge* is given in Schema A.2.1.

Knowledge

NamedElement

$value : String;\ type :\downarrow Type$ [Variables]

Schema A.2.1: Class schema of *Knowledge*.

A.3 Interaction View

A.3.1 Break

The *Break* concept can be used to leave a *Loop* interaction even if the condition for its end is not fulfilled. A common application is to prematurely end an infinite loop or end a conditional loop before its natural end.

Definition A.3.1 (Break in PIM4AGENTS)

A Break refers to a messageFlow that indicates the MessageFlow that needs to be triggered in the case of this Break.

When a *Break* is reached, the current execution immediately exits its innermost surrounding Loop operation and the execution is continued at the *MessageFlow* the *Break* refers to.

A.3.2 TimeOut

The class schema of *TimeOut* is depicted in Schema A.3.2. It includes the variables *mode*, and *messageFlow*. We define the type of *Mode* as follows.

$Mode ::= duration \mid absolute \mid relative$

TimeOut

$mode : Mode;\ messageFlow : \mathbb{P}\ MessageFlow$ [Variables]

Schema A.3.1: Class schema of *TimeOut*.

A.4 Behavioral View

A.4.1 Behavior

The semantics if Behavior is defined through the Object-Z class given in Schema A.4.1.

A.4.2 Parallel

Definition 4.7.7 specifies the abstract syntax of the *Parallel* concepts, its semantics is given in Schema A.4.2.

Behavior

NamedElement

$agent : \mathbb{P}_1\, Agent$ [Variables]

Schema A.4.1: Class schema of *Behavior*.

Parallel

Split

[Variables]

$activeControlFlow = \bigcup \{s : startActivities \bullet s.outFlows\}$ [I1]

Schema A.4.2: Class schema of *Parallel*.

A.4.3 Sequence

A *Sequence* as a specialization of a *StructuredActivity* denotes a list of *Activities* to be executed in a sequential manner as defined by contained *ControlFlows* through their *sink* and *source* attributes. Beside using the concept of a *Sequence* to model the execution of *Activities* in a sequential manner, the particular *Activities* can be linked directly through the *ControlFlow* concept. However, the concept of *Sequence* allows to hide the concrete trace which might be important in the case of very complex *Plans*. The abstract syntax is given in Definition A.4.1.

Definition A.4.1 (Sequence in PIM4AGENTS)

A Sequence is given by a 8-tuple S = (name, steps, flows, condition, localKnowledge, inFlow, outFlow, messageScope)

A *Sequence* is a *StructuredActivity* that executes its contained *Activities* in order. A *Sequence* like any other *Activity* has its own visibility scope. That means that Knowledge related to this *Activity* cannot be accessed from the outside of this *Sequence*.

The construct of *Sequence* allows to explicitly represent the sequential ordering of *Activities*. Its main advantage compared to the direct linking of the *Activities* through *ControlFlows*—which would have the same meaning—is that the information contained within the *Sequence* can be hidden which would make sense in case of complex Plans.

In accordance to Schema A.4.3, for any *Sequence* it must hold that its startActivities have exactly one outgoing *ControlFlow*—in other words, branching within a *Sequence* is forbidden.

A.4.4 Loop

In PIM4AGENTS, a *Loop* is a point in a *Plan* where a set of *Activities* are executed repeatedly until a certain pre-defined *condition* evaluates to false. The *Loop* pattern is a mechanism for allowing

Sequence

StructuredActivity

[Variables]

$\#\{cf : \bigcup\{s : startActivities \bullet s.outFlow\} \mid cf \in ControlFlow\} = 1$ []

$\forall f : flows \mid f \in ControlFlow \bullet f.condition$ []

$condition = \varnothing$ []

Schema A.4.3: Class schema of *Sequence*.

sections of a Plan to be repeated. Is allows looping that is block structured, i.e. pattern may allow one entry and exit point.

Definition A.4.2 (Loop in PIM4AGENTS)

A Loop is defined by a 9-tuple L = (name, steps, flows, condition, localKnowledge, inFlow, outFlow, messageScope, postCondition), where:

- *postCondition: defines the condition*

Loop

StructuredActivity

$postCondition : \mathbb{P}\,\mathbb{B}$ [Variables]

$\#\{cf : \bigcup\{s : startActivities \bullet s.outFlow\} \mid cf \in ControlFlow\} = 1$ []

$condition \neq \varnothing \lor postCondition \neq \varnothing$ []

$completed \Leftrightarrow (\forall e : endActivities \bullet e.completed) \land condition \land postCondition$ []

$\forall cf : flows \mid cf \in ControlFlow \bullet cf.condition$ []

Schema A.4.4: Class schema of *Loop*.

A.4.5 ParallelLoop

The abstract syntax of *ParallelLoop* is given by Definition 4.7.9. Its corresponding semantics is expressed through Schema A.4.5.

A.4.6 Task

The semantics of *Task* is expressed in Schema A.4.6.

ParallelLoop

Split

[Variables]

$\#\bigcup\{s : startActivities \bullet s.outFlow\} = activeControlFlows = 1$ [I1]

Schema A.4.5: Class schema of *ParallelLoop*.

Task

Activity

[Variables]

$controlFlow = \varnothing$ [I1]

Schema A.4.6: Class schema of *Task*.

A.4.7 InternalTask

An *InternalTask* in PIM4AGENTS is used to define platform-specific information on the PIM level. For this purpose, the domain designer may want to specify code inside the PIM design that is translated in an one-to-one fashion between PIM and PSM. The abstract syntax of *InternalTask* is given as follows:

Definition A.4.3 (InternalTask in PIM4AGENTS)

An InternalTask is defined by a 8-tuple I = (name, outFlow, inFlow, localKowledge, messageScope, controlFlow, text, code), where:

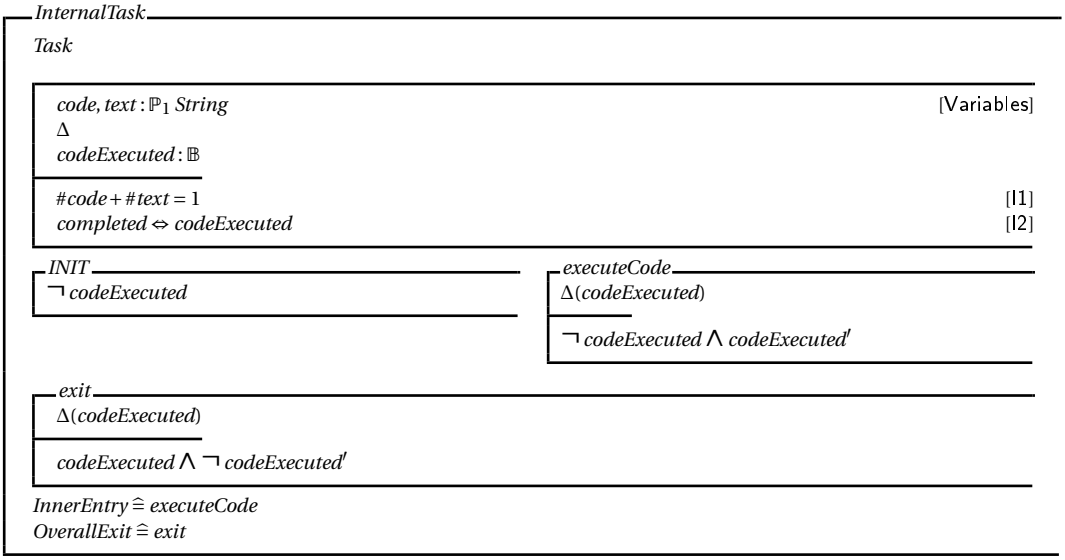
- *name: defines the name of the InternalTask*
- *code: reflects the piece of code that is executed within this InternalTask*
- *text: specifies the piece of text that is given to the outside*

The variables outFlow, inFlow, localKnowledge, messageScope, and controlFlow are defined in accordance to Definition 4.7.10.

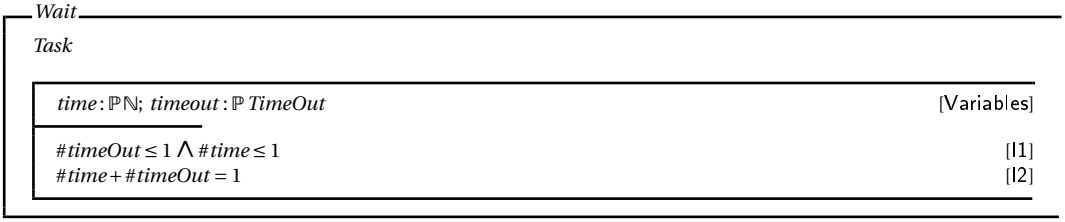
The Object-Z class schema of *InternalTask* is given in Schema A.4.7. Beside the variables formalized by Definition A.4.3, furthermore, the semantics are restricted by Invariant I1 stating that either code or text must be specified by the system designer. The *InternalTask* is completed iff the code/text has been executed (see Invariant I2) which is expressed by the semantic variable *codeExecuted*.

A.4.8 Wait

The *Wait* concept can be used for synchronization of parallel paths. It is formally defined as follows:



Schema A.4.7: Class schema of *InternalTask*.



Schema A.4.8: Class schema of *Wait*.

Definition A.4.4 (Wait in PIM4AGENTS)

A Wait is defined by a 8-tuple W = (name, outFlow, inFlow, localVariable, messageScope, controlFlow, time, timeOut), where:

- *name: defines the name of the Wait*
- *time: refers to an absolute point in time*
- *timeOut: refers to a Protocol's TimeOut that defines the time constraints for this Wait*

The variables outFlow, inFlow, localKnowledge, messageScope, and controlFlow are used as specified in Definition 4.7.10.

The semantics of *Wait* is given in Schema A.4.8. Invariant I1 ensures that both variables, *timeOut* and *time*, consist of at most of one element. Moreover, the sum of both values is set, in Invariant I2, to 1, meaning that either a *time* is defined by the *Wait* concepts or given by the *time* variable of *TimeOut*.

A.4.9 Begin

A *Begin* activity in PIM4AGENTS is used to indicate the begin of a *Plan* or *StructuredActivity*. All *Activities* following the *Begin* activity are immediately instantiated. A *Plan* or *StructuredActivity* must contain only one *Begin* activity and must not have any incoming *ControlFlow* or *InformationFlow*.

Definition A.4.5 (Begin in PIM4AGENTS)

A Begin is defined by a 6-tuple B = (name, inFlow, outFlow, flows, localKnowledge, messageScope)

The formal semantics of *Begin* is expressed in Schema A.4.9.

Begin

Task

[Variables]

Δ [Semantics Variables]

$parentActivity : \downarrow Activity$

$activePaths : \mathbb{P}\, ControlFlow$

$activePaths = \{o : outFlow \mid o \in ControlFlow \wedge (o.condition \vee o.condition = \varnothing\}$ []

$parentActivity = \{a : Activity \mid self \in a.steps\}$ []

$inFlow = \varnothing$ []

$parentActivity \in Decision \wedge parentActivity.executionMode = XOR \Rightarrow \#activePaths = 1$ []

$OverallExit \mathrel{\widehat{=}} exit$

Schema A.4.9: Class schema of *Begin*.

A.4.10 End

In analogy with *Begin*, the *End* activity marks the end of a *Plan* or *StructuredActivity*. The abstract syntax of *End* is given in Definition A.4.6.

Definition A.4.6 (End in PIM4AGENTS)

An End is defined by a 6-tuple E = (name, inFlow, outFlow, flows, localKnowledge, messageScope)

The semantics of End is given in Schema A.4.10. The class schema of End inherits from the schema of *Task*. Furthermore, it includes the semantic variables

The *End* activity can only be used in the context of an *Plan* and *StructuredActivity* that both are of the kind *Activity* and additional containing further *Activities*. The execution within these concepts is finalized after invoking the *End* activity and the execution is returned to the containing state or invoking *Plan*. The calling or owning *Plan* itself is not terminated.

A.4.11 Fail

Apart from the concepts *Begin* and *End*, a *Fail* characterizes an unwanted trace within a *Plan* that denoted which actions (e.g. *Plan*, *Activity*) to take next. The semantics of *Fail* is given in Schema A.4.11.

End

Task

[Variables]

Δ [Semantics Variables]

$count, synchronizationPaths : \mathbb{N}, readyForFinalization : \mathbb{B}$

$parentActivity : \downarrow Activity;\ activePaths = \mathbb{P} \downarrow Activity$

$outFlow = \varnothing$ []

$parentActivity = \{a : Activity \mid self \in a.steps\}$ []

$activePaths = \{a : \{acf : \bigcup \{cf : parentActivity.startActivities \bullet c.outFlows\} \mid$
$acf \in ControlFlow \wedge acf.condition\} \bullet a.sink\}$ []

$parentActivity \in \{Plan, Sequence, Loop, ParallelLoop\} \Rightarrow synchronizationPaths = 1$

$parentActivity \in \{Split\} \wedge parentActicity.synchronizationMode = AND$
$\Rightarrow synchronizationPaths = \#activePaths$ []

$parentActivity \in \{Split\} \wedge parentActicity.synchronizationMode = NofM$
$\Rightarrow synchronizationPaths = n$ []

$readyForFinalization \Leftrightarrow count = synchronizationPaths$ []

$activityDone \Leftrightarrow readyForFinalization$ []

INIT

$count = 0$

enter

$\Delta(count)$

$count' = count + 1$

$InnerEntry \mathrel{\widehat=}$

$OverallExit \mathrel{\widehat=} exit$

Schema A.4.10: Class schema of *End*.

Fail

Task

[Variables]

$InnerEntry \mathrel{\widehat=}$

$OverallExit \mathrel{\widehat=} exit$

Schema A.4.11: Class schema of *Fail*.

A.5 Environment View

A.5.1 Environment

The concept of *Environment* is the core concept of the environment view as each kind of *Resource* (e.g. *Object*) is part of it. The abstract syntax of *Environment* is given in Definition 4.8.1. Its semantics is defined through Schema A.5.1.

Environment

NamedElement

$resource : \mathbb{P}_1 \downarrow Resource©$

$\forall r_1, r_2 : resource \bullet r_1.name = r_2.name \Rightarrow r_1 = r_2$ [I1]

Schema A.5.1: Class schema of *Environment*.

A.5.2 Resource

Resources are the main component of the *Environment* and are used to represent any non-autonomous entity that is part of the *Environment* such as *Ontologies*, *Services*, or *Objects* used in any form by the *Agents* and *Roles*. A *Resource* is defined as follows:

Definition A.5.1 (Resource in PIM4AGENTS)

A Resource is an abstract concept given by its name.

A.5.3 Attribute

As we have seen in the previous section, *Attributes* in PIM4AGENTS present the kind of data and information an *Object* has available. The abstract syntax of *Attribute* is given in Definition A.5.2.

Definition A.5.2 (Attribute in PIM4AGENTS)

An Attribute is given by a 5-tuple A = (name, value, type, multiplicities, typeRelationship), where

- *name: depicts the name of the Attribute*
- *value: defines the current value of the Attribute*
- *type: illustrates the Type the Attribute conforms to*
- *typeRelationship: defines the type of relationship the Attribute has with other Attributes*
- *multiplicities: describes the multiplicity of these relationships*

Each *Attribute* is characterized by a unique name, a type and a value. Furthermore, a relation to other *Objects* called *typeRelationship* of enumeration type *TypeRelationship* is defined. This kind of relationship between *Objects* can be a common reference relationship (i.e. *Association*), shared containment (i.e. *Aggregation*) and unshared containment (i.e. *Composition*)

$TypeRelationship ::= Association \mid Aggregation \mid Composition$

Beside the type of relationship, the system designer may also define the multiplicity of the relationship. Three variants are in general feasible:

$Multiplicities ::= 0..1 \mid 1..* \mid *$

The type of an *Attribute* can either be again the type of a complex *Object* or of the form *PrimitiveType* that is defined as follows:

$PrimitiveTag ::= int \mid boolean \mid string$

Attribute

NamedElement

$multiplicities : Multiplicities$ [Variables]
$typeRelationship : TypeRelationship;\ type : \downarrow Type;\ value : String$

Schema A.5.2: Class schema of *Attribute*.

A.5.4 Operation

An *Operation* in PIM4AGENTS describes the behavior of an *Object* and thus define the potential changes in state that an *Object* may undergo during its lifetime. The abstract syntax of *Object* is given in Definition A.5.3.

Definition A.5.3 (Operation in PIM4AGENTS)

An Operation is given by a triple O = (name, type, parameter), where:

- *name: defines the type of the Operation*
- *type: represents the return type of the Operation*
- *parameter: illustrates the input parameters of the Operation*

Operation

NamedElement

$parameter : \mathbb{P}\, Parameter©;\ type : \downarrow Type$ [Variables]

$\forall p_1, p_2 : ranparameter \bullet p_1.name = p_2.name \Rightarrow p_1 = p_2$ [I1]

Schema A.5.3: Class schema of *Operation*.

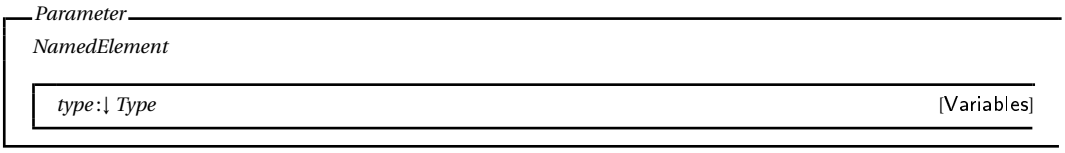
Schema A.5.3 defines the semantics of *Operation*. Beside inheriting from the class schema of *NamedElement* (cf. Schema A.1.1) and including the variables *parameter* and *type* we defined Invariant I1 stating that within an Operation, the parameter names should be unique. The formal definition of *Parameter* is given is Appendix A.5.5.

A.5.5 Parameter

A *Parameter* is considered as the input variables of an *Operation*. Its abstract syntax is given by Definition A.5.4.

Definition A.5.4 (Parameter in PIM4AGENTS)

A Parameter is given by a tuple P = (name, type), where name defines the name of the Parameter and type defining the Type of the Parameter.



Schema A.5.4: Class schema of *Parameter*.

A.6 Deployment View

A.6.1 Membership

The *Membership* concept of PIM4AGENTS defines the *AgentInstances* being member in other *AgentInstances*. However, only *AgentInstances* of type *Organization* contain such *Memberships*. Each *Membership* encapsulates exactly one *AgentInstance* and additionally defines to which *AgentInstance* (of type *Organization*) it actually belongs to. A *Membership* is informally defined as follows:

Definition A.6.1 (Membership in PIM4AGENTS)

A Membership is given by a triple Membership = (name, domainRoleBinding, agentInstance), where:

- *name: defines the name of the this Membership*
- *domainRoleBinding: relates to the kinds of DomainRoleBindings that establish the role bindings of this member*
- *agentInstance: denotes the particular AgentInstance which is represented by this Membership*

A *Membership* may refer to different *DomainRoleBindings* through the variable *domainRoleBinding* to allow the particular *AgentInstances* (i.e. referred by *agentInstance*) playing several DomainRoles at the same time within one and the same *Organization*. The formal semantics of *Membership* is expressed by Schema A.6.1.

Membership

NamedElement

$$domainRoleBinding : \mathbb{P}\ DomainRoleBinding;\ agentInstance : AgentInstance \qquad [\mathsf{Variable}]$$

$$\forall d_1, d_2 : domainRoleBinding \mid d_1 \neq d_2 \bullet$$
$$d_2.roleBinding \notin d_1.roleBinding.conflictsWith \land d_1 \notin d_2.conflictsWith \qquad [\mathsf{I3}]$$

Schema A.6.1: Class schema of *Membership*.