MIAC: Methodology for Intelligent Agents Componentware

vorgelegt von
Diplom-Informatiker
Axel Heßler

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Klaus Obermayer
Berichter: Prof. Dr. Dr. h.c. Sahin Albayrak
Berichter: Prof. Dr. Rainer Unland

Tag der wissenschaftlichen Aussprache: 17. Dezember 2012

Berlin 2013

D 83

Abstract

The DAI-Labor at Technische Universität Berlin has concerned itself with *Agent-Orientation* for more than 15 years. Numerous projects have studied the application of the *Agent-Oriented Programming (AOP)* paradigm to real-world problems in diverse application areas such as *automotive, entertainment, production control, connected living, office* and *software engineering* (as applying research results on the field itself). *Distributed artificial intelligence* and *agent technology* promise to be the key factors to engineering large-scale, distributed and complex software systems and services. They make it easy to understand large problem fields, and provide means to their solution, based on natural, plain and intelligible metaphors.

Unfortunately, the research area of Agent-Orientation is not a homogeneous scientific sector. It shows a high degree of diversification in all aspects of software engineering. For software industry it is not easy to use research results due to an unmanageable and disjoint amount of small solutions, and often numberless distinct answers to the same question. On top of this comes the fact that no or not many products support the industrial software engineer for the development of agent-based software systems and services.

In practice, software engineering is well-understood as can be seen in the software game industry or, as the more recent example, in *social media*. It is now time to choose an integrative approach that picks the best of both worlds and thereby relates agent-oriented solutions to best practices in software engineering. It should enable software engineers to comprehend *when* to use *which* method and technique to understand and solve the *what* in time and budget.

This thesis integrates three keystones of Agent-Oriented Software Engineering (AOSE):

- an agent-oriented framework,
- the agent-oriented methodology
- an agent-oriented and agent-based tool suite.

The result is submitted to an iterative and incremental process itself in order to streamline these building blocks for fast and efficient development of agent-based applications and services.

The methodological approach is directed towards development of applications built on a knowledge-based, service-oriented, multi-agent component architecture. The methodology is named Methodology for Intelligent Agent Componentware (MIAC). MIAC guides you through the process of working out the artifacts which then can be directly implemented, tested and deployed. MIAC is an attempt towards a lean, industrial strength, agent-oriented software-engineering process.

The architecture and its implementation, Java-based Intelligent Agent Componentware (JIAC), has undergone a streamlining process itself. Where JIAC IV has been a full-featured, agent-oriented framework that incorporates many Artificial Intelligence (AI) concepts, the successor JIAC V is a modular architecture that provides a minimal but powerful agent core and infrastructure, and then offers optional plugins that can be used to address different aspects of an agent-based solution.

Parallel to that, developer and management tools support the methodology and complement the framework. Here, the starting point was a full-featured tool suite that supports all development aspects in a 1:1 mapping. While advancing the framework and methodology and building on a standard developer platform, the tool suite itself became more streamlined, modular and easier to use.

Finally, the results of this thesis are products that have been developed, applied and evaluated in projects together with students of the TU Berlin, other Competence Centers of the DAI-Labor and also with partners from different industrial sectors. While many Agent-Oriented (AO) methodologies and tools usually reflect theories and laboratory experiments, the framework, methodology and tools presented in this thesis have developed market-ready prototypes and competitive solutions that have also successfully shown their value in different programming contests.

Zusammenfassung

Das DAI-Labor der Technischen Universität Berlin beschäftigt sich seit über 15 Jahren mit *Agentenorientierung*. Zahlreiche Projekte haben die Anwendung des agentenorientierten Programmierparadigmas auf reale Probleme in verschiedenen Anwendungsgebieten, wie z.B. dem *Automotive-Bereich, der Unterhaltung, der Produktionskontrolle, dem vernetztes Wohnen, dem Büro*, und der *Softwaretechnik* (als Anwendung auf sich selbst), untersucht. *Verteilte künstliche Intelligenz* und *Agententechnologien* sind die Schlüssel für ingenieurmäße Entwicklung von großen, komplexen und verteilten Softwaresystemen und Diensten. Sie vereinfachen das Verständnis unübersichtlicher Probleme, stellen Hilfsmittel zu deren Lösung bereit, und fußen dabei auf natürlichen, einfachen und verständlichen Metaphern.

Leider ist das Forschungsgebiet der Agentenorientierung kein homogener Wissenschaftszweig. Es zeigt ein hohes Maß an Diversifikation in allen Aspekten der Softwaretechnik. Für die Softwareindustrie ist es schwierig, die Forschungsergebnisse zu nutzen, aufgrund der unüberschaubaren Menge an Einzellösungen, wobei man noch aus unzähligen, gleichartigen Lösungsansätzen für ein Problem wählen muss. Dazu kommt, dass es keine oder kaum Produkte gibt, die den Softwareingenieur in der Industrie bei der Entwicklung von agentenbasierten Systemen und Diensten unterstützen.

In der Praxis ist Softwaretechnik eine wohlverstandene Ingenieurswissenschaft, wie man am Beispiel der Spielesoftware-Industrie oder, als neuestes Beispiel, den *Social Media* sehen kann. Es ist jetzt an der Zeit, einen integrativen Ansatz zu erarbeiten, der die besten Aspekte beider Welten auswählt und dabei die agentenorientierten Lösungen mit den bewährten Verfahren der Softwaretechnik verbindet. Das soll dem Softwareingenieur verständlich machen, *wann* er *welche* Methode oder Technik benutzt, um das *Was* zu verstehen und zu lösen; und das in der vorgegebenen Zeit und dem finanziellen Rahmen.

Diese Arbeit integriert drei Grundpfeiler der agentenorientierten Softwareentwicklung:

- ein agentenorientiertes Rahmenwerk

- eine agentenorientierte Methodologie
- einen agentenorientierten und agentenbasierten Werkzeugkasten

Das Ergebnis wird selbst einem interativen und inkrementellen Prozess unterworfen, um diese Bausteine für die schnelle und effiziente Entwicklung von agentenbasierten Anwendungen und Diensten so rationell wie möglich zu gestalten.

Der methodologische Ansatz richtet sich dabei auf die Entwicklung von Anwendungen auf der Basis einer wissensbasierten, dienstorientierten, mehragentischen Komponentenarchitektur. Er heißt *Methodology for Intelligent Agent Componentware (kurz: MIAC)*. MIAC führt durch den Prozess zur Herausarbeitung jener Artefakte, die direkt implementiert, getestet und benutzt werden können. MIAC strebt dabei einen schlanke, in der Industrie akzeptierten, agentenorientierten Softwareentwicklungsprozess an.

Die Architektur und die Implementierung des Rahmenwerks, *Java-based Intelligent Agent Componentware (JIAC)*, unterliegt dabei dem selben Rationalisierungsprozess. JIAC IV ist noch ein umfangreiches, agentenorientiertes Rahmenwerk, welches sehr viele unterschiedliche Konzepte der künstlichen Intelligenz ein sich vereint. Der Nachfolger JIAC V ist eine modulare Architektur, die einen minimalen, aber leistungsfähigen Agentenkern und eine entsprechende Infrastruktur bereitstellt. Dazu bietet JIAC V ein Reihe von Erweiterungen, die optional sind und bestimmte Aspekte einer agentenbasierten Lösung adressieren.

Parallel dazu unterstützen Entwickler- und Administratorwerkzeuge die Methodologie und ergänzen das Rahmenwerk. Der Ausgangspunkt dabei ist ein komplette Werkzeugunterstützung für den gesamten Entwicklungszyklus. Mit der fortschreitenden Entwicklung der Methodologie und des Rahmenwerks, zusammen mit der Benutzung einer Standardplattform für Entwicklerwerkzeuge, konnte die Werkzeugunterstützung selbst rationalisiert und vereinfacht werden.

Die Ergebnisse dieser Arbeit sind Produkte, die in Zusammenarbeit mit Studenten der TU Berlin, anderen Kompetenzzentren des DAI-Labors und mit Partnern aus unterschiedlichen Industriebereichen entwickelt wurden. Normalerweise reflektieren agentenorientierte Methodologien und Werkzeuge theoretische Arbeiten oder Laborexperimente. Mit dem Rahmenwerk, der Methodologie und den Werkzeugen, die in dieser Arbeit vorgestellt werden, wurden bereits produktnahe Prototypen entwickelt, und sie haben ihre Leistungsfähigkeit in verschiedenen Programmierwettbewerben erfolgreich gezeigt.

Danksagung

Die Entwicklung einer Methodologie zur Entwicklung von Software dauert in der Regel mehrere Jahre. Ich danke Prof. Dr. Sahin Albayrak für die langfristige Unterstützung meiner Arbeit.

Ich danke meinen Kollegen Dr. Benjamin Hirsch, Dr. Jan Keiser, Thomas Konnerth, Claus Schenk, Silvan Kaiser, Tobias Küster, Marcel Patzlaff, Tuguldur, Nils Masuch, Marco Lützenberger, Michael Burkhard, Alexander Thiele, Martin Berger, Sebastian Ahrndt und Jakob Tonn für Transpiration und Inspiration.

Ich danke meiner Familie, Ines und Stefan, für die geduldige und beharrliche Unterstützung und meinen Eltern und Schwiegereltern. Ihr seid immer für mich da.

List of publications

This thesis is based on the following research publications:

[**FKH02**] Stefan Fricke, Jan Keiser, and Axel Hessler. *Demo-Storyboard for the JIAC IV Agent Development Environment.* AAMAS Demonstration Session, Bologna, Italy, July 15-19 2002.

[**HHK08**] Axel Hessler, Benjamin Hirsch, and Jan Keiser. *JIAC IV in Multi-Agent Programming Contest 2007.* In: M. Dastani, A. El Fallah Segrouchni, A. Ricci, and M. Winikoff, editors, *ProMAS 2007 Post-Proceedings*, volume 4908 of *LNAI*, pages 262–266. Springer Berlin / Heidelberg, 2008.

[**HHK10**] Axel Heßler, Benjamin Hirsch, and Tobias Küster. *Herding cows with JIAC V. Annals of Mathematics and Artificial Intelligence*, pages 1–15, 2010.

[**HHKA11**] Axel Heßler, Benjamin Hirsch, Tobias Küster, and Sahin Albayrak. *Agentstore — a pragmatic approach to agent reuse.* In *12th International Workshop on Agent-Oriented Software Engineering (AOSE 2011)*, 2011.

[**HKH09**] Benjamin Hirsch, Thomas Konnerth, and Axel Heßler. *Merging Agents and Services — the JIAC Agent Platform.* In: Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 159–185. Springer, 2009.

[**HKHA06**] Benjamin Hirsch, Thomas Konnerth, Axel Hessler, and Sahin Albayrak. *A serviceware framework for designing ambient services.* In: Antonio Maña and Volkmar Lotz, editors, *Developing Ambient Intelligence (AmID'06)*, pages 124–136. Springer Paris, 2006.

[**HKK$^+$09**] Axel Hessler, Jan Keiser, Tobias Küster, Marcel Patzlaff, Alexander Thiele, and Erdene-Ochir Tuguldur. *Herding agents - jiac*

tng in multi-agent programming contest 2008. In: Koen V. Hindriks, Alexander Pokahr, and Sebastian Sardina, editors, *Programming Multi-Agent Systems. 6th International Workshop, ProMAS 2008, Estoril, Portugal, May 13, 2008. Revised Invited and Selected Papers*, volume 5442 of *Lecture Notes in Artificial Intelligence*, pages 228–232. Springer, 2009.

[**HKN+09**] Axel Hessler, Tobias Küster, Oliver Niemann, Aldin Sljivar, and Amir Matallaoui. *Cows and Fences: JIAC V - AC'09 Team Description*. In: Jürgen Dix, Michael Fisher, and Peter Novák, editors, *Proceedings of the 10th International Workshop on Computational Logic in Multi-Agent Systems 2009*, volume IfI-09-08 of *IfI Technical Report Series*. Clausthal University of Technology, 2009.

[**KH08**] Tobias Küster and Axel Heßler. *Towards transformations from BPMN to heterogeneous systems*. In: Massima Mecella and Jian Yang, editors, *BPM2008 Workshop Proceedings*, 2008.

[**KHH09**] Tobias Küster, Axel Heßler, and Benjamin Hirsch. *Modelling and transforming BPMN diagrams with the visual service design tool*. In: *Demo Track at BPM2009*, 2009.

[**KLHH10**] Tobias Küster, Marco Lützenberger, Axel Heßler, and Benjamin Hirsch. *Integrating process modelling into multi-agent system engineering*. In: Michael Huhns, Ryszard Kowalczyk, Zakaria Maamar, Rainer Unland, and Bao Vo, editors, *Proceedings of the 5th Workshop of Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE) 2010*, 2010.

[**LKHH09**] Marco Lützenberger, Tobias Küster, Axel Heßler, and Benjamin Hirsch. *Unifying JIAC agent development with AWE*. In: *Proceedings of the Seventh German Conference on Multiagent System Technologies, Hamburg, Germany*. Springer, 2009.

[**THHA08**] Erdene-Ochir Tuguldur, Axel Heßler, Benjamin Hirsch, and Sahin Albayrak. *Toolipse: An IDE for development of JIAC applications*. In: *Proceedings of PROMAS08: Programming Multi-Agent Systems*, 2008.

Contents

List of Figures

Chapter 1

Introduction

No single language or technology can do it all,
and software development involves the application
of many technologies rather than just one...
(Dave Springgay on Eclipse Perspectives)

1.1 Motivation

Agent-Oriented Software Engineering (AOSE) has been the focus of research for some time now. While during the second half of the 20th century Artificial Intelligence (AI) [MMRS55] and later Distributed Artificial Intelligence (DAI) [BG88, OJ96] have worked with concepts of agents, it has been only recently that Agent-Oriented Programming (AOP) and AOSE became a research area by its own right.

Initially, researchers focussed on methods and formalisms to describe human behaviour and thought processes, which found its highlight in an agent model based on the concepts of Belief Desire Intention (BDI) (see Rao and Georgeff [RG91]). In this model the agent makes assumptions about the world, the *Beliefs*. From these beliefs the agent derives goals, its *Desires*, about what the world should look like and what the agent believes is achievable and eligible. Finally, *Intentions* are seen as partial plans of some courses of actions that the agent has selected to fulfil these goals and has committed to attempt its execution.

During formulation of the theory of rational agents, these formalisms were (partially) put into practise with dedicated languages and interpreters (see Shoham [Sho93]). Shoham defines the AOP paradigm, where computation is done by a society of computational units, the agents, that are controlled by *agent programs* and interact with one another, using concepts

from speech act theory [Sea69]. From engineering point of view these agent programs are written in an agent programming language that allows to specify conditions for making commitments. These commitments are made by the agent based on its beliefs and carried out at the appropriate times by the agent interpreter.

It soon became clear that for complex applications, it was important to support the development of Multi-Agent System (MAS) on a technical level, by frameworks that support the easy development of agents. *dMARS* [dKLW98] is one early example of such a framework.

At the same time, the focus shifted towards the early steps of the software life cycle, analysis and design (for example with the *Gaia* methodology [WJK00]). This can also be seen as the beginning of modern agent-oriented software engineering. Gaia defines a role-based conceptual framework of models that capture structural and interaction characteristics of multi-agent systems, together with a simple process model that shows how these models depend on each other.

In the following years, high level concepts such as roles, intentions, beliefs, behaviours and more, together with accompanying methodologies for analysis and design, were suggested to design complex distributed systems.

The world did of course not stand still outside of the agent community. Object orientation and later service orientation as well as aspect orientation developed as well, incorporating different elements into main stream software engineering (and the other way around). Both, AOSE and classical software engineering (SE) evolved and influenced each other. However, where SE focused on actually implementing systems and has a large portfolio of methods, frameworks and tools for managing and executing large scale software projects, AOSE is still mainly of interest for the research community.

While there *are* languages, frameworks and tools to implement agent systems, there are very few approaches that can be applied in industrial settings, for a variety of reasons. Often there are gaps in the methodologies that are vital for software development, such as requirement elicitation, quality management and tool support. While there are many prototypical implementations of tools and frameworks, few have the quality or maturity for main stream industrial projects.

1.2 Scenarios in this Thesis

The DAI-Labor of TU Berlin has been dedicated to the research of the application of the agent-oriented paradigm to real-world, industry-oriented

projects and products. Since its inception in 1992, the DAI-Labor under the direction of Prof. Dr. Sahin Albayrak has successfully developed intelligent solutions for telecommunications and telematics [AG98, Alb98, Alb99]. The application areas in more than 100 research projects reach from production control [Alb92] to smart assistents for healthcare [ZWA07], recreation [WCR05] and information retrieval [AWV+07]. Recent activities cover the home environment [FBLA06, WBA10] and electromobility [LMH+11, LAH+11, KGM+11].

One principle has ever since been to reduce the time and expenses during application and service development through the provisioning of frameworks [AW98, FBK+01, HKH09] and tools [Cal04, THHA08, KLHH10]. This principle is the driving force behind this thesis.

1.3 Structure of the Thesis

In Chapter 2, the thesis investigates the best practices in software engineering and the state of the art in agent-oriented software engineering. The main contribution of this thesis is then described in the following three chapters: the agent-oriented framework (Chapter 3), the description and steamlining of the methodology (Chapter 4), and the tool evolution (Chapter 5). Framework, methodology and tools have been used and evolved in many projects; representative for them the application in the Multi-Agent Programming Contest and two projects are described in Chapter 6.

Chapter 2

Methodologies and Technologies

For a successful technology,
reality must take precedence over public relations,
for Nature cannot be fooled.
(Richard P. Feynman in Appendix F -
Personal observations on the reliability of the Shuttle)

2.1 Best Practices in Software Engineering

Strangely enough, when talking about AOSE, researchers seldom mention common software engineering techniques such as version management, bug tracking, testing, or integration. Although they are essential for the success of any software project, these techniques seem to be not needed when developing multi-agent systems, or they are so well understood that even researcher could not imagine that people do not think about them.

At DAI-Labor, we have tackled many projects and the following aspects of common software engineering have crystallized to be necessary consistently:

2.1.1 Version Management

> *Version management is essential to software development and is considered the most critical component of any development environment.* [eet07]

Version management is a discipline of Software Configuration Management (SCM) and controls revisions of source code. It usually tracks changes

of the source code together with who has changed it and when. Version control systems file all versions of a project's source code and it is possible to restore any older version if needed.

Version control systems also coordinate several developers concurrently access the source code. Thus it becomes possible to collectively work on the same work piece and to share the results. Furthermore, version control systems allow, through branching, to fix bugs and to develop new features and variations without influencing the whole system.

We distinguish two different types of version control systems: client-server systems and distributed version control systems.

2.1.1.1 Classical Version Control Systems

Classical version control systems are characterized by a central server, which holds the repository of source code, controls access to the repository and manages changes, branching and merging. Here, the source code is backuped and restored if necessary. Developers use command-line clients or Graphical User Interface (GUI) frontends to generate repositories, commit changes, update their local copy and more.

At DAI-Labor we have mainly used two different classical version control systems over a longer period time: CVS [Fog03] and SVN [CSFP04].

2.1.1.2 Distributed Version Control Systems

Distributed version control systems have several advantages [Cla07] for developers over centralized systems, although they can also be employed in classical processes.

- No need for backup. Every local repository is a backup of the codebase and the change history.
- Disconnected operation. This is much faster for the developer and she stays productive as well.
- Simple branching and merging. The version control system itself provides means for experimenting with new ideas and ad-hoc collaboration with other developers.
- Higher integrity. The main development branch becomes more stable and developers are working from a more stable codebase.
- Easy release. Just merge the features into the release branch.

2.1.2 Build System

The build system is a software program or full-featured programming language, which is capable of handling the whole build process for a shapshot version of a project's code or a software release. It must cover all steps from source code checkout, compilation and configuration up to packaging the whole system as it is needed during test, installation or deployment.

Build systems are usually hard to use, as they not only require deeper knowledge of development processes, dependency and configuration management, but often also presume programming skills.

One of the first, and still widely used build system, is *make*, originally a dependency tracking build utility for the UNIX operating system from 1977 [Mec04]. Make's language is similar to declarative languages, which makes it difficult for most programmers used to imperative programming. *Ant* [Hol05] is a Java-based build tool, released in 2000, which uses XML and Java. It is a platform-independent tool for automating the software build process.

Apache Maven [Apa10] is a powerful, easy to use and easily extensible build automating sytems. It is also Java- and XML-based and is characterized by a sophisticated development model derived from best practices and a strict dependency management with download support. Maven can be integrated into Integrated Development Environments (IDEs) und thus seamlessly supports the productivity of the developer.

2.1.3 Testing

Software testing comprises a number of software development techniques to assure the validity of the code and to verify that the product meets the customer's needs. Testing software most often results in a higher quality of programmed and integrated code and also responses to changes in the code base of components and the system as a whole. Modern agile methodologies use the Test-Driven Development (TDD) approach as inspired by Beck [Bec02] and Astels [Ast03]. Combined with automated unit and integration tests this builds a powerful and highly efficient tool for development of quality applications and services.

2.1.4 Continuous Integration

Martin Fowler in his paper "Continuous Integration" [Fow00] stated that many projects do not have a reliable build process including testing, which is very surprising. Moreover, the build process can be done fully automat-

ically and can run many times a day. Continuous Integration (CI) detects integration errors as early as they are introduced by developers, submitting their changes or enhancements into the software or service.

In a more recent version of the above article [Fow06], Fowler reports on the advantages of CI:

> Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

In the article, Fowler sums up the key features of CI, which are further extended in [DMG07]. CI is a number of practices that are applied in daily life of the developer and integrates most of the above chapters into an automated process:

- **Single Source Repository** – this practice is also covered in the paragraph about version management (see Section 2.1.1); taking into account distributed version management systems, we adopt this practice by maintaining a CI branch, where changes countinuously are committed into
- **Automatic Build** – this practice presumes a build system that covers all aspects of software building: source code compilation, testing, packaging and deployment. It also recommends a dedicated CI server where builds can be run frequently and obtain reproducable results
- **Self-testing code** – a suite of automated tests that can check large parts of the code
- **Any environment** – having an automated build, it becomes possible to test the outcome on any environment, inclusive a clone of the production server and different operating systems
- **Automated Deployment** – the final build is deployed to multiple environments, speeding up the process and reducing errors
- **Easy Access to Executable** – providing the latest build for demonstrations, user testing and reuse
- **Communication** – the states of the build will be communicated to all involved parties such as programmers and project managers who then can immediately react on failures

CI servers are usually extensible frameworks to establish and control the CI process. *CruiseControl* [Tho01] was one of the first open-source CI servers that we used at DAI-Labor for building our agent frameworks, applications and services, which has also been demonstrated at the AAMAS conference in 2004 [HKF+04]. As CI servers also evolve over time we then switched to *Apache Continuum* [Fou04a], which had a better integration of the Maven build system, benefitting from the sophisticated Maven build process. Both systems helped us to speed up the development process and to discover problems early.

2.1.5 Issue Tracking

Issue tracking is one of the most underestimated disciplines in software engineering. Issues normally can be classified into two types: bugs and feature requests. When done right, these issue tracking can trigger a next iteration (in case of bugs) or increment (in case of feature requests) in the software development lifecycle. Issue tracking also gives good hints towards quality and usability of the software system.

Issue tracking systems need to manage both highly detailed issue reports and easy issue creation. This is a bit antithetic because, on the one hand, developers want to know as much about the issue as possible to better fix the bug or meet the need of the feature requester. On the other hand, creating a ticket must be as easy as possible otherwise people will not do it. So the best issue tracking system would be one which hides the internal bureaucracy from the tester's or user's side.

Issue tracking systems are often bundeled with other developer tools, which makes the overall performance of the developer team more efficient. These are for example Wikis or project management systems (see for example Trac [tra08]). Or they sync with version management systems (see Section 2.1.1) to follow changes on the codebase attached to the according issue (such as Bugzilla [bug10] or MantisBT [Man08]).

2.1.6 Use of Patterns

Design patterns in software development are methods of reuse, which document the solution and expertise to a certain problem in order to have it ready next time when the problem occurs. Design Patterns have been first documented in architecture by Alexander [AIS78] and have been adopted to software design by the so-called "Gang of Four" [GHJV93], and further implemented using different programming languages [Met06, Bis07].

The use of patterns has then be assigned to other disciplines of software development, such as analysis [Fow96], software architecture [Fow02], organizational structure and culture [Cop04] or interaction in distributed systems. Finally, anti-patterns show common errors or inefficient solutions.

2.1.7 Model Driven Engineering

Model Driven Engineering (MDE) and OMG's Model Driven Architecture (MDA) [MM03] are intended to improve and accelerate software development. The motivation is that software development should focus on a program's specification instead of the implementation. By extensive use of generators the code is correct, quickly produced, reusable, conforming to standards and of high quality.

In MDE a number of models are used for different views on the system, each with a different level of abstraction.

1. The *Code* can be seen as the lowest level of abstraction.
2. A *Platform Specific Model (PSM)* is abstracting from the program code while preserving its basic concepts, like classes and methods. A popular example for this kind of model is Unified Modeling Language (UML) Class Diagrams.
3. The *Platform Independent Model (PIM)* is giving a very abstract view on the system. Examples for platform independent models are for instance some UML diagrams, like UML Usecases and Activity Diagrams and the Business Process Modeling Notation (BPMN).

In MDE the PIM is used for modeling the program's functionality from a high level view ("programming in the large"). This model is especially useful to involve stakeholders and domain experts in the development. The PIM then can be used to generate PSM for further refinement and completion, while non-functional, implementation specific details, such as exception handling and resource management, do not have to be taken into account. Finally the program code is generated from the PSM. In the ideal case no manipulation of the code is necessary; it can be fully generated from the higher level models. Some tools also provide *round tripping*, meaning that changes made to the code are automatically transferred to the models, keeping them up to date.

When the project has to be modified, in MDE it is best practice to change the models and to regenerate the code. This way MDE encourages *incremental and iterative design* and the model is kept up to date, as shown

in Figure 2.1. Further the model is *independent of platform and programming language*. The same model can be used to generate an equivalent program in a different language, making programs developed using MDE highly portable: When a company has to change to a different programming language or framework the new software can be generated from the same models.

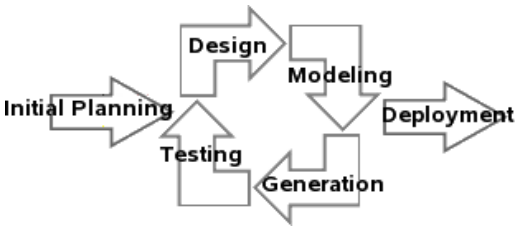

Figure 2.1: Modeling and regenerating in a model driven software development lifecycle

MDE might look like an unnecessary additional effort. Creating the different models is time-consuming and understanding the generated code and the interaction of the several generated modules can be much more difficult than understanding code written by oneself. But in projects with many developers MDE can help to provide a consistent coding style and program structure that can be understood by everyone involved in the project.

The method of *Model Driven Engineering* is best suited to supplement an agent framework at least in two aspects: from the perpectives of the application domain and the business domain [Sch06].

2.2 Accomplishment in Agent-Oriented Software Engineering

In the "Call For Papers" of the "AOSE Workshop 2011"[1] the organisers state:

> *Since the early 1990s, multi-agent system researchers have developed a large body of knowledge on the foundations and engineering principles for designing and developing agent-based systems.*

But in the same line of motivation for the workshop they also state:

[1] `http://distrinet.cs.kuleuven.be/events/aose/2011/print.php. Retrieved2011-03-14`

. . . we aim to find a way out of the increasing fragmentation and fuzziness on software engineering in AOSE.

Let us have a look at the main achievements in AOSE so far, although there are numerous others. The field is divided into three sections: agent-oriented methodologies, frameworks and tools.

2.2.1 Frameworks

Frameworks promise higher productivity and shorter time-to-market through design and code reuse. [Rie00]

With this said, agent-oriented frameworks should promise higher productivity and shorter time-to-market of agent-based applications and services through design and code reuse. In the following, the most prominent representatives of agent-oriented frameworks are described with a special focus on what can be reused when developing an agent-based application or service.

2.2.1.1 JADE

Java Agent DEvelopment framework (JADE) [BPR99, BCPR03, FB07] is a framework for realizing agent-based applications developed by the Telecom Italia Lab. JADE is the most popular agent framework in the world: about 20 universities and a number of commercial organizations have used JADE in teaching or in projects, respectively. During the *AgentLink-AgentCities* project [LW03], which has seen the largest deployment of agent systems to date, from more than 100 platforms hosted in more than 20 countries, about 80 percent were JADE platforms or extended (as for example BlueJADE [2]).

A JADE application is in principle a distributed application characterized by a hierarchical organization of runtime containers. Containers run on different Java editions (J2EE, J2SE, J2ME) providing a homogeneous middleware for developers to build their applications upon. JADE agents have their own thread of execution, can find each other and interact by sending asynchronous, direct messages to one another (peer-to-peer communication) using Foundation for Intelligent Physical Agents (FIPA) standards.

The agents' internal control is realized by so-called *Behaviors*. A Behavior is an *EventHandler* that reacts on state changes or messages and

[2] http://sourceforge.net/projects/bluejade/

implements an *Action* that will be executed when the EventHandler is activated. There are several behaviors that allow to implement any kind of control flow.

Additionally, JADE provides the concept of ontology to describe the domain specific vocabularies that agents can use in messages. The use of ontologies is a bit tricky as one cannot directly instantiate class objects. Instead, one has to use so-called *abstract descriptors* that reify elements of an ontology. To call instances or concrete objects *abstract descriptors* is a determining factor that makes the use of JADE more complicated than it should be.

JADE provides a FIPA-compliant infrastructure with Agent Management System (AMS), Directory Facilitator (DF) and the Message Transport System (MTS). JADE agents may also expose their agent services as webservices using the Web Service Integration Gateway (WSIG) [Boa08]; the registration and use of webservices as agent services is not supported by the framework.

2.2.1.2 JACK

JACK Intelligent Agents (JACK) [BRHL99, Win05] is a development platform for autonomous systems and has been developed by AOS Australia. It is thought to develop applications that "interact with a complex and ever-changing environment". In JACK, agents are described in terms of beliefs, desires and intentions (BDI) and have their own thread of control. JACK has been written in Java and scales from consumer device to multi-CPU computers.

To program a JACK agent, the developer will use the JACK Agent Language (JAL). JAL extends the Java syntax with agent-oriented concepts of *Agent, Event, Plan, Beliefset, View* and *Capability*. A JACK program must be compiled with the JACK pre-compiler before it can be executed by the agent kernel.

There exist a number of extensions: *JACK Teams* extends JACK to "facilitate the modelling of social structures and coordinated behavior". In this team concept, a team is a seperate entity apart from its members and has own beliefs and reasoning capabilities. *CoJACK* extends the BDI architecture to better simulate human actors. In this model, one can easily evaluate how physiological and emotional factors interfere with cognition.

The JACK agent framework is bundled with the JACK Development Environment (JDE), which supports the design of agent applications, code generation and the tracing of agent execution (see also Section 2.2.3.3).

2.2.1.3 Cougaar

Cougaar [HTW04, HW05] is an agent framework for developing scalable distributed agent-based applications, build during the UltraLog program of the US Defense Advanced Research Projects Agency (DARPA).

Each Cougaar agent has its own *blackboard*, which allows inter-component communication via a subscribe-notify-mechanism, and can be extended by components, which are called *Plugins* that implement application specific behavior. *Binders* are components that enable agents to access external, non-agent resources and *Agent Framework Services* denotes the sum of all infrastructure functionalities of a cougaar *Node* (the Jave Virtual Machine (JVM)): White and yellow page services, and the message transport service (MTS). The MTS is the basis for inter-agent communication by message passing and can be adapted to the underlying network regarding reduced bandwith, unreliable connections or high security requirements.

The *Logical Domain Model (LDM)* builds the domain specific vocabulary to describe application data and defines the logic to use it in messages or to translate them between domains. Cougaar already supplies domains for workflow-base planning and logistics, the two main application areas of Cougaar.

2.2.2 Methodologies

In the following the most influential methodologies are described and analysed regarding their integrative potential.

2.2.2.1 Gaia

Gaia is a high-level, abstract agent-oriented analysis and design methodology that has been first published by Wooldridge, Jennings and Kinny in 2000 [WJK00]. Since then it has seen two revisions in 2003 [ZJW03] and 2005 [ZJW05]. The main focus of this methodology lies on the definition of *roles* and the implications from seeing autonomous, computational entities (the agents) as part of organisations, as in human, real-world organisations. This leads to a MAS architecture that reflects *organisational structures*, follows *organisational rules* and is embedded in an *environment* of (non-agent) resources that must be perceived and acted on. The methodology also adheres to a rigid, waterfall-like process model for the elaboration of the high-level entities and relationships (see Figure 2.2). As the Figure shows, Gaia defines eleven interconnected models that developers need to build. Some of those models are refinements of earlier versions (role model, interaction

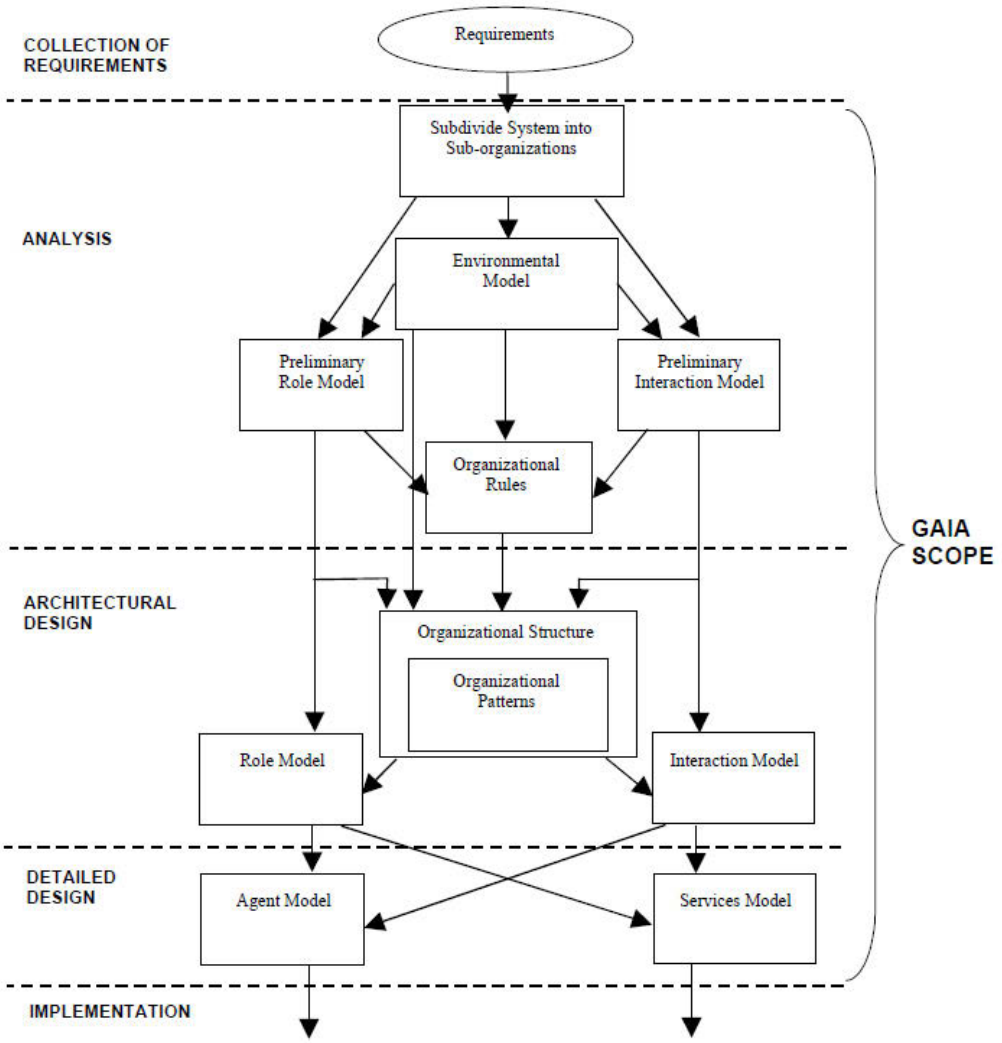

Figure 2.2: The Gaia process model (source [ZJW03])

model). In the end, the developer has two models to be implemented: agent model and service model.

Gaia is a top-down, global view approach without noticing that the development of a MAS is still software engineering, where iterative and incremetal methods are crucial success factors [LB03]. Gaia also negates or ignores more promising, earlier approaches to organisation abstractions and role modeling as for example by Kendall [Ken98] or by Drogoul and Collinot [DC98]. Especially, the segragation of interaction model from the role model and reducing it to the notation of *protocol* has caused much confusion among researchers [PBFO04].

But the major drawback of Gaia is its practical applicability. While Gaia produces very detailed models, they are never checked against requirements (or customers feedback) nor is the feasability tested, for example by means of prototypes. Therefore the design does most probably not meet the customer's needs nor can it be implemented as designed (in terms of functionality and completeness).

2.2.2.2 Tropos

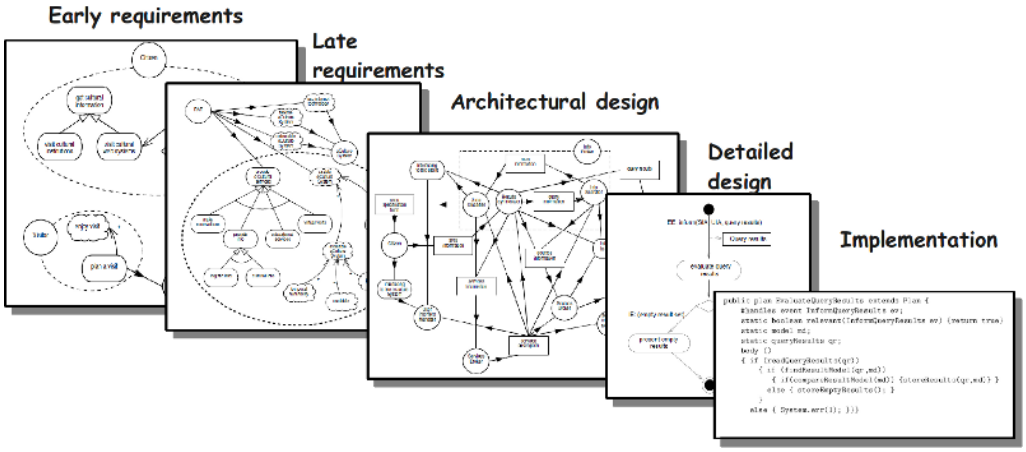

Figure 2.3: The Tropos process model (source www.troposproject.org)

Tropos is a agent-oriented development methodology [BGG+04] that comprises five phases: *Early Requirements, Late Requirements, Architectural Design, Detailed Design* and *Implementation* (see Figure 2.3). Its main application area is the development of multi-agent systems, but has also been used for realising self-adaptive systems and robot control [MPP08]. It is based on *i** [Yu95], which provides an excellent theoretical background for early requirements analyses. i* has been extended with the concepts of agent and mental states based on BDI agent architectures. Tropos has also been extended to model security concerns: Secure Tropos [MG07].

The essential feature of Tropos is the continuity during the whole development process regarding vocabulary and modelling notation, even in different abstraction levels. Implementation can be seemlessly done with any agent framework featuring BDI concepts.

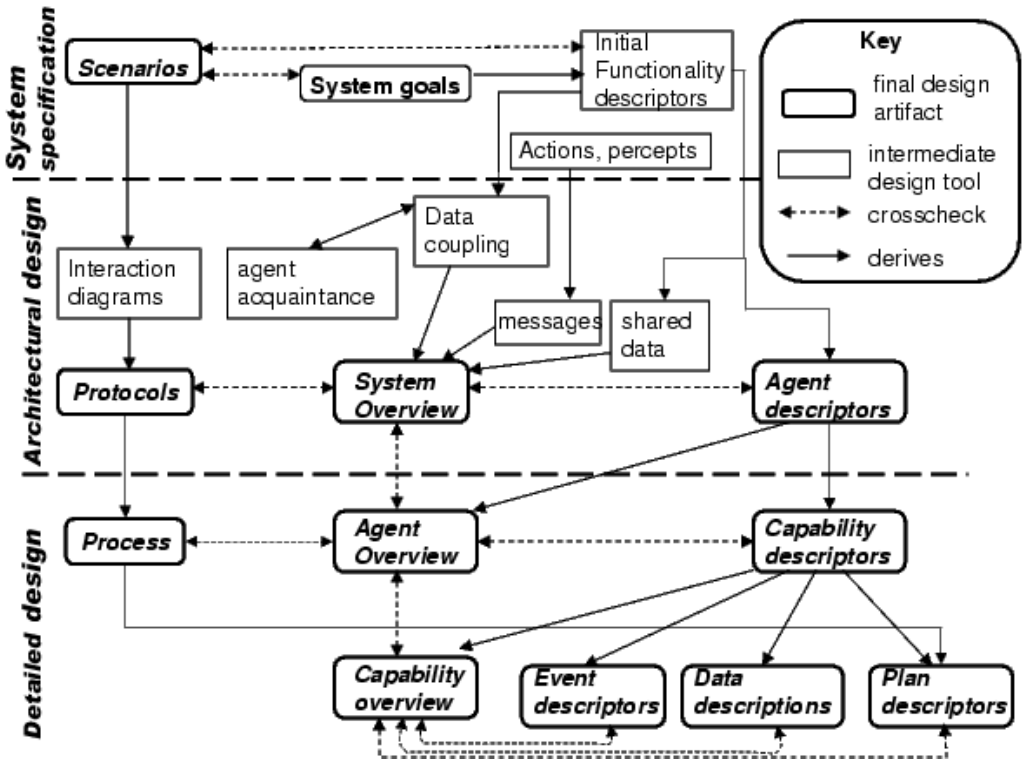

Figure 2.4: The Prometheus process model (source [PW04])

2.2.2.3 Prometheus

The Prometheus methodology [PW02, PW04] is intended to teach the development of intelligent agent systems to "industry practitioners and undergraduate students who do not have a background in agents". Prometheus has a strong link to the JACK agent tookit (see 2.2.1.2) as the preferred target platform for implementation. Both, Prometheus and JACK, share the same agent meta-model.

Prometheus is divided into three phases (see Figure 2.4): *System specification*, *Architectural design* and *Detailed design*. The System specification phase starts identifying the systems goals, their clarification and analysis. Scenarios contain use cases that are developed. It is also necessary to analyse the external, non-agent environment in order to find out what agents perceive and what they act on. Finally, functionalities are identified and written to functionality schemas. in the Architectural design phase functionalities are grouped to form agent types. A system overview diagram connects agents with their shared data and events. Interaction diagrams and protocols complete the architectural design phase. The detailed design

phase concentrates on the single agent behavior. The internal behavior is defined in terms of capabilities, which constitute plans, events and beliefs of the agent.

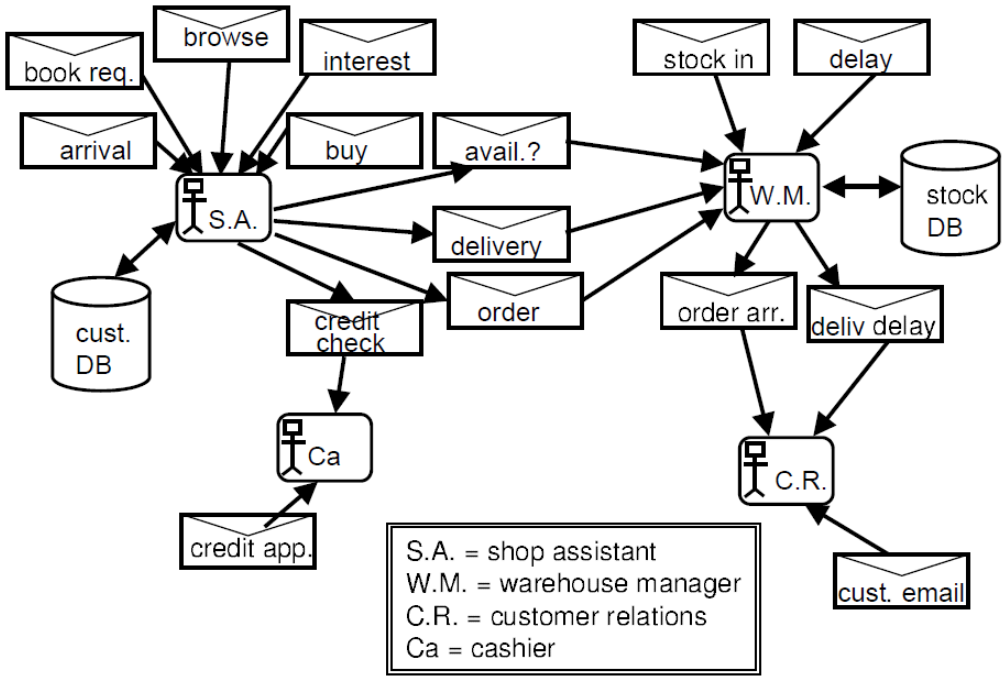

Figure 2.5: Example Prometheus system overview diagram (source [PW02])

Throughout the Prometheus phases the developers are encouraged to use a pictogram-like notation that is quite intuitive to use as shown in Figure 2.5. The methodology is supported by two tools, the JACK Development Environment (JDE) (see 2.2.3.3) and the Prometheus Design Tool (see 2.2.3.4).

2.2.3 Tools

We have evaluated a number of agent development tools, some of them already seeing the 3rd generation [age, THHA08].

2.2.3.1 The agentTool System

The *agentTool* system [WJ05, pp. 245–259] is a visual design environment for top-down design of multi-agent systems, supporting the steps of the *MaSE* methodology [Woo00]. In its 3rd edition it supports *O-MaSE* [DeL06] with its nine different model types and is supplemented

by a consistency checker, code generation, a metrics calculator and process engineering support. All aspects of an agent-oriented design can be modelled nicely, whereas the code generation feature is more or less an open issue.

2.2.3.2 The IDK

The INGENIAS Development Kit (IDK) is a visual development tool supporting the *INGENIAS* methodology [GSFF03, GMGFF09] and targeting the *INGENIAS Agent Framework*. Any model element results in a skeleton that can be extended by the programmer. The IDK provides a extensibility features and has been developed as open source project on the basis of an extensible plug-in architecture [GMGSA08]. The main focus, however, lies on code generation plug-ins. The most popular implementation in this category is the INGENIAS Agent Framework (IAF) [GSP06]. The plug-in is included within the standard IDK setup and provides a translation of the design into executable Java Agent DEvelopment Framework (JADE) code.

2.2.3.3 JDE and the JDE Design Tool

The JACK Development Environment (JDE) [WJ05, p. 261–277] supports the development of *JACK* based applications. It provides for the creation and manipulation of each JACK component by means of visual system engineering and includes several other specialised tools like the JDE *Graphical Plan Editor* or the *JDE Design Tool*. Project management is accomplished by the *JDE Browser*. MAS configuration is accomplished by the *JDE Design Tool* [Age05] and allows visual system engineering on the basis of drag and drop. Code generation and execution is provided by the JDE as well. The *Compiler Utility Tool* translates the developed diagrams into Java classes and supports both, execution and debugging.

2.2.3.4 PDT and the CAFnE Toolkit

The Component Agent Framework for domain-Experts (CAFnE) Toolkit [JTPW06] has been developed as a successor to the Prometheus Design Tool (PDT). The tool provides domain experts a suitable way to easily build and modify multi-agent systems and accomplish modifications in complex agent-based domains, while its intuitive operability particularly addresses persons with limited programming skills. The toolkit allows visual modelling, code generation, compilation and execution of agent based applications. MAS development is achieved by the specification of diagrams. CAFnE operates on a framework unspecific and simplified BDI domain model, which

provides platform independent as well as BDI compliant MAS configurations. An effective error avoidance mechanism is provided by CAFnE as well. Next to design-features, CAFnE provides the generation of framework dependent executable code. A complete transformation module for JACK is available. By consulting both, a transformation configuration and a set of transformation rules, an internal transformation module converts the platform independent domain model to an executable agent configuration.

Chapter 3

Agent Framework

To pursue science is not to disparage the things of the spirit. In fact, to pursue science rightly is to furnish the framework on which the spirit may rise.
(Vannevar Bush)

The design of JIAC V was guided by the simple paradigm to take the successful features of JIAC IV and rebuild them with modern software-libraries and technologies. However, while the technologies and technical details may have changed, most features of JIAC IV are still present. Nevertheless, we made some deliberate design changes to the agent architecture. This was mainly aimed at simplifying things for beginners and for the programmer, as we felt that ease of learning and usability were the two aspects that needed the most improvements.

The main objectives of JIAC's architecture are:

- Transparent distribution
- Service based interaction
- Semantic Service Descriptions (based on ontologies)
- Generic Security, Management and Authentication, Authorisation, and Accounting (AAA)[1] mechanisms
- Support for flexible and dynamic reconfiguration in distributed environments (component exchange, strong migration, fault tolerance)

[1] AAA refers to security protocols and mechanisms used for online transactions and telecommunication. For more information see [NN05].

JIAC V agents are programmed using JIAC Action Description Language (JADL++) [KHA06]. This new language features knowledge or facts based on the ontology language Web Ontology Language (OWL) [MvH04] as well as an imperative scripting part that is used for the implementation of plans and protocols. Moreover, it allows to semantically describe services in terms of preconditions and effects, which is used by the architecture to implement features such as semantic service matching and planning from first principles. The architecture implements dynamic service discovery and selection, and thus the programmer does not have to distinguish between remote services and local actions.

The JIAC V agent model is embedded in a flexible component framework that supports component exchange during runtime. Every agent is constructed of a number of components that either perform basic functionalities (such as communication, memory or the execution cycle of an agent) or implement abilities and access to the environment of an agent. These components are bundled into an agent by plugging them into the superstructure of the agent.

During runtime, all parts of an agent, i.e. all components as well as the agent itself, can be monitored and controlled via a generic management framework. This allows either the agent itself or an outside entity such as an administrator to evaluate the performance of an agent. Furthermore, it allows the modification of an agent up to the point where whole components can be exchanged during runtime.

The execution cycle of an agent supports the BDI [Bra87] metaphor and thus realises a goal oriented behaviour for the agents. This behaviour can be extended with agent abilities like planning, monitoring of rules, or components for e.g. handling of security certificates.

Finally, communication between JIAC V agents is based around the service metaphor. This metaphor was already the central point in the design of the JIAC IV architecture. However, JIAC V extends the rather restricted service concept of JIAC IV to include multiple types of services, thereby allowing to integrate all kinds of technologies, ranging from simple Java-methods to semantic service described in OWL-S [BHS+04].

3.1 JIAC Meta-model

The JIAC agent framework supports the development of MAS using BDI agents and standard Java technologies. The framework has been implemented using the Java programming language. Two building blocks

constitute the basic agent architecture: a Java Spring[2] based component system and the language (JADL++). The basic architecture of a JIAC-based application is summarised in the MAS meta–model, which is shown in Figure 3.1.

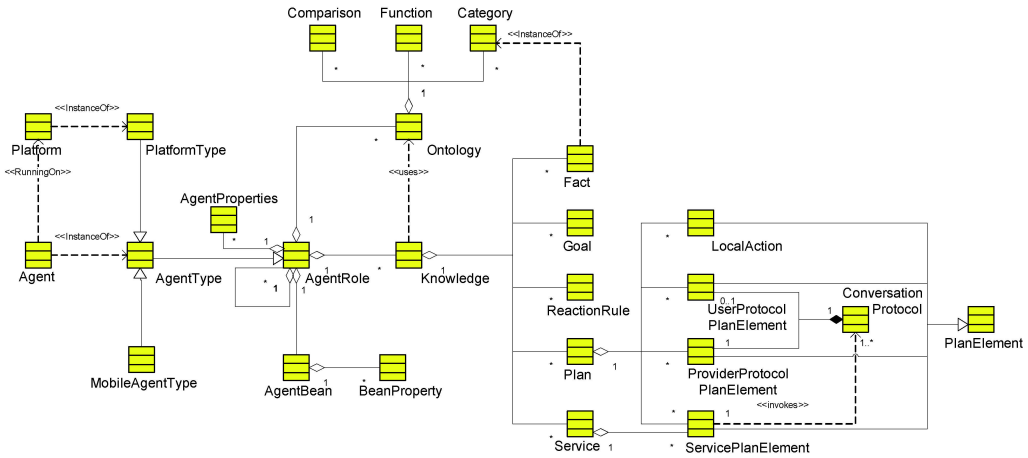

Figure 3.1: JIAC MAS meta-model

In the framework, the following concepts are defined and must be supported by tools:

- Domain Vocabulary
  - *Ontologies* define *classes*, which are used to create the beliefs and the interaction vocabulary of the agents.
  - In addition to classes, ontologies provide *properties* which can describe relationships between class instances.
- Knowledge
  - Initial beliefs (*facts*) using these categories are created before the agent is started.
  - *Reaction rules* constitute the reactive behaviour of an agent.
  - *Actions* define the behaviour of the agents. They can be deliberatively selected and then become intentions. Actions can be used to aggregate more complex plans by either the developer or a planning component as part of an agent.

[2] `http://www.springframework.org/`

- Component
  - *Agent beans* are core components and also used to wrap or connect the non-agent environment via Java Application Programming Interfaces (APIs) or user interfaces. They communicate with each other using the agent's memory.
- Deployment
  - *Agent roles* are composites of agent functionalities and interaction capabilities (services) from the above concepts.
  - *Agents* are agent roles that have standard components as well as domain specific agent roles and are able to run on an agent platforms.
  - *Agent nodes* are the infrastructure for each computer, which play the role of an AMS and a DF [Fou04b], i.e. they provide agent management and white and yellow page services, and constitute the agent environment and infrastructure services. All nodes that know each other are called *Platform*.

3.2 JIAC V

3.2.1 Platform

JIAC V is aimed at the easy and efficient development of large scale and high performance MAS. It provides a scalable single-agent model and is built on state-of-the-art standard technologies. The main focus rests on usability, meaning that a developer can use it easily and that the development process is supported by tools.

The framework also incorporates concepts of service oriented architectures such as an explicit notion of service as well as the integration of service interpreters in agents. Interpreters can provide different degrees of intelligence and autonomy, allowing technologies like semantic service matching or service composition. JIAC V supports the user with a built-in administration and management interface, which allows deployment and configuration of agents at runtime.

The JIAC V methodology is based on the JIAC V meta-model and derived from the JIAC IV methodology. JIAC V has explicit notions of rules, actions and services. Composed services are modelled in BPMN and transformed to JADL++. We distinguish between composed services and

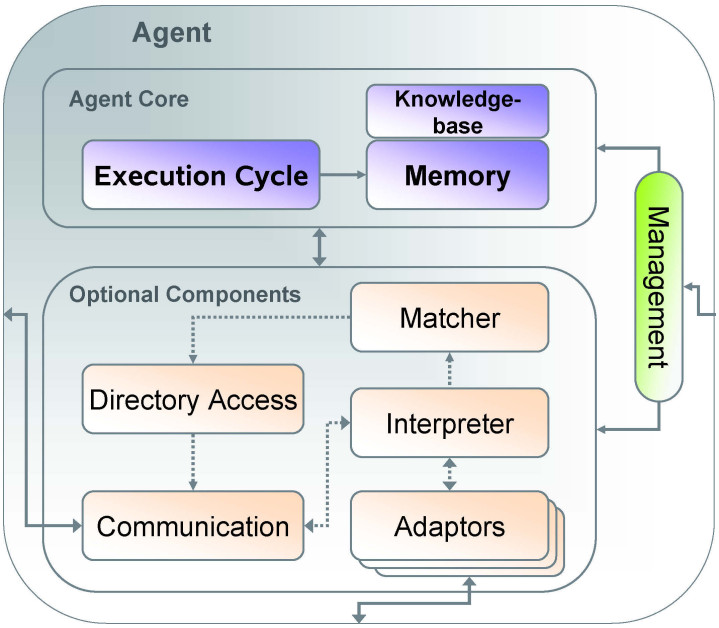

Figure 3.2: The architecture of a single agent

infrastructure services. The former can be considered a service orchestration with some enhancements (e.g. service matching) whereas the latter describes special services that agents, composed services, or basic actions can use, e.g. user management, communication or directory services. Rules may trigger actions, services or updates of fact base entries. Basic actions are exposed by AgentBeans and constitute agent roles, which are plugged into standard JIAC V agents. The standard JIAC V agent is already capable of finding other JIAC V agents and their services, using infrastructure services, and it provides a number of security and management features.

The core of a JIAC V agent consists of an interpreter that is responsible for executing services (see Figure 3.2).

Our approach is based on a common architecture for single agents in which the agent uses an adaptor concept[3] to interact with the outside world. There exists a local memory for each agent to achieve statefulness, and each agent has dedicated components (or component groups) that are responsible for decision making and execution.

In JIAC, the adaptor concept is used not only for data transmission, but also for accessing different service technologies that are available today. Thus, any call to a service that is not provided by the agent itself can be

[3]Each of these adaptors may be either a sensor, an effector, or both.

pictured as a call to an appropriate effector. Furthermore, the agents' interpreter allows to execute a set of different services. These services' bodies may also contain calls to different services or subprograms. Consequently, an agent is an execution engine for service compositions.

In the following, a brief explanation of the function of each component is given:

- **Matcher:** The Matcher is responsible to match the invoke commands against the list of known services, and thus find a list of applicable services for a given invoke. The service templates within the invoke commands may differ in completeness, i.e. a template may contain a specific name of a service together with the appropriate provider, while others may just contain a condition or the set of parameters.

  Once the matcher has identified the list of applicable services, it is up to the interpreter to select a service that is executed. Note that this selection process includes trial&error strategies in the case of failing services.

- **Memory:** The interpreter uses the agent's memory to manage the calls to services as well as the parameters. We use a simple *Linda*-like tuple space [Gel85] for coordination between the components of an agent. Additionally, the current state of the execution can be watched in the memory any time by simply reading the complete contents of the tuple space, allowing for simple solutions for monitoring and debugging.

- **KnowledgeBase:** The knowledge base provides functionalities for reasoning and inferences within an agent. All declarative expressions within either a service description or an action invocation are evaluated against this knowledge base. In contrast to the Memory above, the knowledge base is a semantic memory rather than a simple object store and has a consistent world model.

- **Interpreter:** The interpreter is the core for service execution. It is able to interpret and execute services that are written in JADL++. Essentially, all atomic actions that can be used within the language are connected to services from either the interpreter or the effectors of the agent.

- **Adaptor:** The adaptors are the agent's connection to the outside world. This is a sensor/effector concept in which all actions that an agent can execute on an environment (via the appropriate effector) are

explicitly represented by an action declaration and optionally a service declaration that is accessible for the matcher. Thus, all actions may have service descriptions that are equivalent to those used for actual services.

3.2.2 Language

JIAC V features an agent programming language called *JADL++* While it is possible in JIAC to implement agents using plain Java, JADL++ allows the programmer to implement agents with a high abstraction level and the explicit usage of knowledge based concepts.

JADL++ consists of a scripting and a service description language part. It is designed to support programmers in developing different types of agents, ranging from simple reactive agents to powerful cognitive agents that are able to use reasoning and deliberation. The language is the successor of the *JIAC Action Description Language (JADL)* [KHA06].

In order to understand some of the design choices made, it is useful to compare the main features with JIAC V's predecessor JIAC IV and its agent programming language JADL.

Our experiences with JADL were mixed [HFKP08]. On the one hand, using a knowledge oriented scripting language for programming agents worked quite well. With JIAC IV, agents and applications could be programmed very efficiently and on a high abstraction level. Also, the addition of STanford Research Institute Problem Solver (STRIPS) style preconditions and effects [FN71] to action and service descriptions allowed us to enhance agent programs with error correction and planning from first principles, which in turn resulted in more robust and adaptive agents.

However, there were a number of drawbacks that prompted us to change some of the features of JADL++ quite radically. The first of these was the use of a proprietary ontology description language. The use of ontologies to describe data types is a sound principle [CJB99], but the proprietary nature of JIAC IV ontologies defeated the purpose of interoperability. Although we did provide a mapping to OWL ontologies [MvH04], thereby providing some measure of interoperability, it was a clear disadvantage to work with proprietary ontologies, as the whole idea of ontologies is to share knowledge, and to include publicly available knowledge bases. Therefore, we now use OWL for ontology descriptions and OWL-S [BHS+04] for service descriptions in JADL++.

Furthermore, JADL allowed exactly one method of agent interaction, namely though service calls. Although a programmer was free to use any FIPA speech acts [Fou02a] within that service protocol, the core service

protocol itself was always wrapped by a FIPA request protocol [Fou02b]. The restriction of communication to services allowed us to implement strong security features which were instrumental in JIAC IV becoming the first security certified agent framework according to common criteria level three [GKP04, com99a, com99b, com99c].

The service approach proved to be quite comfortable for furnishing services with security or quality of service, but it did have some distinct drawbacks with respect to more simple types of communication. For example, it was not possible to send a simple Inform speech act to one or more agents in JADL, unless those agents provided a service that received that speech act. Therefore, JADL++ now features a message based interaction method that allows both, single Inform messages and multicast messages to groups of agents.

Last but not least, we changed the syntax style. JADL++ uses a C-style syntax for most procedural aspects of the language, while JADL used a LISP-like syntax. The reason for this is that the acceptance of JADL's LISP-style syntax among programmers was not very good, and programmers tended to confuse the logical and procedural parts of the language which both had a similar syntax. JADL++ now clearly discerns between the two programming concepts.

3.2.2.1 Specifications and Syntactical Aspects

In order to explain the role of JADL++ within an agent, we need to give a brief description of how a JIAC V agent is constructed and how it operates. A more detailed explanation of an agents structure is given in Section 3.2.1.

The basic concept of a JIAC V agent is that of an intelligent and autonomously acting entity. Each agent has its own knowledge, its own thread of execution and a set of abilities called actions that it can apply to its knowledge or its environment. Furthermore, agents are able to use abilities provided by other agents, which we then call services.

In order to implement a new agent, a programmer needs to identify the roles the agent is supposed to play. For each of these, relevant goals and actions that are necessary to fulfill the role are specified. Moreover, each action is either marked as private or accessible from other agents. Finally, the programmer needs to implement the actions.

The implementation of the actions can be done in various ways. An action is made available to the agent by the inclusion of a component that executes the actions functionality. Consequently, the most basic way to implement an action is to implement it in pure Java, and to include a

component that calls the Java code in the agent. Other options include the usage of web services or other existing technologies.

However, while implementing an action with Java and plugging it into an agent is very straightforward, it has some distinct drawbacks. First of all, the composition of multiple actions into a single workflow is complex and error prone. Action invocation in JIAC V is asynchronous and thus the programmer has to take care of the synchronization if he wants to invoke multiple actions. Furthermore, while the agent's memory can be used with simple Java objects, one strong feature of JIAC V is the possibility of using OWL ontologies to represent knowledge. Access to these ontologies is possible from Java, but not very comfortable as most APIs work in a very generic way and require a programmer to know the used ontology by heart.

To alleviate the above issues, we introduce the agent programming language JADL++. This language does not aim at introducing elaborate language concepts but is merely devised to simplify the implementation of large and complex applications with JIAC V. At the same time, it tries to embed OWL and especially OWL-S and support programmers that do not have a logics background in their usage of both. An action that is implemented in JADL++ can be plugged into an agent via a special component that implements an interpreter for the language and can hold multiple scripts. To the agent, it looks like any other component that provides actions implemented in Java.

JADL++ is fairly easy to learn, as it uses mostly elements from traditional imperative programming languages. The instruction set includes typical elements like assignments, loops, and conditional execution. For the knowledge representation part of the language, JADL++ includes primitive and complex data types, where the notion of complex data types is tightly coupled to OWL-based ontologies. The complex data types are the grounding for the integrated OWL support, and a programmer is free to either use these complex data types like conventional objects or he can use them within their semantic framework, thus creating more powerful services.

A simple example of the language can be seen in Figure 3.3. This example shows the implementation of a service that searches the memory of an agent for a car and then calls a registration service. The service has two input parameters: the name of the owner and the color of the car. Additionally, it has an output parameter that returns the found car. If the name of the owner is left empty, the service will search any car with a matching color.

For the memory search, the script first creates a template (either with the name of the owner or without it). In line 13, `new CarOntology:Car` is used to create a new object instance from the class `Car` as described in the

```
include de.dailab.ontology.CarOntology;
import de.dailab.service.RegistrationOffice;

service FindAndRegisterCarService
  (in $name:string $color:string)
  (out $foundCar:CarOntology:Car)
  {

    if ($name != null) {
      // create a car template with name and color
      // and search the memory for it

      $carTemplate = new CarOntology:Car();
      $carTemplate.owner = $name;
      $carTemplate.color = $color;
      $foundCar = read $carTemplate;
    }
    else {
      // if no name is given, any car will do

      $carTemplate = new CarOntology:Car();
      $carTemplate.color = $color;
      $foundCar = read $carTemplate;
    }

    // now call the registration service to register the found car
    var $result:bool;
    invoke CarRegistrationService ($foundCar) ($result);

    // and print the results
    if($result) {
      log info "Success";
    } else {
      log error "Failure";
    }

    // end of service
    // the output variable $foundCar is already filled,
    // so no return is needed.
  }
```

Figure 3.3: JADL++ example

ontology `CarOntology`. Properties of a car are accessed via the `.` operator. For example, in line 15, `$carTemplate.color` denotes the color of the car object that is stored in the variable `$carTemplate`.

Access to the memory is done using a tuple space semantic. That means that the memory is given a template of an object that should be retrieved and then tries to find the object stored in the memory that gives the best match with the template. The match may not have property values that contradict the values of the template. However, if any property values are not set (either for the template or for the matching object) the match is still valid.

During the course of the script, another service is invoked. This service is used to register the car with the registration office. The service is imported via the `import` statement at the top of the example, so no fully qualified name is necessary. The invoked service has one input and one output parameter and as we have given no further information regarding providers etc., the agent will call the first service it can find that has a matching name and matching parameters.

Figure 3.4 shows a shortened version of the syntax of JADL++ in eBNF notation. We omit production rules for names and identifiers, as well as definitions of constants. The complete syntax with accompanying semantics can be found in [HKBA10].

Data Types & Ontologies The primitive data types are an integral part of the language. Internally, data types are mapped to corresponding XML Schema Definition (XSD) datatypes[4] [W3C04], as this makes the integration of OWL simpler. Instead of implementing the full range of XSD types only the most important ones, namely `bool`, `int`, `float`, `string`, and `uri`, are currently supported.

An important aspect of the language is the integration of *OWL-Ontologies* as a basis for service descriptions and knowledge representation in general. As mentioned already, services are defined using the OWL-S service ontology. OWL provides the semantic grounding for data types, structures, and relations, thus creating a semantical framework for classes and objects that the programmer can use. In contrast to classical objects, however, the programmer does not need to directly and fully instantiate all properties of an object before they are usable, as default-values, inheritance, and reasoning are employed to create properties of an instance that have not been explicitly set.

[4]XML Schema, a W3C standard for the definition of XML documents.

```
Model  : header = (Header)*
         elements = (Service)*
Header : "import" T_YADLImport ";" | "include" T_OWLImport ";"
Service : "service" T_BPELConformIdentifier
         declaration = Declaration
         body = Seq

Declaration : "(" "in" input = (VariableDeclaration)* ")"
              "(" "out" output = (VariableDeclaration)* ")"
VariableDeclaration : "var" Variable ":" AbstractType
Variable : name = T_VariableIdentifier
          (complex = "." property = Property)?
Property : name = T_BPELConformIdentifier

Script : Seq | Par | Protect | TryCatch | IfThenElse | Loop | ForEach | Atom

Seq        : "{" (Script)* "}"
Par        : "par" "{" (Script)* "}"
Protect    : "protect" "{" (Script)* "}"
TryCatch   : "try" "{" (Script)* "}"
             "catch" "{" (Script)* "}"
IfThenElse : "if" "(" Expression ")" Script
             ("else" Script)?
Loop       : "while" "(" Expression ")" Script
ForEach    : "foreach" "("
             element = T_VariableIdentifier "in"
             list = T_VariableIdentifier ")"
             Script
Atom       : Assign | VariableDeclaration | Invoke
             | Read | Write | Remove | Query | Send

Assign     : Variable "=" AssignValue ";"
AssignValue : Expression | Read | Remove | Query

Invoke : "invoke"
         name = T_BPELConformIdentifier
         "(" input = (Term)* ")"
         "(" output = (T_VariableIdentifier)* ")" ";"
Read   :  "read" Term ";"
Write  :  "write" Term ";"
Remove : "remove" Term ";"
Query  :  "query" "(" subject=Term property=Property object=Term ")"";"
Send   :  "send" receiver = T_BPELConformIdentifier
          message = Term ";"
Receive: "receive" message Id = BPELConformIdentifier ";";

Value : "true" | "false" | "null" | URLConst | StringConst | FloatConst
      | IntConst | HexConst | "new" object = ComplexType "(" ")"
Term  :   Variable | Value

Expression     : (not = "!")? headTerm = ExpressionTerm
                 tails=(ExpressionTail)*
ExpressionTerm : Value | Variable | BracketExpression
BracketExpression: "(" expression = Expression ")"
ExpressionTail : operator = Operator term = ExpressionTerm

Enum Operator : And = "&&" | Or = "||" | NotFac = "!" | Add = "+" | Sub = "-"
         | Mul = "*" | Div = "/" | Mod = "%" | Lower = "<" | LowerEqual = "<="
         | Equal = "==" | NotEqual = "!=" | Grater = ">" | GreaterEqual = ">="

AbstractType: SimpleType | ComplexType;
SimpleType : datatype = StringType | URIType | BoolType | FloatType
             | IntType| HexType | DateType | TimeType | AnyType
ComplexType : ontology = T_OWLOntology ":"
              owlClassName = T_BPELConformIdentifier
```


Figure 3.4: JADL++ syntax

Currently, we are using ontologies that are compatible to the OWL-Lite standard, as these are still computable and we are interested in the usability of OWL in a programming environment rather than theoretical implications of the ontological framework.

Control Flow These commands control the execution of a script. They are basically the classical control flow operators of a simple *while*-language, consisting only of assignment, choice,and a while loop (see for example [NN99]), but are extended by commands like *par* and *protect* to allow an optimised execution.

- **Seq**: This is not an actual statement, but rather a structural element. By default, all commands within a script that contains neither a *Par* nor a *Protect*-command, are executed in a sequential order.
- **IfThenElse**: The classical conditional execution.
- **Loop**: A classical while-loop which executes its body while the condition holds.
- **ForEach**: A convenience command that simplifies iterations over a given list of items. The command is mapped to a while-loop.
- **Par**: This command gives the interpreter the freedom to execute the following scripts in a parallel or quasi-parallel fashion, depending on the available resources.
- **Protect**: This command states that the following script should not be interrupted, and thus ensures that all variables and the agents memory are not accessed by any other component while the script is executed. This gives the programmer a tool to actively handle concurrency issues that may occur in parallel execution.

Agent Programming Commands There are a few other commands in the language, namely:

- **read**: Reads data from the agent's memory, without consuming it.
- **remove**: Reads data from the agent's memory and consumes it, thus removing it from the memory.
- **write**: Writes data to the agent's memory.
- **send**: Sends a message to an agent or a group of agents.

- **receive**: Waits for a message.
- **query**: Executes a query and calls the inference engine.
- **invoke**: Tries to invoke another service.

Access to the memory is handled similarly to the language *Linda* [Gel85]. The memory behaves like a tuple space, and all components within an agent, including the interpreter for JADL++, have access to it via the **read**, **remove** and **write** commands, which correspond to *rd*, *in* and *out* in *Linda*.

The command for messaging (**send**) allow agents to exchange simple messages without the need for a complex service metaphor and thus realise a basic means for communication.

receive allows the agent to wait for a message. It should be noted here though that received messages are by default written in the agent's memory, and a non-blocking receive can thus be implemented by using a **read** statement.

The **query** command is used to trigger the inference engine for reasoning about OWL statements. The queries are encapsulated in order to allow the agents control cycle to keep control over the queries, so the agent can still operate, even if a more complex query is running.

Another important feature of JADL++ is the **invoke**-operation , which calls services of other agents. Catering to industry, we have hidden goal oriented behaviour behind a Business Process Execution Language (BPEL) [Oas07] like service invocation. Rather than only accepting fully specified service calls, the invoke command accepts abstract and incomplete service descriptions that correspond to goals, and subsequently tries to fulfill the goals. Informally, if the abstract service description only states certain qualities of a service, but does not refer to a concrete service (e.g. it only contains the postcondition of the service, but not its name or provider), the agent maps the operation to an achievement goal and can consequently employ its BDI-cycle to create an appropriate intention and thus find a matching service. However, if the service template can be matched to one and only one service or action, the agent skips its BDI-cycle and directly executes the service. This gives the programmer many options when programming agents, as he can freely decide, when to use classical strict programming techniques, and when to use agent oriented technologies.

We describe the matching algorithm in detail in the next section.

Communication JIAC V features two distinct methods for communication, which are mirrored in JADL++. The first is a simple message based

method, that works by sending directly addressed messages to an agent or to a group of agents. An agent that receives a message automatically writes the contents of the message into its memory, and from there it is up to the agent to decide what to do with the message. The advantage of this approach is that it makes very little assumptions about the agents involved and thus constitutes a very flexible approach to agent interaction. However, for complex interactions it does not offer much support.

Nowadays, the notion of service has become a very popular approach to software interaction, and JIAC V uses this notion as the predominant means of communication. The approach in JADL++ for service invocation is based on the premise that in a MAS, a programmer often does not want to care about specific agents or service instances. Rather, agent programming is about functionality and goals. Therefore, JADL++ supports an abstract service invocation, in which a programmer can give an abstract description of the state he wants to achieve, and this template is then used by the agent to find appropriate goals, actions and services to fulfill the request. Based on this template, the agent tries to find an appropriate action within the MAS. A service within JIAC V is merely a special case of an action, namely an action that is executed by another agent and thus has to be invoked remotely. However, this remote invocation is handled by the architecture, thus the programmer can use it as if it were a local action.

3.2.2.2 Semantic Service Matching and Invocation

In the previous section we have presented the syntax of a JIAC V agent, and the structure of the language JADL++. As mentioned above, JADL++ is based on a *while* language with a number of extensions, but the interpretation is rather straightforward.

The concept of data structures based on ontologies extends the goal oriented approach of agents. Informally, service matching means to have an expression that describes what the agent want to achieve (its goal), and for each service that may be applicable, to have an expression describing what that service does. To find a matching service, the agent try to find a service with an expression that is semantically equivalent to the goal expression. While other agent programming languages like 3APL [HBvdHM99] or AgentSpeak [Rao96] typically use the underlying semantics of traditional first order logic to identify matching expressions, our approach allows to extend this matching also to the semantic structures of the arguments as they are described in OWL, resulting in a better selection of matching services for a given goal.

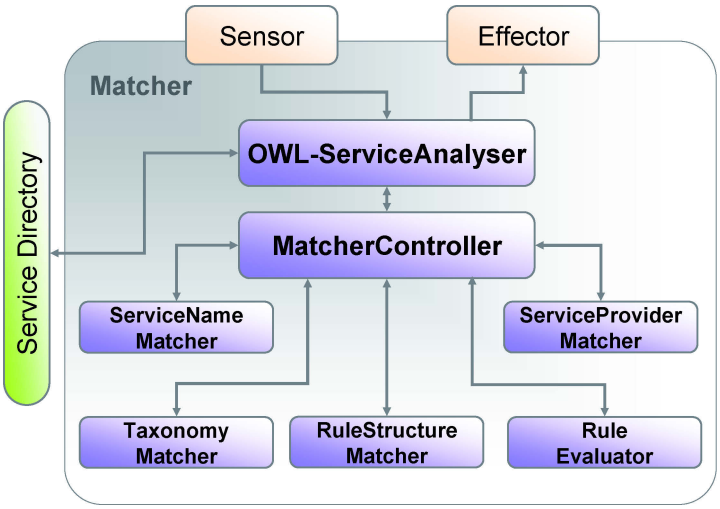

Figure 3.5: The components of the JIAC Matcher

Although OWL provides the technical basis for the description of semantic services, it does not offer a structural specification of the semantic services design itself. In order to bridge this gap OWL-S [BHS+04] has been developed that allows the semantic description of web services. The main intention of OWL-S is to offer the discovery, classification and finally invocation of resources.

Therefore, the principal challenge of the JIAC Matcher is to compare the service attributes that are embedded in the OWL-S context. In general, these are Input parameters, Output parameters, Preconditions and Effects (IOPE) of a service. Furthermore, user-defined attributes such as Quality of Service (QoS), the service's provider, the service name and the category to which the service belongs can represent additional matching information. This information can either be passed on to the JIAC Matcher in OWL-S notation or in a serialized Java class structure.

Figure 3.5 illustrates the internal layout of the JIAC Matcher components. The Sensor entity receives a service query and forwards it to the OWL-ServiceAnalyser module, which parses the OWL-S file for the relevant service attributes (if the service description has not already been passed as a Java object). The ServiceDirectory component in turn provides all existing service descriptions within the platform. It is asked iteratively for available service descriptions, and both the requested service information and the advertised description are being passed to the MatcherController. This entity finally initiates the service matching, which is divided into several categories, which we describe more detail below. The result of the match-

ing process leads to a numerical rating value for each service request/service advertisement pair. After this the service description/value pairs are sorted in a list and returned to the requesting instance via the Effector module.

In contrast to most of the other existing OWL-S service matchers [JGC+05, KFS06] the JIAC Matcher[5] implementation is able to compare services not only by input and output parameters but also by precondition and effect as well as by service name and service provider.

Depending on the information given by the service request the matching algorithm compares only one, multiple, or all parameters. Furthermore the JIAC Matcher uses a rating-based approach for the classification between a service request and a service advertisement. This means the matching procedure is separated into several matching modules, each of them returning a numerical value indicating the matching factor regarding this particular module. All of these values are added to a final result value which indicates the total matching factor of the specific service advertisement in relation to the service request. However, not every module gets the same weighting since some matching aspects are considered as more important than others. The algorithm procedure for each parameter type is explained below.

Service Name Matching The service name is the unique identifier of the service. Therefore if an agent is searching for the description of a certain service he can send a requesting template to the JIAC Matcher containing just the service name. The matching algorithm will then perform a string comparison between the requested service name and all advertised names. Additionally, if the name comparison failed, it is further checked whether the service name is contained within another one, which can indicate that the functionality of the offered service resembles the requested one.

Service Provider Matching The service provider attribute declares which agent offers the respective service. Since agents can vary in their QoS characteristics greatly, it is possible to search for services provided by particular agents. Similarly to the service name, the service provider is a unique identifier and the matching also results from a string comparison between requested and advertised service provider.

[5]The JIAC Matcher participated at the Semantic Service Selection Contest 2008, 2009 and 2010 in the context of the International Semantic Web Conference (ISWC) in Karlsruhe where it had the names *JIAC-OWLSM* and $SeMa^2$ achieved good results.

3.2.2.3 Parameter Taxonomy Matching

Input and output parameters can either be simple data types or more complex concepts defined in ontologies. These concepts are organised hierarchically and therefore the matching algorithm should not only be able to check for the exact equality of requested and advertised input and output parameters but also for some taxonomy dependencies between them. In general the JIAC Matcher distinguishes between four different matching results when comparing two concepts:

- **exact:** the requested concept R and the advertised concept A are equivalent to each other
- **plug-in:**
  - output concept comparison: concept A subsumes concept R
  - input concept comparison: concept R subsumes concept A
- **subsumes:**
  - output concept comparison: concept R subsumes concept A
  - input concept comparison: concept A subsumes concept R
- **fail:** no equivalence or subsumption between concept R and concept A has been recognized

For instance, if a service request searches a service with input parameter "SMS" and an advertised service expects a parameter "TextMessage" as an input the JIAC Matcher would analyse this as a plug-in matching as far as the ontology describes concept "SMS" as a subclass of concept "TextMessage". Since a SMS is a special form of a text message (consider other text messages like email, etc.) it is reasonable that the proposed service might be suitable for the requester although he has searched for a different parameter. This task of taxonomy matching of input and output parameters is done by the TaxonomyMatcher component within the JIAC Matcher (see Figure 3.5).

In contrast to other OWL-S service matchers, the result of a total input/output parameter comparison (a service can require more than one input/output parameter) does not lead to the result of the worst matched parameter pair but to a numerical value. Each of the four different matching levels (exact, plug-in, subsumes, fail) is mapped to a numerical result which is given to each concept pair. The total input/output concept matching result is then computed by the mean value of all concept pair results.

This approach has the advantage that the matching quality of services can be differentiated more precisely, since a mean value is more significant then a worst case categorisation. The disadvantage of this procedure however is that a service can be no longer categorised into one of the four levels that easily, because the mean value can lie between two matching level values.

Rule Structure Matching (Precondition/Effect Matching) OWL allows the use of different rule languages for the description of preconditions and effects. One of the most accepted languages is Semantic Web Rule Language (SWRL) [NPM04]. A rule described in SWRL can be structured into several predicate elements which are AND related to each other. The JIAC Matcher breaks up a precondition/effect rule into the predicates for the requesting rule as well as for the advertised rule (processed by RuleStructureMatcher component). This enables the matcher to compare the predicates with each other. If they are exactly the same, an exact matching is recognized. Since predicates are also hierarchically structured, a taxonomy matching has to be done as well. Therefore, if an exact match is not found, the JIAC Matcher tries to find out if either, a plug-in or a subsumes matching, exists. Just like the taxonomy matching of input/output concepts, each result is mapped to a numerical value. This approach applies for preconditions as well as for effects and is done by the PredicateMatcher component. Most of the predicates describe references between subjects and objects, therefore the arguments have to be checked as well. As arguments can be of different types, a type matching between advertised and requested arguments is also necessary. Again, the rule structure matching returns a numerical value which indicates the matching degree between requested effect and advertised effect.

Rule Inference Matching Preconditions contain states that the requesting instance must fulfil in order to use a service. Given an email service for example it is reasonable that the service call of sending an email must not only contain any kind of recipient address as an input parameter, but in particular a valid one (e.g. it is not malformed) . This requirement can be expressed as a precondition. Now the challenge of the Matcher is not only to check if the preconditions of the requester are the same as the advertiser's ones (which rarely might be the case) but to also verify whether the requesting parameter instances really fulfil the advertiser's preconditions. This has been done with the help of a rule engine which is able to derive if an instance fulfils a given rule. Since rules are described in

SWRL, a promising rule engine in this aspect is Jess[6] in combination with the OWL API Protégé[7]. The rule engine stores all precondition rules of the advertiser. Then the requested information instance is passed to the Knowledge Base (KB) of the rule engine. If it fulfils the rule's conditions it returns the requesters information instance as a result, which implies that the request corresponds to the advertiser's precondition (in the above example, this would be the recipient address). However, since the updating of the rule engine's KB by inserting all the ontological knowledge of the requesting instance can be very expensive this matching task is only suitable when using the service matcher directly within the requesting agent. Within the JIAC Matcher architecture the rule engine is processed by the RuleEvaluator component.

3.2.2.4 Other Features of the Language

An interesting aspect of JADL++ and the underlying JIAC V framework is that JADL++ makes no assumptions about an action, other than that the architecture is able to handle it. JIAC V was designed with the primary requirement that it should be able to handle multiple kinds of actions, be it JADL++ scripts, web services, or Java methods. The common denominator is the action– or service description which can come in two variations. There is a simple action descriptions which merely covers input, output and an action name. This is tailored for beginners and allows a programmer to become familiar with JIAC V. The more advanced version utilises OWL-S descriptions for actions and services and thus allows the programmer to use service-matching and the BDI-cycle.

Nevertheless, both action-descriptions are abstract in the sense that they only require some part of the agent to be responsible for the execution. Thus only this component in the agent has to know how to access the underlying technology. For example, the interpreter for JADL++ is the only part that knows about the scripting-part of JADL++. There can be a component dedicated to the invocation of web services. Or there can be multiple components for multiple web services. And so on. This allows us to easily and quickly get access to multiple technologies from JIAC V, and at the same time always use the same programming principles for our agents.

[6] `http://herzberg.ca.sandia.gov/`
[7] `http://protege.stanford.edu/`

3.2.3 Available Tools and Extensions

Our former framework JIAC IV already came with an extensive tool support which has been described in Section 5.1. Consequently, the design of JIAC V was always guided by the requirement to have existing tools be applicable to JIAC V. An overview of these tools and their role in the development process for JIAC applications is given in Sections 5.2 and 5.3.

3.2.3.1 Standards Compliance, Interoperability, and Portability

In terms of standards, JIAC V has changed considerably from its predecessors, as we focussed on the use of software standards heavily. However, as of today one important standard, the FIPA speech act, is not explicitly supported. It is of course possible to design messages that comply with the standard but it is not a requirement. However, the underlying technologies are all based on today's industry standards, such as OWL and OWL-S for ontologies, but also Java Management Extensions (JMX) [Per02] for management functionality, Java Message Service (JMS) [MHC00] for message delivery, and web service integration. For portability to small devices, we have developed a cut-down version of JIAC called MicroJIAC.

3.2.3.2 MicroJIAC

The MicroJIAC framework is a lightweight agent architecture targeted at devices with different, generally limited, capabilities. It is intended to be scalable and useable on both resource constrained devices (i.e. cell phones and Personal Digital Assistants (PDAs)) and desktop computers. It is implemented in the Java programming language. At the moment a full implementation for Connected Limited Device Configuration (CLDC) devices is available, which is the most restricted Java 2 Platform Micro Edition (J2ME) configuration available.

The agent definition used here is adapted from [RN03]. It is a biologically inspired definition where agents are situated in some environment and can modify it through actuators and perceive it through sensors. Thus the framework is also split into environment and agents. The environment is the abstraction layer between the device and agents. It defines life cycle management and communication functionalities. These functionalities include a communication channel through which the agents send their messages.

Agents are created through a combination of different elements. The predefined element types are sensors, actuators, rules, services and components. Actuators and sensors are the interface between the agent and the environment. Rules specify reactive behaviour and services define an

interface to provide access to specific functionalities. Finally, components maintain a separate thread and host time consuming computations. All elements are strictly decoupled from each other and are thus exchangeable.

In contrast to JIAC, MicroJIAC does not use an explicit ontology language, goals or an agent programming language such as JADL++. Furthermore, agent migration is restricted to Java configurations which support custom class loaders and reflection. It should be noted here that both architectures, MicroJIAC and JIAC V, are targeted at different fields of application and have different development histories. However, they use a common communication infrastructure to enable information exchange between agents.

Chapter 4

Agent Methodology

Now why should the cinema follow the forms
of theater and painting
rather than the methodology of language,
which allows wholly new concepts of ideas
to arise from the combination
of two concrete denotations of two concrete objects?
(Sergei Eisenstein)

4.1 Basic Methodology

In this chapter, the basic methodology is described, which has been developed and used in industry and teaching projects at the DAI-Labor as well as in programming competitions. Here, we understand methodology as a set of activities, called disciplines, that reoccur in every project. With every discipline a number of work products have to be generated or refined, stipulated by the iterative process model.

4.1.1 Process Model and Work Products

MIAC embraces 6 disciplines (see Figure 4.1), which have been bound together in the process model:

- **Requirements Management**: At the beginning of each project and at the beginning of each iteration the requirements and use cases of all stakeholders in the development process are collected, analysed, specified and prioritised. The requirements of the users are indirectly assigned to the developer by the business model of the customer. The user herself usually expresses requirements in a later iteration when

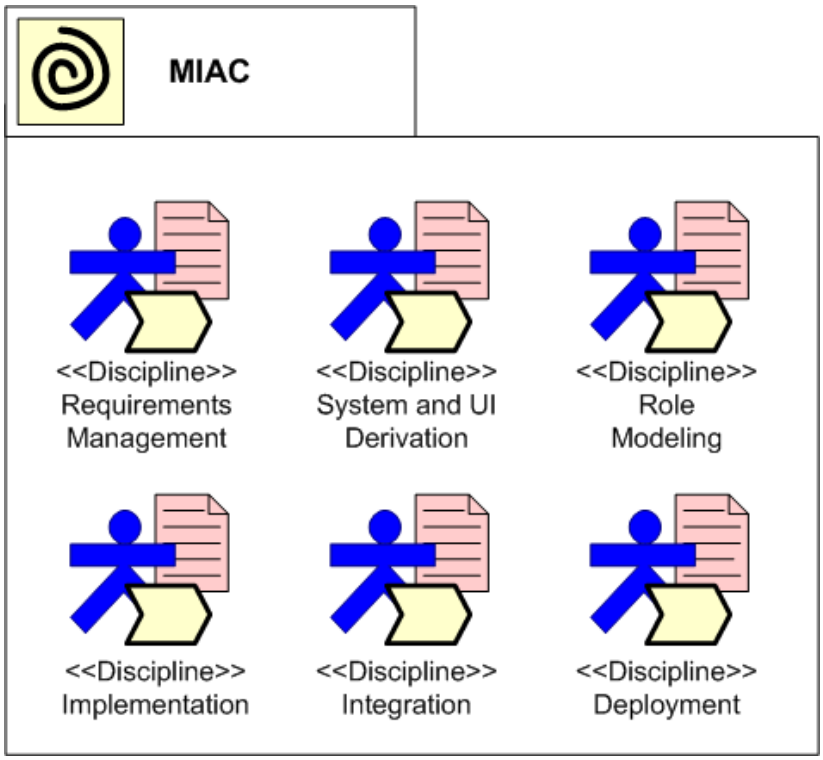

Figure 4.1: Disciplines of the MIAC process

it comes to acceptance and usability tests. The requirements of a developer normally show after the first iteration when the feasibility and the approach becomes clearer for the developer. It is always necessary to accommodate and/or re-prioritise requirements based on the results of an iteration or incidents that occurred in the meantime.

- **System and UI Derivation**: The requirements are refined and formalised to the extend to which they enable the creation of a system architecture and the user interface and where acceptance tests can be derived. A first system architecture normally shows a number of agents that have been found in an ad-hoc manner. The user interfaces start with a number of prototypes for the use cases that with higher priority. They will be refined and extended and integrate with each other in every following iteration.

- **Role Modeling**: Identified agents are decomposed into roles. There are three main questions to answer:

  – Which roles exist (distribution of functionality)?

– Relationships of roles with other roles (interaction between roles)?
– Relationships of roles to the overall system and its environment and infrastructure, also extern resources, legacy systems and dedicated machines

The answers show up in the form of role structures and courses of actions as well as interaction protocols and supporting roles that are not directly related to required functionality.

- **Implementation**: During implementation we use the JIAC Agent Framework, JADL, AUnit- and JUnit-Testing (AUnit tests testing plans whether the effect will be fulfilled when the precondition holds), and the UI language MAP-AIDL [RCF+05].
- **Integration**: The Multi-agents system (MAS) is assembled and tested. Integration tests check the interplay between components, agents, modules, and libraries.
- **Deployment**: Using the parts which have been assembled during integration, one or more MAS are deployed. They run on the platform of the customer or users and are ready be evaluated under real terms.

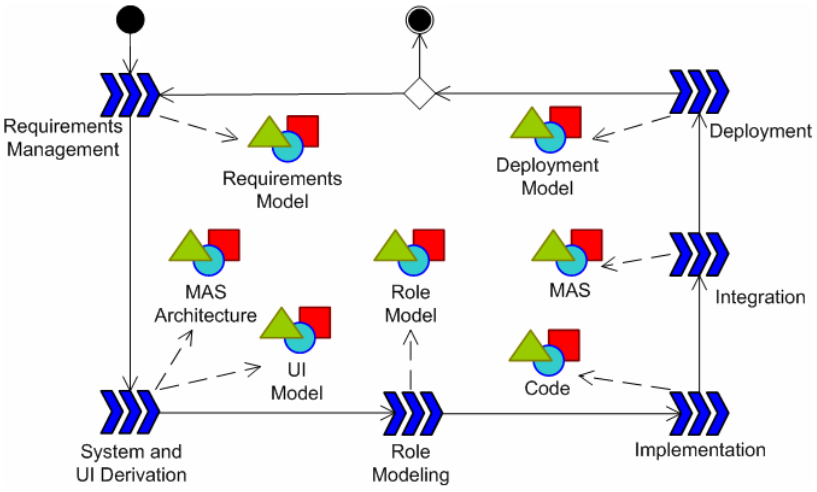

Figure 4.2: MIAC Iterative Process Model in SPEM [Obj05] notation

The MIAC disciplines are embedded into an iterative overall process (see Figure 4.2). At the end of each iteration there is an evaluation which determines the further direction of the project.

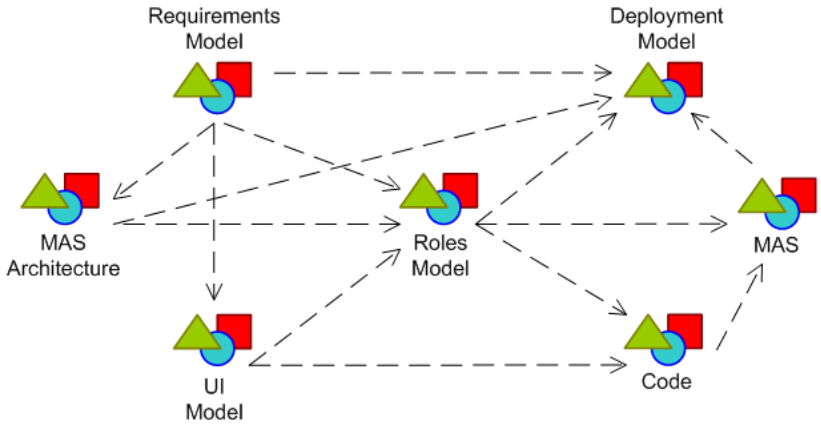

Figure 4.3: Dependencies of work products in MIAC

The disciplines generate work products which can be used as input for successive processes and which may trigger particular work packages of different processes (see Figure 4.3). Each work product is as complete and consistent as required. *Required* means that, for example, a single role model can be incomplete or less detailed; however, it can contain the essential features that are needed to start working on the following process without the fear of completely dropping the work product when the model changes. See the JIAC IV contribution to the Multi-Agent Programming Contest [HHK08]: the *Walking* capability has not been completely analysed when we started to design and implement it. Successive iterations analysed the *why*, *when*, and *where* the agent should walk. But this did not mean that the whole work was for the birds. The following sections describe the MIAC desciplines in greater detail.

4.1.2 Disciplines

4.1.2.1 Requirements Management

The discipline **Requirements Management** comprises of a number of activities (see also Figure 4.5):

- **Requirements elicitation** (or change after an iteration): identification of the desires, potential needs and expectations of all stakeholders. Desires, needs and expectations are then transformed into rough requirements. Requirements are collected from the user and customer points of view and are the basis for an initial ontology.

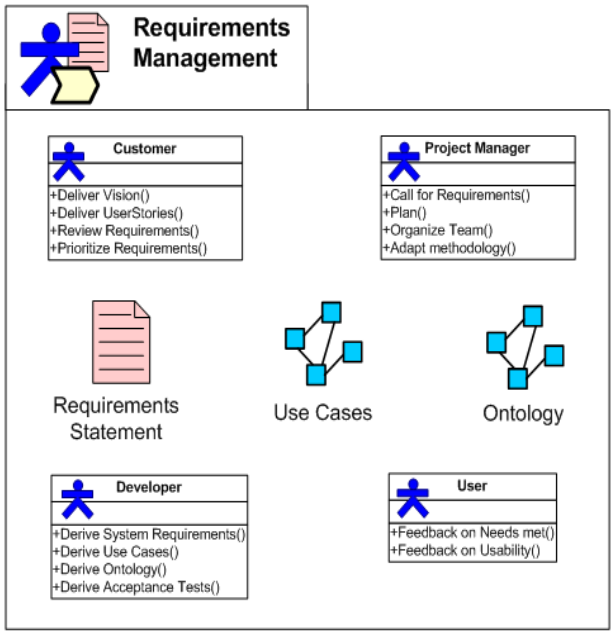

Figure 4.4: Discipline "Requirements Management"

- **Reuse** of existing requirements descriptions: advancement of quality by reusing of approved high quality requirements (requirements that have gone through the whole process of analysis and specification, that have been implemented and evaluated, and have shown that they hold in the users' or customer's eyes). This activity raises the productivity of the development team by avoiding the same mistakes during repeated identification, analysis and specification. It also reduces time-to-market and development costs.

- **Requirements analysis**: to study, categorise, decompose, organise, model, quantify, refine, prioritise, and justify requirements as well as to track each requirement to its root. Textual requirements are transformed into semi-formal (such as UML diagrams) or formal (in case of a formal method or language) requirement definitions. Priorities and validities of the requirements must be negotiated with the stakeholders. Estimates and guesswork of requirements must be checked. Raw requirements are worked on until they have reached a required rate of quality concerning clarity (for example the absense of ambiguity), completeness, consistency, correctness, feasibility (technically, financially, temporally), verifiability, comprehensibility. All requirements must be understood adequately in order to specify them correctly.

For example, in the Multi-Agent Programming Contest [HHK08] we identified some basic requirements at the beginning: *Explore the world*, *Pick and Drop gold*, *Transport gold to the depot and score*. When we analysed the requirements we found that both *Exploring* and *Transporting* depend on a basic functionality: *Walking*. This was the "quality level" to start thinking about and implementing an agent capability that allows the agent to walk through a grid world.

- **Requirements specification**: the creation and provision of approved and changed requirements respectively. Important products are the "Requirements Statement" or "System Requirement Specification", Use Cases and the domain model (ontology).

- **Iteration**: Requirements management is subject to iteration like all processes in software engineering. Forgotten, unclear, and controversial requirements must be identified and followed up before they can be released to the development team. Frequently, the specification process is accompanied with architectural and UI prototypes so that some requirements may refer to the next phase "System and UI Derivation".

The following activities are orthogonal to all of the above activities in that phase:

- Manage the recording and storage of all requirements and their metadata (with the help of the right tool)

- Manage the negotiation and approval of requirements with customers, users and developers

- Organise access to requirements according to roles in the development and according to different point of views (status, responsibility, ...)

- generate status reports regarding the requirements (number and completeness of requirements and packages)

- Allow the tracking of requirements through linking of the work products (e.g. the Goal is $x \rightarrow$ Constitutes the Requirement $y \rightarrow$ Will be provided by Agent $z \rightarrow$ has been implemented in Role r)

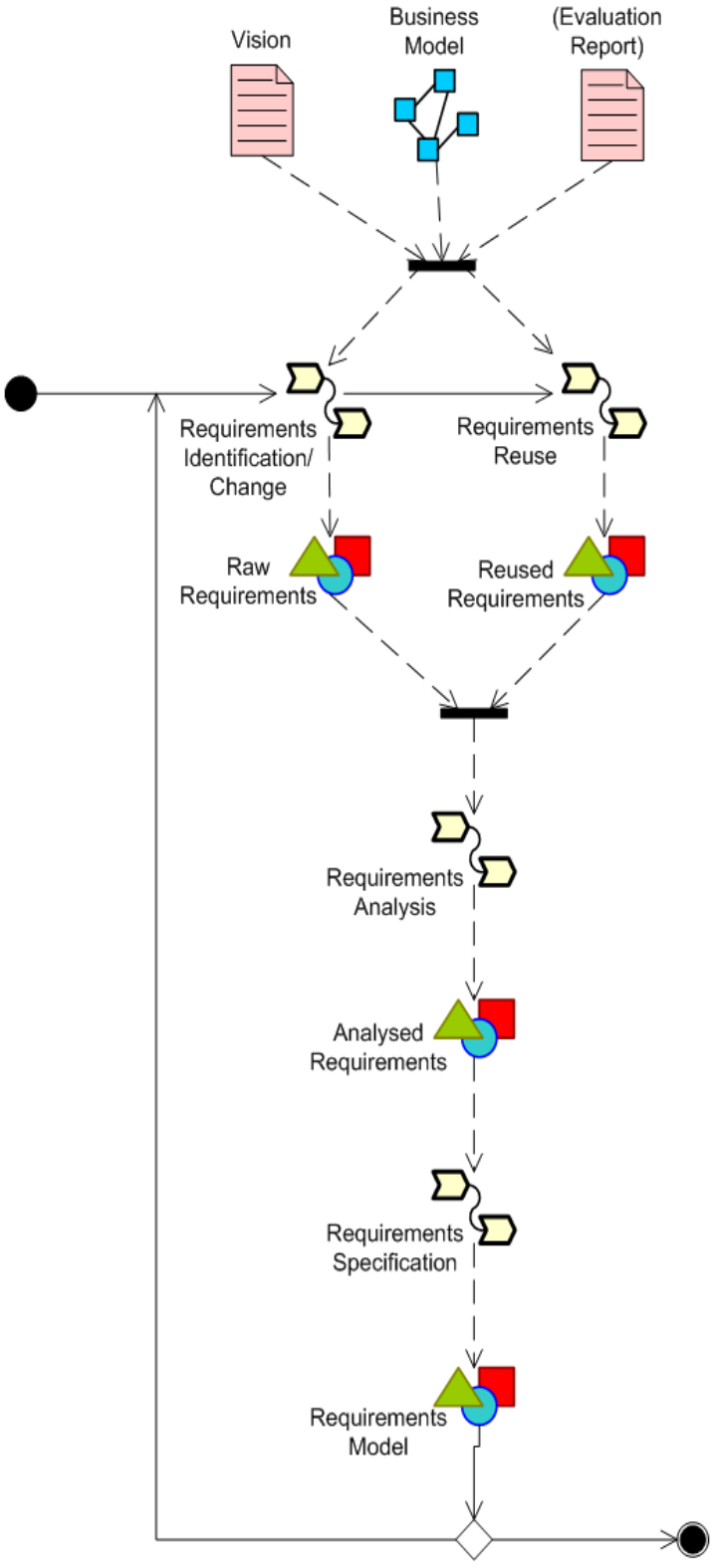

Figure 4.5: Requirements Management Workflow

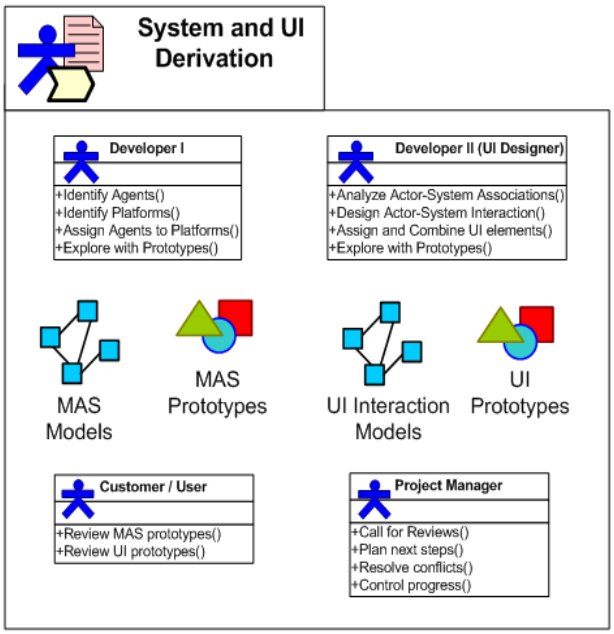

Figure 4.6: Discipline "System and UI Derivation"

4.1.2.2 System and UI Derivation

The discipline "System and UI Derivation" refines the requirements specification concerning the system architecture and UI design. The input for this discipline (especially *Use Cases*) is coming from the requirements management discipline. The results are different prototypes which are used to evaluate the system architecture and the user interface. The prototypes also serve as input for the successive considerations (e.g. integration, deployment, distribution, specification of non-functional tests).

A system architecture consists of number of agents that stand for a requirement or a number of requirements (the agent will do this). So, the activities here are to identify agents, organise them on agent platforms and to find links to other agents and the environment. This will form a first MAS, and, after a number of iterations, your final MAS.

In addition, the prototypes are used to define further requirements regarding the security and management infrastructure and to ensure usability and manageability (see also Figure 4.7).

The idea behind is here that generally a system (and UI) is not created from scratch, but that ready building blocks coming with the framework are employed. In other words, the main features of the system already exist and only the the missing parts and the differentiation needs to be

addressed. Those are first discovered, and then then designed, implemented, and evaluated them.

4.1.2.3 Role Modeling

Role modelling constitutes the actual analysis and design of the MAS or of the agent-based service. Here, a role denotes functionality, the associated interactions, and the required infrastructure. As tools, static UML diagrams for the decomposition of roles into subroles and their composition into higher-level roles, and, on the other hand, UML diagrams for modelling the dynamic behaviour of the MAS or the services can be used.

The following two work packages describe the analysis and synthesis of roles (see also Figure 4.9):

- **Role Extraction and analysis**: Identified agents are analysed concerning their intended functionality (the role that they play in the MAS). We break the role down until we get an atomic function (or action). If we already have found a solution for a role, a set of roles or a part of a role we may stop here with the analysis. For each atomic action a suitable algorithm has to be found. Dependencies to other roles, clarify their needs for interaction and conversation protocols need to be defined next. External (non-agent) systems and resources must be made accessible. The demand of organisation between single roles must be analysed in order to determine which type of agent infrastructure should be provided. Role decomposition should use formal methods (Task trees, goal-oriented, or UML class diagrams). For an overview about actions and interactions, the UML communication diagram (since UML 2.0; or collaboration diagram in previous versions) is recommended. Cources of actions and conversation protocols can be best displayed using UML activity and sequence diagrams. Structural and behavioural elements are then assign to concepts of the MAS meta-model (see also Figure 3.1).

- **Role Synthesis and MAS Revision**: Atomic actions, interactions and infrastructure roles can now be combined. The identified roles can be assigned to a single agent or any number of agents if necessary. Very often the first (ad-hoc) architecture will change. While a minimal solution is to leave the system architecture untouchedit is generally better to use the possibility of relative freedom of combinations to re-design the MAS and take aspects of scalability and distribution of functionality into account. If the generic agent has planning capabilities the synthesis of agent roles is not necessary, but

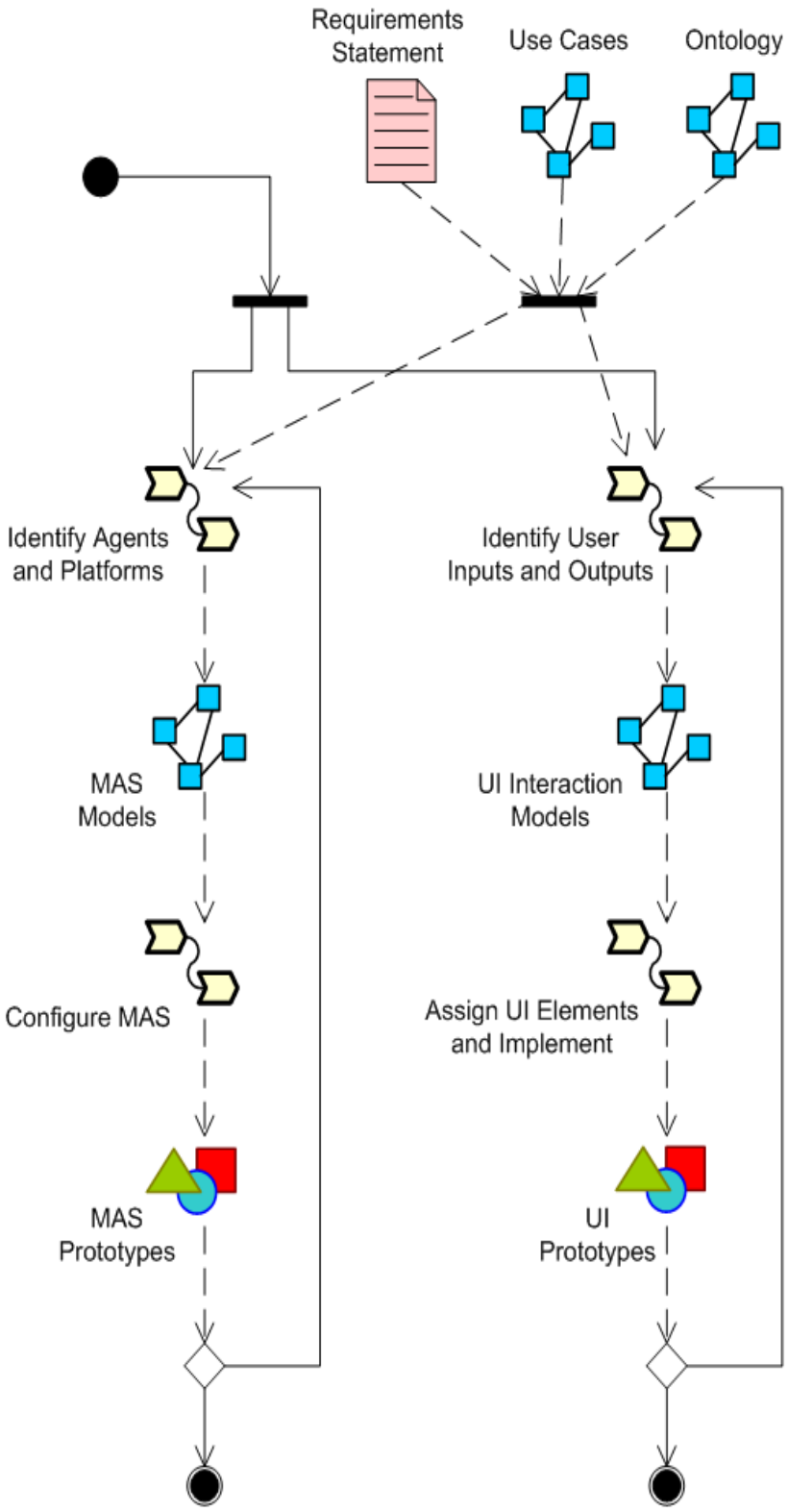

Figure 4.7: System and UI Derivation Workflow

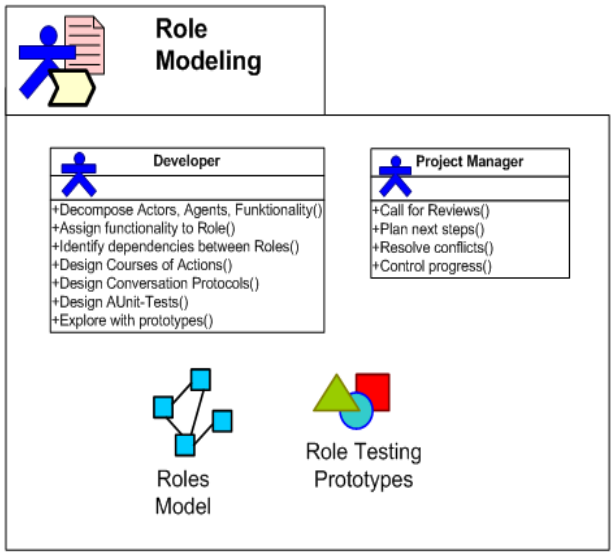

Figure 4.8: Discipline "Role Modeling"

the functional capabilities must just be available in the MAS. different combinations of roles should be tested, especially with non-functional test such as performance tests and reliability tests.

4.1.2.4 Implementation

The implementation discipline consists of all activities that transform the role model and UI into executable code. The definition of single work packages is extremely dependent on the MAS meta-model so that we show the implementation discipline using the JIAC IV meta-model. The implementation discipline is traditionally the best supported discipline with a lot of guidance such as tutorials, examples, and programmers' guides, and methods and regulations of how to code.

We define four major working packages with regard to the JIAC IV meta-model (see also Figures 3.1 and 4.12):

- **Ontology Implementation**: Ontologies are implemented in JADL. They contain the specified categories with their attributes and constraints. Additionally, functions and comparisons between categories may be defined. The JIAC IV Toolipse generates code during requirements elicitation and the modelling the applicational domain, which can be modified now.

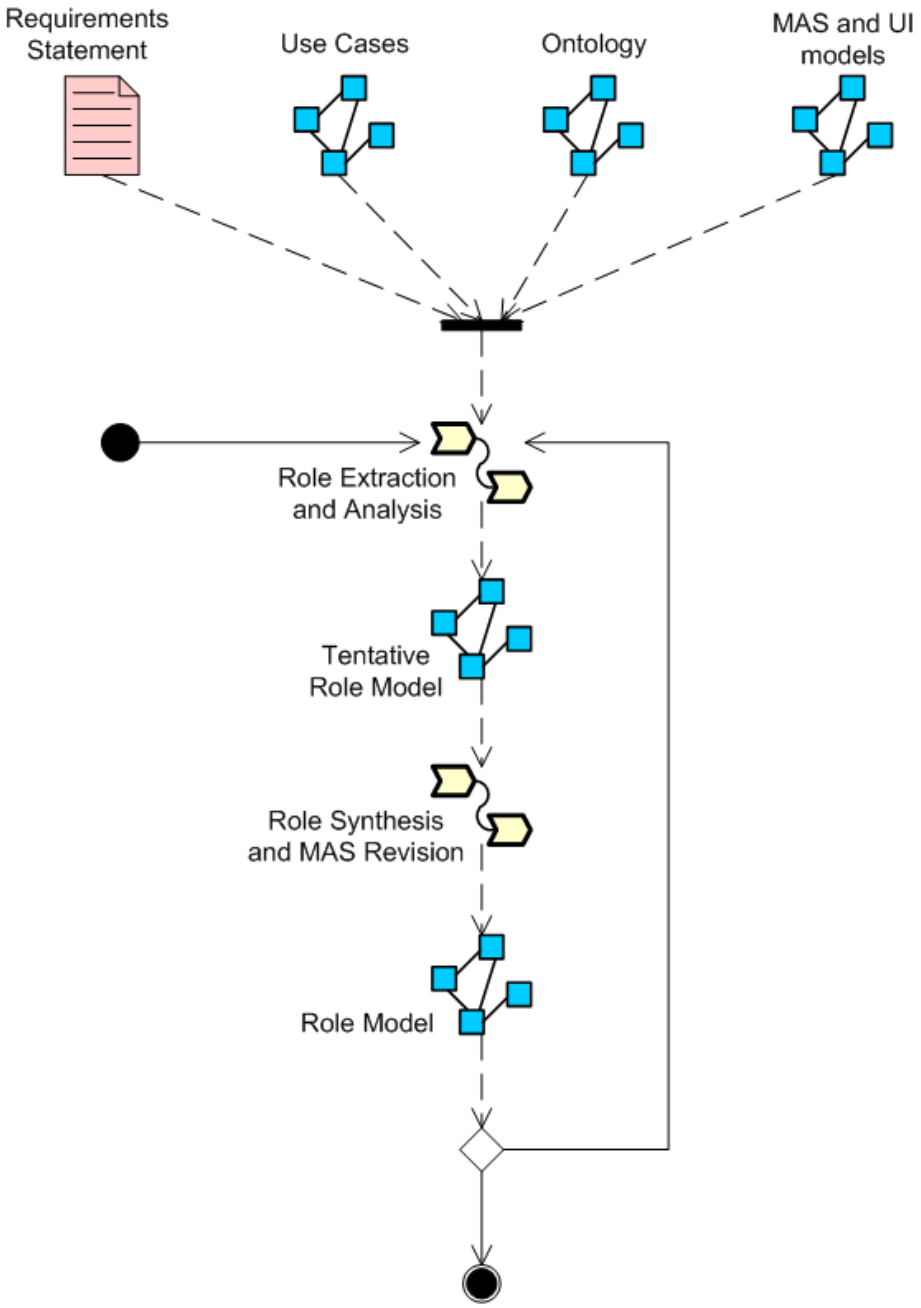

Figure 4.9: Role Modeling Workflow

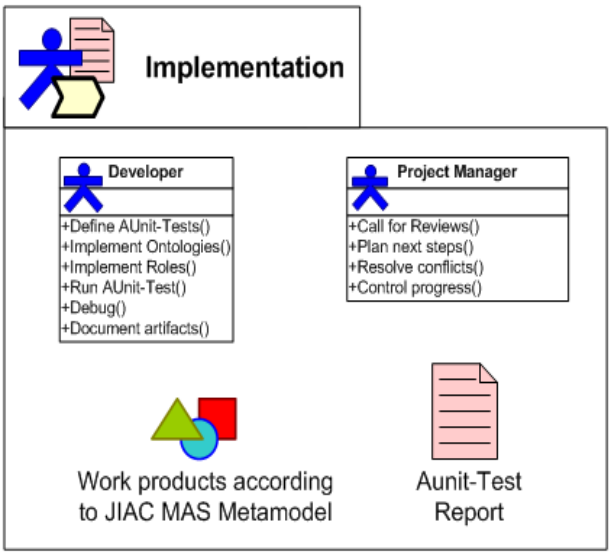

Figure 4.10: Discipline "Implementation"

- **Plan Implementation**: The factual and practical knowledge is implemented in JADL. AUnit-Test are defined for local action and services.
- **AgentBean Implementation**: Parts of the functionality will be implemented using Java. AgentBeans wrap the Java APIs, while JUnit tests can be used to ensure a high quality of the AgentBean implementation.
- **Role, Agent and Platform Configuration**: Finally, so called AgentRoles are created where the agents and agent platforms that make up your program or service are configured rather than programmed

  Beware: the decision whether an interaction between two roles is realised as a JIAC service or not depends on several factors and should have been made during role modelling. The basic criterion depends on the system architecture: if two roles are played by different agents, an interaction between these two roles is implemented as JIAC service. If scalability is emphasised, this should be used. On the other hand, if two roles are played by one agent or if performance has a higher priority then AgentBean-Message-Passing or the factbase with reaction rules are the implementation of choice for this interaction. Maximal

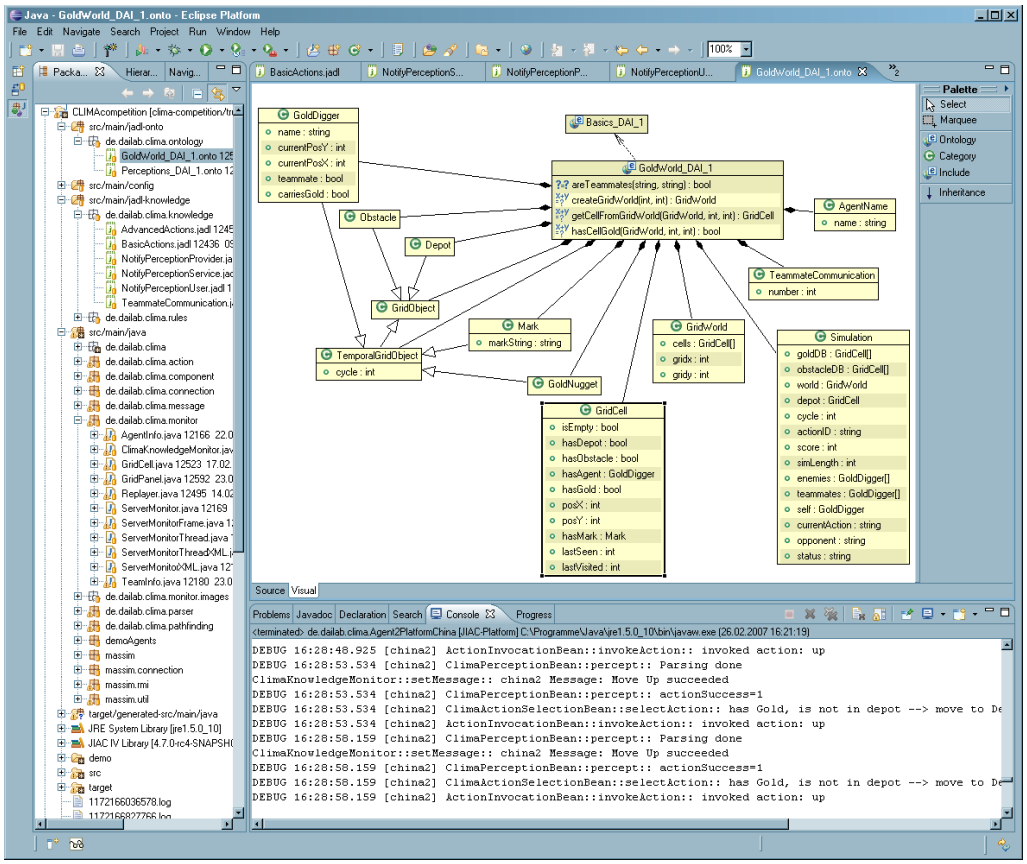

Figure 4.11: JIAC IV Toolipse - Eclipse-based IDE, the figure shows the visual ontology editor

flexibility is reached if the two roles are implemented using all concepts at the same time but this of course raises development expenses.

4.1.2.5 Integration

Integration is an activity which combines and tests components of an application or service to subsystems or the whole system. Integration should be done continuously (see e.g. Continuous Integration [Fow00]) in order to discover and correct errors and incompatibilities early during development. Integration happens from roles to roles, roles to agents, agents to agents, and agent to MAS. AUnit tests can check the interactions between roles, the distribution of roles over several agents and the distribution of agents over

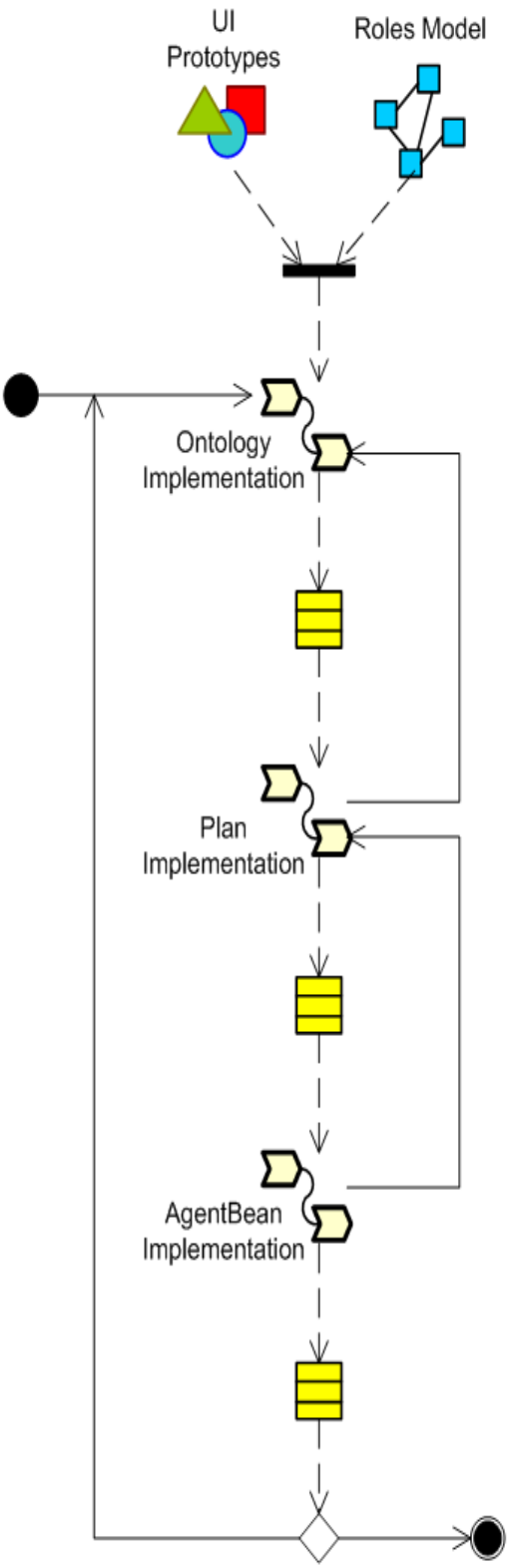

Figure 4.12: Implementation Workflow

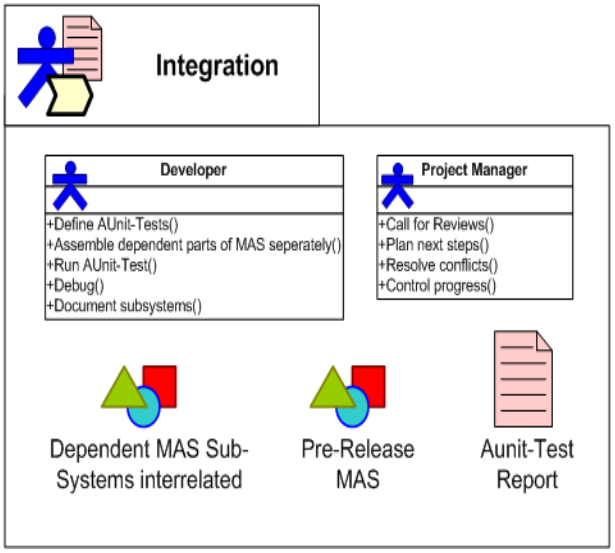

Figure 4.13: Discipline "Integration"

several platforms. Furthermore, the UI implementation has to be tested with respect to functionality and usability.

The integration discipline comprises two major work packages (see Figure 4.14):

- **Role Integration**: Integration of roles to (higher level) roles, agents, and platforms. Directions of how to assemble roles, agents and the MAS can be derived from the role model and the MAS architecture. The integrated system needs to be tested according to the coordination of its components and the functioning of the overall system.
- **UI Integration**: Merge the UI code with the functionality. Here we have to test the correct interaction of UI with functionality on the one hand, and a user driven usability test on the other hand.

4.1.2.6 Deployment

Deployment means to deliver and install the system at the customer site.

Discipline Deployment consists of four work packages (see Figure 4.16):

- **Adaption to Target System**: basically, the customisation of the MAS is configuration. Usually, concrete and correct information about host, network, firewalls, etc. can be changed using property configurations.

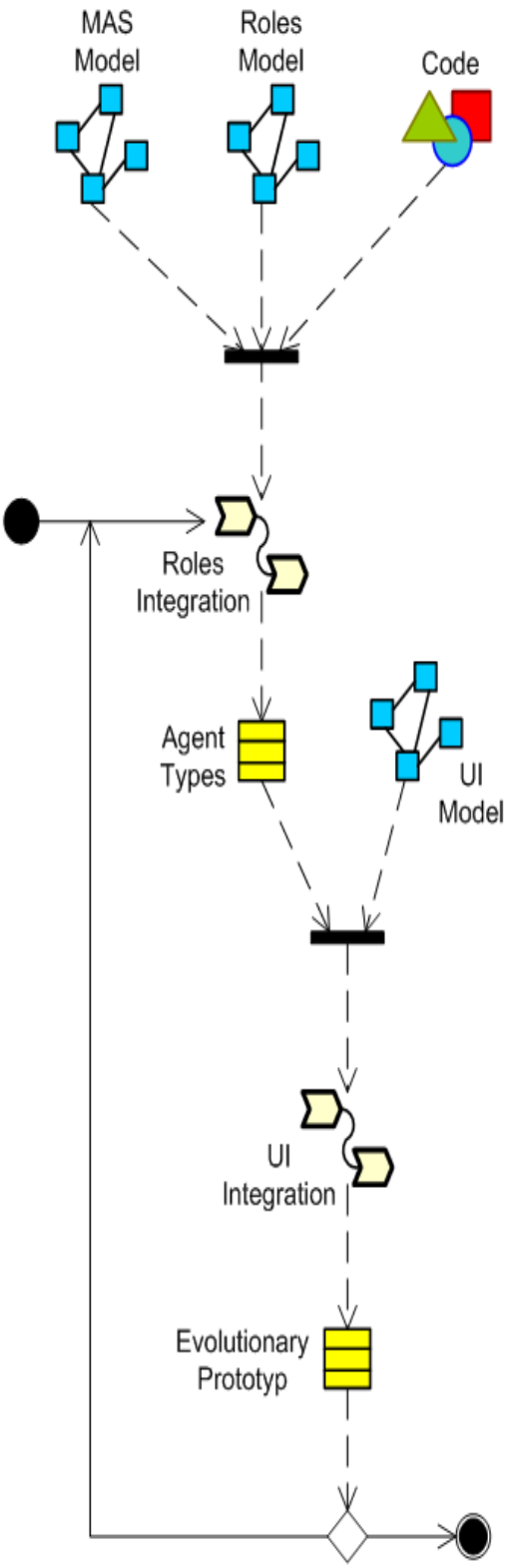

Figure 4.14: Integration Workflow

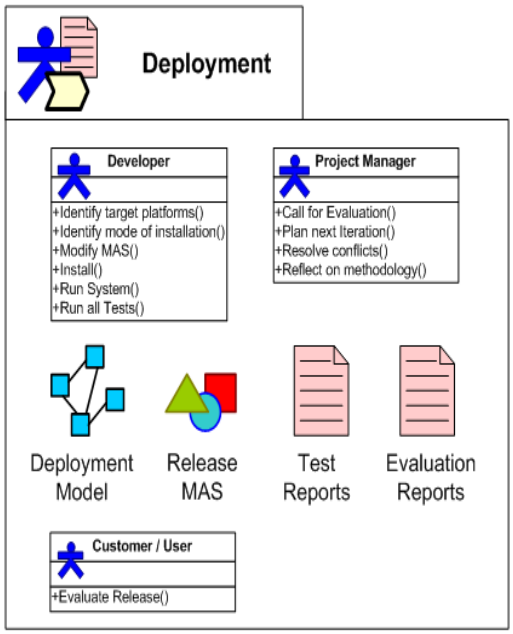

Figure 4.15: Discipline "Deployment"

- **Packaging**: Assembly of code, libraries, configurations, together with install and start scripts
- **Distribution**: delivery of the packages over different ways according to what has been agreed in the requirements. You may ship the MAS via storage media, download, or even automatically via an application server.
- **Installation**: Run the installation scripts and then start the system.

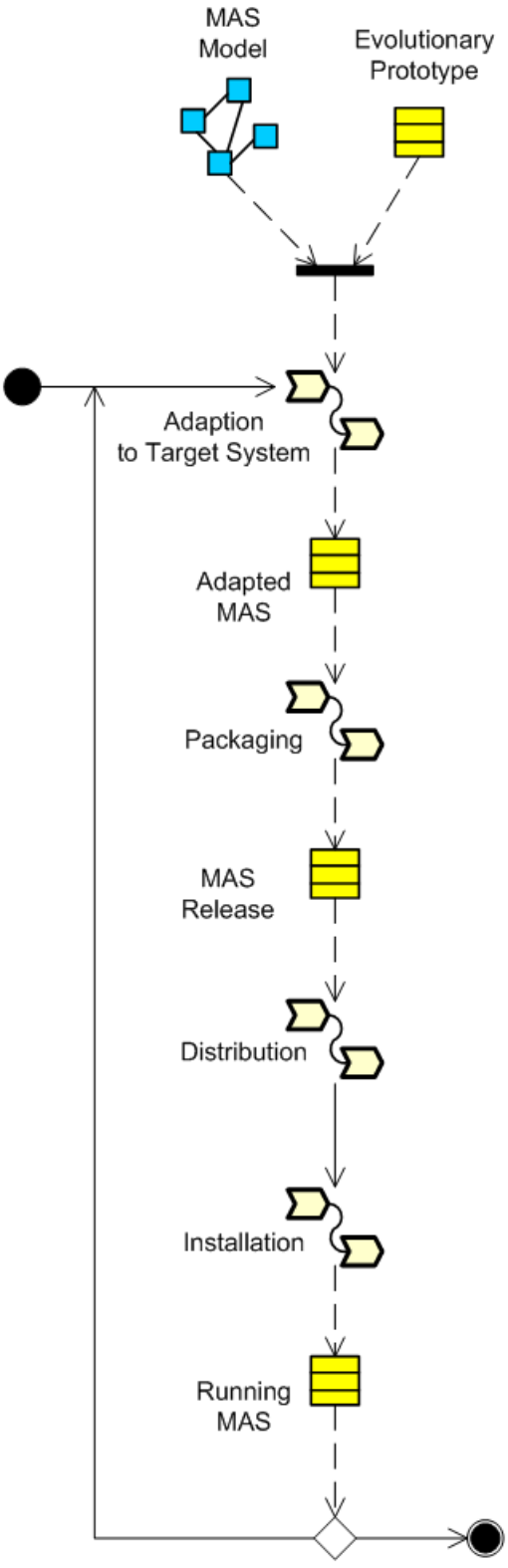

Figure 4.16: Deployment Workflow

4.2 Ontology Engineering

An important part of every software engineering project concerns itself with the description and definition of the entities that it uses in computation. Concepts can be captured using taxonomies, class diagrams, domain specific languages, or ontologies. They all have in common that they model entities, their attributes and their relations that can be observed in the problem domain.

Agent methodologies often utilise ontology languages to capture domain knowledge, due to the conceptual similarity to knowledge-based systems. However, it is often not clear to the developer or programmer *how* this is done, and even what is meant by ontology is often not clearly specified. For example, the web standard ontology language OWL [MvH04] consists of variants with different expressiveness; additionally, the "niceness" (ontology ratio) of an ontology [DPF+05] depends on whether individuals are part of the ontology and make seperation of concerns hard for the software engineer or programmer, when it comes to data handling apart from its conceptualisation in domain models, database tables or strict ontologies.

One of the guiding principles is usually reusability of domain knowledge. To this end, concepts like *upper ontologies* are often suggested (see Suggested Upper Merged Ontology (SUMO) [PNL02] or Cyc [MWK+05]). While valid on an theoretical level, this principle leads to computational expensis that are out of scope of most software projects. This is not an argumentation against reuse at all. Although it is easier to build ontologies from scratch and often difficult to understand someone else's model, reuse in ontology engineering avoids wasting effort, supports standardisation and often results in higher quality of the model.

Why (and when) should I use ontologies? An ontology is a good tool to analyse a domain, make domain assumptions explicit and share a common understanding of the structure and relationships of domain entities among people and programs. It is also a good way to separate static domain knowledge from dynamic operational knowledge, which in turn enables easier reuse of domain knowledge in other applications.

The exact ontology language, however, is not just a theoretical question; tools, computational efficiency, and the application where the domain knowledge is used also play an important (and maybe more important) role when deciding on an ontology language or ontology reuse. Finally, it is a good idea to publish ontology beyond the project scope, namely in an ontology repository [NGM08, TNNM10].

In the following, a generic description of how to define ontologies is given:

4.2.1 Ontology Preprocessing

A good start is always to find sources in natural language that describe the domain as whole or parts. This can be text documents, manuals, or oral descriptions. Drawings or diagrams where links in a domain and from the domain to peripheral areas are shown are also very helpful.

Then make a glossary were you record relevant terms and their definitions. This step in ontology preprocessing can also be done (semi-)automatically by linguistic or philological software ("concordancers") that generate lists of word occurances together with their contexts. However, human expertise is necessary to revise the glossary (or concordance) to eliminate "synonyms" (words with the same meaning, but different formative), "homonyms" (words with the same spelling, but with different meanings) and "semantic pleonasm" (words with overlapping meaning), which are causes for the major pitfalls when developing an ontology.

Another ontology revision analyses the scope of the ontology, i.e. what is the developer going to use the ontology. In this step write down a catalogue of questions the ontology should answer. Use this catalogue often throughout the development of the ontology. Also, answer the questions of who will use or maintain the ontology in terms of persons or software agents.

4.2.2 Ontology Construction

Ontology construction or generation starts with selecting a relevant term (usually a noun) from the glossary and looking it up in reusable ontologies. If it is already existing and you want to reuse it, reference the ontology where the term occurs. If not, add the term to your ontology by deciding whether the term is a class or an individual. A good hint for a class is always when you can name a number of examples that are instances of this class.

Then relate the term to other terms in the ontology or in a referenced ontology by identifying specialisations and generalisations (class–sub-class relationships). The common pitfall here is that people mix up subclass-of (*is-a*) with part-of (*has-a*) relationships: a "window" is a part of a "house", not a special house type. And also "MyHouse" is not a subclass-of a house, but an instance of class "house".

Next, find attributes that describe details of the selected term, usually depicted by adjectives, and are modelled as object or data properties (also called slots), relating the term to other terms or to a range of possible values. For example, in a text about windows you find out that they are

held in place by frames. Windows can have wooden, plastic or metallic frames. So the class *Frame* would have a property *made-of*.

The trick here is to choose exactly those attributes that are needed to describe a number of individuals and distinguish them from one another. The catalogue of question should be consulted frequently and extended if necessary. Also think about the number of values a slot can have (cardinality) and what the value type is (such as *string*, *boolean*, or *number* in case of data properties or classes in case of object properties).

Finally, if a certain value range consists of an enumeration of individuals, add them to your ontology.

4.2.3 Ontology Enactment

When an ontology is to share concepts of a certain domain for a certain purpose between people and systems, it is necessary to enact the ontology somehow. In literature, the term *Ontology Repositories* (e.g. Hartmann et al. [HPGP09]) is coined for a controlled way to publish ontologies, in order to use, apply and reuse it in different contexts. In the ontology repository ontologies are stored and it usually comes with a number of process models that say how to publish, maintain, find and use existing ontologies.

Another way of how to enact an ontology is to provide the API in a programming language that is used in the project. This is a comfortable way for programmers to use the power of ontologies without dipping to much into theory or without the necessity to learn yet another language. E.g. in [Cow09] gives an overview about how semantic web ontologies (written in OWL) can be bound to Object-Oriented Programming (OOP), namely to the Java programming language. The intention is clear: an easy way to read/write, send and persist objects. Instead of saying

```
addTriple(personURI, nameURI, "Hirsch")
```

the user wants to say

```
person.setName("Hirsch")
```

There are clear similarities between the two approaches that enable the mappings back and forth. But if you can avoid multiple inheritance, avoid it because of ambiguities in behaviour and features of the classes. Also be careful when declaring properties: properties in OWL can inherit other properties and can be attached to more then one class. Not so in Java.

Basically, there are two ways to do the mapping: *Annotation-based mapping tools* and by *code generators*. Both ways have their pros and cons.

Annotation-based mapping tools allow annotation to class, variable and method declarations that are then automatically mapped to according ontologies. This method is best used when you use existing ontologies and want to enable your code (existing or new) to use these ontologies. The drawback is when the ontology does not exist you write two taxonomies and only the mapping is done by the tools.

Code generators offer a comfortable way to generate the API from an existing ontology. This method is very efficient when you have large and elaborate ontologies. Loose range type declarations in the ontology are the downside of code generators. Prominent ontologies such as Friend Of A Friend (FOAF)[1] or Dublin Core (DC)[2] use the *Literal* type from the Resource Description Framework (RDF) schema definition; this can be literally anything.

From capturing the things in the domain's static structure to capturing the procedural knowledge and domain dynamics brings us to the next section of this thesis.

4.3 Capturing How Things Are Done

One of the major challenges in software engineering is to capture not only the goals and objectives of the involved stakeholders but also *how* these goals and objective are attained and achieved. Besides clarifying and analysing the desired outcome, using, for example, an goal-oriented approach such as the *I*-framework* [Yu95], it is also very often necessary to find out how domain experts and businesses work or ought to work, to the extent that a developer can design software agents or that a tool can generate program code that exactly behaves like them.

The field of Business Process Management (BPM) deals with relating and structuring activities or tasks in order to develop products and services, in an application domain independent way. Usually, business models are created and consumed by business analysts and managers. But when software developers are involved the models must be understood and realized also by them. This requires that all stakeholders share a common semantics and a large number of modelling languages exist to formalize this semantics: for example, BPMN [OMG11] and Event-driven Process Chain (EPC) [Men09] as graphical notations, or XML Process Definition Language (XPDL) [KLL09] and BPEL [Oas07] as machine interpretable languages.

[1] http://xmlns.com/foaf/spec/20100809.rdf

[2] http://dublincore.org/2010/10/11/dcterms.rdf

By now, the mapping from BPMN to BPEL has been implemented in numerous tools, greatly assisting the business architect in the creation of BPEL processes. However, most of these tools are tailored especially for this transformation, neglecting the original purpose of BPMN: Providing a language independent process model. To address this shortcoming, a pure BPMN editor is needed, being dynamically extensible with several export features and added editing functionality. In this paper, we present a tool that follows this approach, not only providing a compelling transformation to BPEL but at the same time being extensible to other languages.

The goal of process modelling, as of Model Driven Engineering in general, is to provide an abstract view on systems, and to design those systems in a language and platform independent way. For that purpose BPMN [Obj06] has been standardised by the Object Management Group. It can be understood intuitively by all business partners, even those who have great knowledge in their domain but do not know too much about Service Oriented Architecture (SOA) or programming in general. At the same time, BPMN is formal enough to provide a basis for the later implementation and refinement of the business process. Given a respective mapping, a BPMN diagram can be used for generating readily executable code from it. A brief introduction to BPMN is given for instance in [Whi04a].

Today, BPMN and the specified mapping to BPEL are supported by a growing number of tools — we will have a closer look on some representatives later in Section 4.3.4. However, the problem with the majority of existing tools is that while they do provide the usual transformations from BPMN to BPEL, they are focused only on this one aspect of BPMN. Often the editors and even the underlying metamodels are adapted to BPEL in many ways. While this may be desired in order to provide highest possible usability and to support the user in the creation of executable BPEL code, the consequence is that business process diagrams created with these tools can neither be transformed to other executable languages, nor can the process model be used with other tools that might provide different transformations. Thus, while process modelling and BPMN should be independent of a specific executable language, the *tools* are not.

The solution to this problem is to keep both the underlying BPMN metamodel and the diagram editor free from influences from the BPEL world and to use pure BPMN instead, so that diagrams created with such a tool will be truly independent of any concrete language — apart from what influenced the BPMN specification in the first place. Based on this, several mappings to different target languages can be implemented and integrated into the editor as plugins, which may also contribute to the editor in order to support the business architect with language-specific editing features.

Following this approach, the Visual Service Design Tool (VSDT) has been implemented as an Eclipse plugin, inherently providing the necessary modularity, as we will see in Section 4.3.1. For the export of BPMN diagrams to executable languages a transformation framework has been designed, which we will describe in more detail in Section 4.3.3. The transformation has been subdivided in distinct stages, so that significant parts of it are reusable, e.g. the challenging transformation of the control flow. Thus the actual mapping to a given language can be integrated in a very straight-forward way. While the usual mapping from BPMN to BPEL has been realised as a proof of concept (see Section 4.3.3.2), the main intent behind the VSDT is to provide a transformation from business processes to multi-agent systems such as the JIAC language family [Ses02]. The respective mappings are currently under development and will be discussed briefly in Section 4.3.3.4. Our ultimate goal is to provide transformations not only in different, but also in heterogeneous systems — just like they are used in the real business world. Future work in order to achieve this goal, as well as a conclusion to this paper, will be discussed in Section 4.3.5.

4.3.1 The Visual Service Design Tool

The first version of the VSDT has been developed as a diploma thesis [Küs07] in the course of the Service Centric Home (SerCHo) project at Technische Universität Berlin (TUB) in early 2007. As the work continued it matured to a feature-rich BPMN editor with an extensible transformation framework and has already been used in a number of service orchestration scenarios in the context of a smart home environment[3], one of which will be shown later in Section 4.3.3.3.

4.3.1.1 The Metamodel

The BPMN specification [Obj06] describes in detail how the several nodes and connections constituting a BPMN diagram have to look, in which context they may be used and what attributes they have to provide. However, it does neither give a formal definition of the syntax to be used for the metamodel, nor an interchange format, e.g. using an XSD. Thus the editor's metamodel had to be derived from the informal descriptions in the specification. As it was our main concern to keep as close to the specification as possible, we decided not to reuse the existing Eclipse SOA Tool Platform (STP) BPMN Editor, which uses a simplified model of BPMN[4].

[3] `http://energy.dai-labor.de`

[4] `http://www.eclipse.org/stp/bpmn`

Instead, almost every attribute and each constraint given in the specification has been incorporated into the metamodel, allowing the creation of any legal business process diagram. Still, some attributes have not been adopted in the metamodel: For instance the possibility to model nested or even crossing Lanes has been dropped, as it turned out that this feature seems to be virtually never used in practical business process design. Further, redundant attributes, such as the Gateway's `defaultGate` attribute, are emulated using getter methods to prevent inconsistency in the diagram model.

Concerning the transformation to BPEL and other executable languages, which in most cases are block-oriented, an extension to the usual BPMN metamodel has been designed, featuring equivalents to the basic block structures, such as sequences, decisions, parallel blocks and loops. These elements are described in a separate metamodel, extending the editor's metamodel. They are used only during the transformation process, especially for the mapping of the structure, as we will see in Section 4.3.3.

4.3.2 The BPMN Editor

Like many others, the VSDT editor has been created using the Eclipse Graphical Modeling Framework (GMF), automatically equipping the editor with numerous features, such as support for the Eclipse properties, outline and problem view and unlimited undo and redo, just to name a few. Being embedded in the Eclipse workbench, the editor is easy to use while at the same time providing a powerful tool for professional business architects and service developers.

While GMF provides a solid basis for the editor, several customisations have been made to the code, further improving the editor's overall usability and supporting the creation of new business processes. For example, the generated property tables have been supplemented with custom-made sheets, in which the several attributes are more clearly arranged. For managing the non-visual elements given in the BPMN specification, such as Properties, Messages and Assignments, a number of clear and uniform dialogs has been created. The various constraints given in the specification were translated to several audit constraints used to validate a given business process diagram. A screenshot showing some of the editor's features can be seen in Figure 4.17.

As already mentioned, the VSDT was designed to be a pure BPMN editor and independent of BPEL, so the business process diagrams can be transformed to other languages, too, given the respective export plugins. Of course, the downside of this approach is that the editor lacks built-in

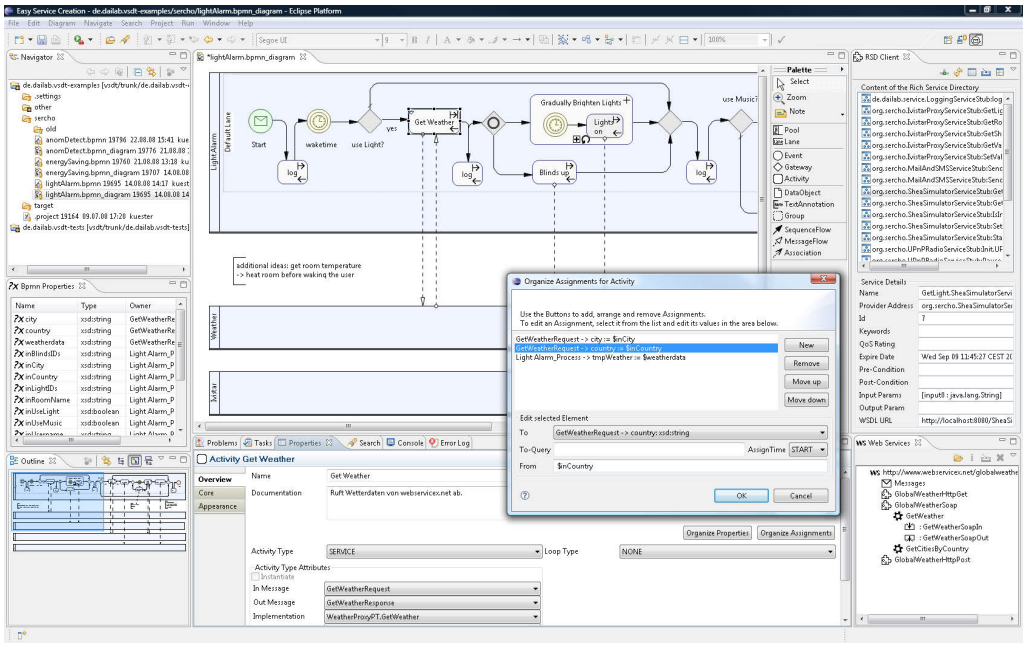

Figure 4.17: The Visual Service Design Tool, with colors and additional markers being enabled. Clockwise: Editor view, RSD client, Web services view, Organize Assignments dialog, customized property sheet, visual outline, properties inspector, navigator.

support for BPEL, e.g. the editor itself does not validate an expression given in the diagram to conform to the BPEL syntax. However, it is possible to supplement the editor with additional plugins, which can contribute e.g. to the property sheets or provide whole new views with language-specific functionality.

One example of how the VSDT can be extended with features specific to a certain target language — in this case: BPEL — is the Rich Service Directory (RSD) View, which can bee seen in Figure 4.17, too: A client for the RSD, a special kind of Web service repository. Using the RSD View, existing Web services that have been registered at the RSD server can be inspected and imported into the diagram. In the process, an Implementation object is created for the Web service as well as a set of Message objects, matching the service's input parameters and result. Optionally, also a new Pool will be created for the service, which can be connected to the currently selected Activity via a pair of Message Flows. Further, the Implementation and the Message objects will be associated to the Activity and its type will be set to SERVICE. Thus, the orchestration of existing Web services in a

BPEL process can be simplified greatly. Similar features can be created for other target languages, too.

Once the business process diagram is completed it can be validated and exported. As the VSDT is intended to provide export features to arbitrary target languages, and to support the tool smiths in the creation of these features, we have created an elaborate export framework, which we will have a closer look at in the next section.

4.3.3 The Transformation Framework

The core of the Visual Service Design Tool clearly is the transformation to executable code. While by now the transformation to BPEL is the only one that can be conveniently used in practice, and thus will serve as an example later in this section, there are currently several other transformations under development.

The transformation framework has been designed from the very beginning to be as *extensible* and *reusable* as possible. For that purpose the process of transformation has been subdivided into several stages, which are sequentially applied to the input model:

1. *Validation*: Validate the input model.
2. *Normalisation*: Prepare the input model for transformation.
3. *Structure Mapping*: Convert the input model to a block-like structure.
4. *Element Mapping*: Perform the actual mapping, create target model.
5. *Clean Up*: Remove redundancies, improve readability, etc.

The transformation is operating on a copy of the model to be transformed, which can be modified in the course of the transformation without affecting the original diagram. The several stages are realised either as a set of graph transformation rules, a top-down pass through the input model, or a combination of both. For the graph transformation rules the Tiger EMF Model Transformation (EMT) [BEK+06] has been employed, providing a fast pattern matching and backtracking algorithm for Eclipse Modeling Framework (EMF) models. In EMT, rules can be specified using a convenient graphical editor. For the VSDT, however, the EMT has been modified so that instead of a Left Hand Side (LHS) with Negative Application Conditions (NACs) and a Right Hand Side (RHS), the rules feature an LHS, NACs and an `execute` method, which may contain arbitrary Java code. Given the several cases to consider in BPMN this has proven more feasible.

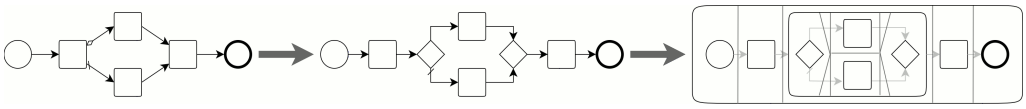

Figure 4.18: Simple example of normalisation and structure mapping.

4.3.3.1 Stages of the Transformation

The Validation, Normalisation and Structure Mapping are to a great part independent of a specific target language, and in most cases the standard implementations provided with the transformation framework can be used. However, it can be advantageous to extend them with additional checks and rules.

For instance, in the *Validation* stage, all identifiers are validated to contain only characters that are legal with respect to the given target language, which can be achieved by extending the standard implementation and using a respective regular expression for the validation of names. Further, the validation includes a pass through the model, checking if each element needed is in place, thus reducing the number of checks necessary in the actual transformation, and providing clearer error messages to the user in case something is missing.

The intent of the *Normalisation* stage is to put the process diagram in a uniform form, and to transform it to what in the following will be referred to as the Business Process Diagram (BPD)'s *normal form*, a semantically equivalent representation of the diagram following more strict structural constraints than those given in the BPMN specification. The transformation rules that are used in this stage are rather simple. For instance, one rule will check if there are any Activities with multiple incoming or outgoing Sequence Flows attached to it, in which case a Gateway of type XOR or AND will be inserted in between, depending on whether the Sequence Flows have any conditions. Another rule will insert a "no-op" Activity in between any two Gateways that are directly connected by a Sequence Flow. The advantage is that after the application of the normalisation stage there will be much fewer cases to consider in the structure mapping, which will be described in the next paragraph. A simple example of the consecutive execution of normalisation and structure mapping can be seen in Figure 4.18.

One of the challenges in transforming BPMN to executable languages is the mapping of the process model's graph-oriented structure to a more rigid block-oriented structure. For that reason it is of great benefit making this part, the *Structure Mapping*, independent of the actual target language, so it can be reused in mappings to other block-oriented languages. We decided

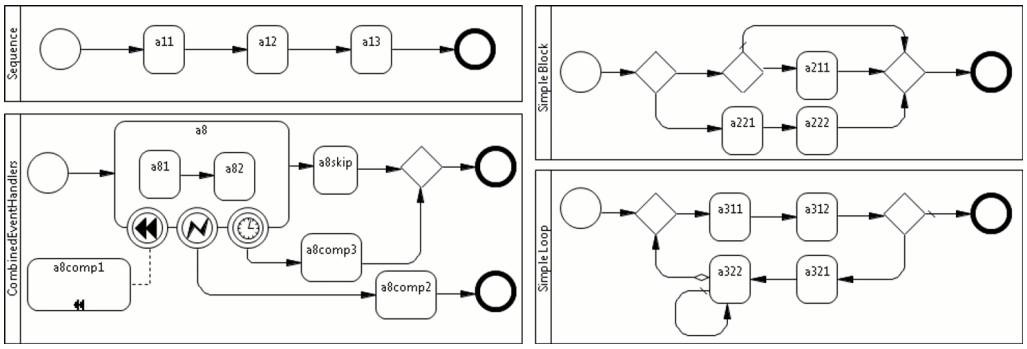

Figure 4.19: Some examples of transformable BPMN graphs.

to follow a *Structure Identification* strategy [MLZ05], being independent of BPEL's Link element. As mentioned in Section 4.3.1.1, the transformation is using an extension of the BPMN metamodel used in the editor, allowing the introduction of additional elements representing sequences, blocks for parallel and alternative routing, loops, and event handler blocks, i.e. the basic building blocks of block-oriented languages. Now, in the structure mapping stage, the model is searched for graph patterns which are semantically equivalent to these blocks, e.g. two Flow Object nodes connected with a Sequence Flow, or two Gateways connected by a number of branches of Flow Objects. When such a pattern is found, it is replaced with the respective structured element, removing the involved Sequence Flow edges in the course, which are then no longer needed (their conditions, if any, are preserved in the newly created structured elements). With these elements themselves being Flow Objects again, the rules of the structure mapping are applied until the entire process within each Pool has been reduced to a single complex element, e.g. a sequence, or until it can not be reduced any further due to structural flaws. Some examples of BPMN graphs that can successfully be mapped to equivalent block structures and further to executable BPEL code can be seen in Figure 4.19. Of course, this stage can be adapted or entirely omitted, too, if the target language is structured differently.

After the rule-based structure mapping, in the *Element Mapping* stage, the several BPMN elements can be mapped in a relatively simple top-down pass through the model. We decided to use a top-down pass instead of rules in this stage, as it is faster and easier to maintain, but the framework does allow for other implementations as well. As the element mapping is very dependent on the actual target language we will go further into detail later, regarding the transformation to BPEL.

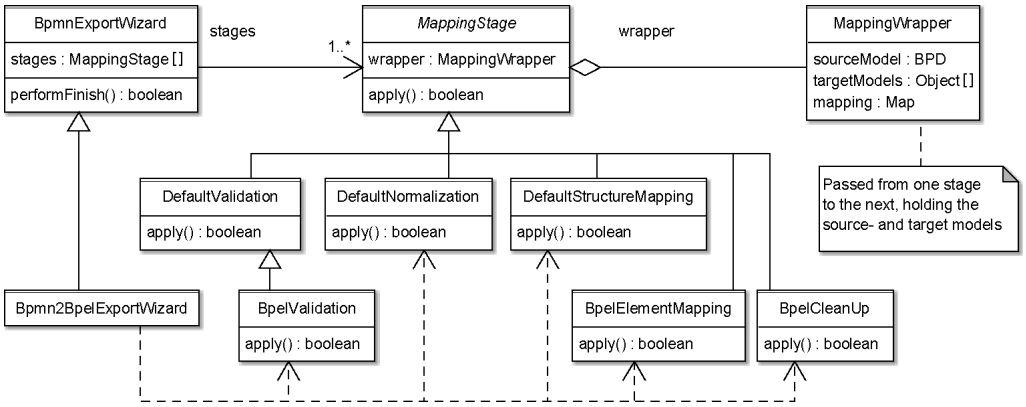

Figure 4.20: Essential classes of the transformation framework, including the BPEL case.

Finally, in the *Clean Up* stage, a set of rules is applied on the newly created target model. While this stage is optional, it can be of great use for improving the readability of the generated code while at the same time keeping the required logic out of the earlier stages. For instance, nested sequences will be flattened, or sequences that hold a single element are replaced by that element itself. Further, elements that resulted from no-op Activities inserted in the normalisation stage should be removed again in this stage. As this stage operates on the target model, is has to be implemented anew for each target language.

For implementing a specific transformation, all that has to be done is to specify the element mapping, which can be done in any desired fashion by extending a special abstract class (see Figure 4.20). In case the target language uses different block concepts, the structure mapping has to be adapted, too, but should still be reusable to some point. In the majority of cases, implementing the other stages is optional.

4.3.3.2 Transformation to BPEL

The transformation to BPEL presented in this work covers nearly the entire mapping as given in the BPMN specification [Obj06, Chapter 11], including event handlers, inclusive OR and event-based XOR Gateways, just to name a few. Still there are some elements for which the mapping is not given very clearly, such as TIMER Start Events, independent Sub Processes or multi-instance parallel loops. While these elements will be transformed as described in the specification, the resulting BPEL processes will require some amount of manual refinement. Besides the BPEL process files a Web

Service Description Language (WSDL) definitions file is created, holding the message types derived from the process properties and the input and output messages and interfaces (port types) for the several Web services being orchestrated by the process. Still, the WSDL's binding and service blocks and necessary schema types, if any, can not be generated automatically yet, due to insufficient information in the source model. We are currently investigating ways of extending the BPMN metamodel in order to include more information in the model and at the same time making it more independent of the BPEL language.

In the validation used for the transformation to BPEL, all identifiers are tested to contain only characters that are legal with respect to BPEL. Additionally all expressions used e.g. in Assignments and loop conditions are scanned for occurrences of Property identifiers. So if a Process `Proc` has a Property `foo` and there is an Assignment with an expression like `"foo+1"`, the expression will be changed to

`"bpws:getVariableData('Proc_ProcessData','foo')+1"`.

Thus the user does not have to care about the way Properties are represented with messages in BPEL but can use a Property's plain name in expressions.

4.3.3.3 Example

The following example will show one of the scenarios being used in a smart home environment in the SerCHo project. The resulting BPEL processes were validated and tested with the *ActiveBPEL* designer and process engine.[5]

The BPMN diagram in Figure 4.21 is showing a "Light Alarm" process, that is used to open the blinds in the user's room to wake her up in a more pleasant way than the usual alarm clocks do. For that purpose, firstly information on the current weather is retrieved using an external Web service. Thereafter, based on the weather data, either the sunblinds are opened, or the ceiling light is turned on, or both. In case the user does not get up, which is checked using an Radio-Frequency IDentification (RFID) based localisation solution, the stereo is turned on, playing her favourite song or alternatively an unpleasant alarm sound.

For each of the above devices — blinds, lights, localisation, and stereo — Web service interfaces were written, so they can be integrated in a BPEL process. While the WSDL file that is used by the process, holding the definitions for the various orchestrated services, has to be extended with

[5] `http://www.activebpel.org`

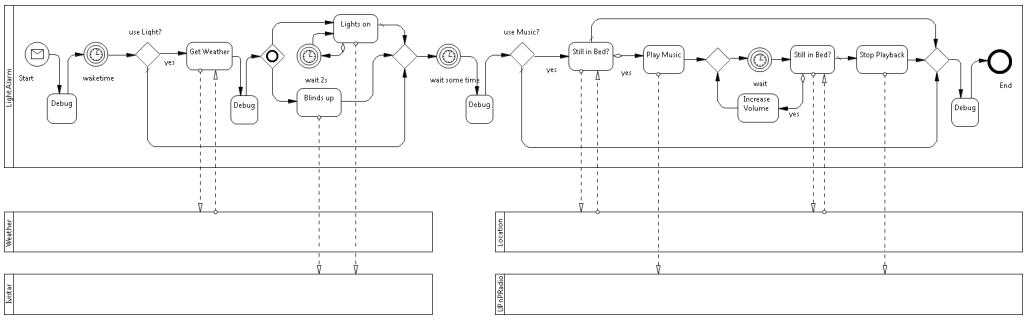

Figure 4.21: "Light Alarm" Example Process

the service bindings, the generated BPEL code resulting from this example is readily executable.

4.3.3.4 Transformation to JIAC

Concerning our goal of transforming BPMN diagrams to MAS the work is still at an early stage. First, a *normal form* for BPMN diagrams has been investigated, to facilitate the mapping [EHKA07]. Later, the first steps of the actual mapping have been developed, basically mapping Pools to agents, Processes and Flow Objects to the agents' plans and the control flow, and Message Flow to the exchange of messages between the agents [EKHA07].

A first prototype targeting the agent framework JIAC IV [Ses02] has already been implemented. As the theoretical part of the mapping is not yet fully matured, there is still some work to do. However, with the given transformation framework every addition to the mapping can quickly be adopted.

4.3.4 Related Work

The Business Process Modelling Notation has been adopted in a large number of tools. Although many of these are merely diagram drawing tools and do not support the transformation to BPEL, let alone other languages, there are some powerful tools as well. In the following we will introduce some of these. A more extensive list can be found at `http://www.bpmn.org/BPMN_Supporters.htm`.

With the free *eBPMN*, Soyatec provides a very nice BPMN editor, but it does not implement the mapping to BPEL.[6] The same applies to the free community edition of eClarus' *Business Process Modeller*, while the

[6] `http://www.soyatec.com/ebpmn`

commercial *SOA-Architect* version provides a transformation to BPEL, although it seems to have some limitations.[7] A very mature BPM product can be found in the *Intalio BPMS*.[8] BPEL code is generated on-the-fly and can be deployed to the Intalio process engine. However, the mapping of workflow structures is limited, e.g. we found it impossible to merge a branch originating from an event handler back into the normal flow. Another limitation arises from the tight coupling to the in-house BPEL engine, which is using some proprietary extensions. Further, Intalio has donated parts of the code to the Eclipse STP. While the *STP BPMN Editor* itself does not provide a transformation to BPEL, Giner et al. were able to combine it with the *BABEL Bpmn2Bpel* tool [GTP07], yet both the editor and the transformation tool are using very simple metamodels.

Concerning the transformation from graph-oriented to block-oriented process models, as in the BPMN to BPEL case, Mendling et. al. have evaluated several transformation strategies [MLZ05], ranging from a straightforward mapping of BPMN Sequence Flows to BPEL Links, similar to the one in [Whi05], to a more sophisticated *Structure Identification* strategy, like the one applied in this work, or *Structure Maximisation*, as followed by Aalst and Lassen [vdAL08]. In their theoretically well-founded, pattern-based transformation from Petri nets to BPEL, they focus on the readability of the resulting code. However, they do not regard how highly *unstructured* workflows can be transformed to structured ones. As pointed out in [RM06], there is a "mismatch" between BPMN and BPEL, both on the domain representation and the control flow level, that is not easily to overcome. Many authors have investigated whether different graph patterns can be transformed to an equivalent structured form [KtHB00, LK05, SO00], and came to the conclusion that even slight unstructuredness can require the introduction of additional variables or the duplication of parts of the workflow, even though the workflow models used in these works are much simpler than BPMN. For structuring such workflows, Koehler et al. present a rule-based transformation based on continuation semantics [KH04]. Another approach is followed by Ouyang et al., using BPEL event handlers as a form of `goto` command [ODBtH06]. Their examples show how complicated a simple workflow can become when being structured.

Thus, as workflow design will be facilitated greatly if the user is not restricted to the use of block-oriented processes, a transformation of unstructured workflows to readily executable code will be highly desirable, so that such processes can be created by means of Model Driven Engineering.

[7] `http://www.eclarus.com`
[8] `http://www.intalio.com`

4.3.5 Conclusion

In this paper the VSDT has been introduced: a BPMN editor featuring a state of the art transformation to BPEL, while at the same time being easily extensible with export functionality targeting other languages. The editor itself has been designed to be language independent, so it can be used for generating code for any language, given that a respective mapping from BPMN to that language exists. Transformations implementing these mappings can be plugged in to the VSDT together with additional editing features helping the user in the creation of diagrams to be exported to that language. For supporting the developer of these plugins, the VSDT comes with a transformation framework, based on the EMT graph transformation tool. Being subdivided into several stages, large parts of it can be reused throughout different mappings, such as the refactoring of the process graph to block-oriented structures.

With respect to its BPMN editing functionality and the transformation to BPEL, the VSDT does not have to hide behind its commercial competitors. Implementing the mapping to BPEL as given in the BPMN specification, the tool can be used to generate readily executable code. Still it is recommended to validate the results with a native BPEL editor: While the creation of processes will be easier and faster using the VSDT, its desired independence of a specific language prohibits some BPEL specific features, such as editing assistance for assignment expressions. However, due to the plugin architecture provided by the Eclipse platform, such functionality can be added together with the actual transformation features.

As the key feature of the VSDT is the extensibility with additional export features, further transformations from BPMN to executable languages are currently under development. One of the main goals of our research in this field is a mapping from BPMN to multi-agent systems, combining the intuitive design of business processes with the flexibility of software agent.

4.3.6 Future Work

Some work still can be done in the field of transformation of unstructured processes. Currently the tool can handle slightly unstructured workflows, such as one Gateway being used for merging multiple decision blocks, but will fail when faced with e.g. overlapping blocks or multiple exits from a loop. Here, further evaluations of the different possibilities to handle such workflows and ways of adapting them to the more complex BPMN diagrams will be necessary.

Concerning the transformation to BPEL, the support for complex data types will need further refinement. Here, the *Rich Service Directory* introduced earlier will be of great use, providing the necessary information about the involved Web services. Finally, the mapping to multi-agent systems has to be completed, and mappings to further languages will be evaluated.

Chapter 5

Agent Tools

Do not wait; the time will never be "just right."
Start where you stand, and work with whatever tools
you may have at your command,
and better tools will be found as you go along.
(George Herbert)

5.1 Basic Tooling

Developing agent-based applications without an IDE is often difficult and error-prone. Providing good IDEs to the developers eases agent programming and enhances the quality of the output, which perhaps helps the agent-oriented paradigm to become more widely accepted. To achieve this important objective, we have developed Toolipse, a fully featured IDE prototype, based on the Eclipse platform, for the development of JIAC applications. Toolipse has been used and evaluated in teaching and a number of projects in different domains and helped their users creating pinpoint solutions.

Toolipse is already the third attempt agent tools. The first two were stand-alone implementations [FKH02, HKF+04, Müc08], not using any of the known tool platforms. The following requirements had to be met by the Toolipse approach:

- allow fast and efficient development of JIAC-based applications,
- narrow the gap between design and implementation,
- support beginners, advanced learners and experts at the same time,

- base on standards and best practices in software engineering and tools programming.

The outcome of the development provides visual and source code editors, extensive help components and project resource management, which are described in the following sections.

5.1.1 Overview

Toolipse is a fully functional prototype of an IDE based on the Eclipse platform, which facilitates the development of agent applications with the JIAC agent framework, increases their quality and shortens the development time. The aim was to hide the language syntax from the developers as much as possible, to allow them to develop an agent application visually and to assist them where possible. To achieve that, it provides the following main functionalities:

- creating and building projects, managing their resources and providing an internal resource model;
- creating JADL ontologies, manipulating them visually and importing ontologies from other ontology languages;
- developing agent knowledge in a visual environment;
- testing agent behaviours with agent unit tests;
- implementing agent beans in Java;
- configuring and deploying agent roles, agents and platforms visually;
- helping and guiding the developers through the entire development process with documentations, interactive how-to's and interactive tutorials.

Each functionality is realised as an Eclipse feature consisting of one or more plugins and typically comprises wizards, editors and views, which are arranged in an own perspective.

In Toolipse, wizards are used for creating projects and skeletal structures of JIAC files; each file type has its own wizard. After creating a file, the agent developers can edit the file with the associated editor, which is in the majority of cases a multi-page editor consisting of a source code editor and of a visual editor. The source code editors support syntax highlighting,

warning and error marking, folding, code formatting and code completion which suggests possible completions to incomplete language expressions. In contrast to the source code editors, which require from the developers good knowledge of the language, the visual editors of Toolipse allow to work with abstract models, to create and modify instances of the meta-model graphically. This facilitates the agent development and minimises errors. In order to achieve this, the visual editors model the JIAC concepts with the EMF, visualise them graphically with the Graphical Editing Framework (GEF) and provide simple graphical layouts such as radial layout, zooming and modifying properties of the visualised elements with the associated dialog windows as well as with the Properties view of Eclipse.

This Properties view belongs to one of the so-called workbench part concepts: views. They are typically used to navigate through resources or to assist the editors with extra functionalities. For example, all our editors support the Outline view where the outline of the file which is currently open is displayed. In addition to the Properties and Outline view, which are general views of Eclipse, Toolipse provides its own views that navigate the developers through the JIAC resources, present results of agent unit tests, to help or to guide them through the development process. Figure 5.1 shows the JIAC perspective with an editor and some of these views.

In the following subsections, we go into detail on the above-mentioned main features of Toolipse.

5.1.2 Resource and Project Management

The primary feature of an integrated development environment is a resource and project management. It supports organising project structures, navigating through resources, parsing files, caching and providing abstract resource model elements and building projects. For these purposes, Toolipse provides a resource manager, two incremental builders, a project wizard and a navigator view.

The resource manager maintains files, agent configurations and JADL language concepts in an internal model for all open JIAC projects. It is used by other components such as builders or editors. The internal model extends the Java model of the Eclipse Java Development Tools (JDT) with the JIAC meta-model. The model is always kept synchronous with the file system through a resource listener, which listens for changes on the file system and updates the model if required. Moreover, the manager uses caching methods to keep the size of the model reasonably small. For example, it creates instances of JIAC resources only once and sets the reference counter accordingly if the resource is referenced in many projects.

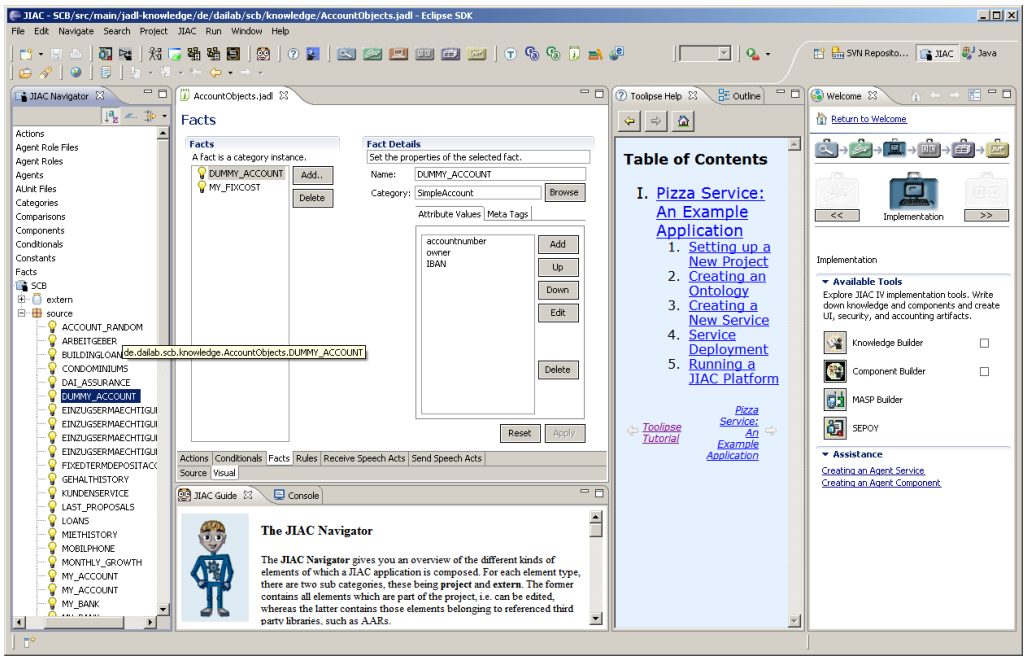

Figure 5.1: Toolipse with the following components (from left to right): JIAC navigator, knowledge editor (center), JIAC guide (bottom), interactive tutorial and user guide.

Two incremental builders are part of the resource and project management tools: the ontology builder which translates JADL ontologies into Java classes and the knowledge builder which converts JADL facts, reaction rules and plan elements into an executable form. Like the resource manager, the incremental builders listen for changes in the file system and trigger the build process if required. Here, "incremental" means that the builders compute all resources which depend on the changed resource and build only these relevant resources. In case of errors and warnings, the builders mark the affected resources with the corresponding annotations which are displayed in the Problems view of Eclipse, amongst others. An example build process looks as follows if an ontology file has changed or has been deleted:

1. The ontology builder calculates all ontology dependencies on the changed ontology and translates them along with the changed ontology into Java source files.

2. The JDT Java builder is activated afterwards because of new or modified Java files.

3. Finally, the knowledge builder computes the knowledge dependencies and builds the affected knowledge files, which completes the build process.

The projects created by Toolipse are also compatible to Maven, so that they can be build without the IDE. This feature is used, e.g. when different developers frequently integrate their work [Fow00].

In contrast to the resource manager and the incremental builders, which are invisible to the users, the project wizard and the navigator view have User Interfaces (UIs) and are used by the developers directly. The project wizard creates a project and registers the ontology builder, the Java builder and the knowledge builder on the project as project builders. Additionally, it adds so-called project natures to the project. One of them, the JIAC project nature, indicates that the resource manager should scan the project for resources, create the corresponding model elements and update the resource model.

The last tool is the navigator (Figure 5.2) which displays the resource model to support the developers with a resource view that shows only JIAC files and their contents. Moreover, the navigator also filters editor type specific resources. For example, it shows only ontologies and categories if an ontology has been opened by an ontology editor.

5.1.3 Domain Vocabulary

The development of an agent application typically starts with collecting the domain vocabulary, which is used to create the beliefs and the interaction vocabulary of the agents. In Toolipse, this development step is assisted by a wizard for creating JADL ontology files and a multi-page editor, which consists of a source code editor and a visual editor. While the source code editor supports syntax highlighting and code completion, the visual editor allows the developers to create and manipulate ontologies graphically by visualising ontologies in UML-like class diagrams (see Figure 5.3).

The visual ontology editor models an ontology as a UML class, which contains only methods (functions and comparisons), and does not contain any attribute. The categories of an ontology are represented as UML classes, which are connected with the parent ontology by UML compositions and contain only attributes. Inheritance relationships between categories are modelled as UML generalisations. In addition to the visualisation, the visual editor provides ontology editing functionalities which include importing other ontologies, creating and modifying categories graphically, implementing ontology functions and comparisons and editing attributes of a category.

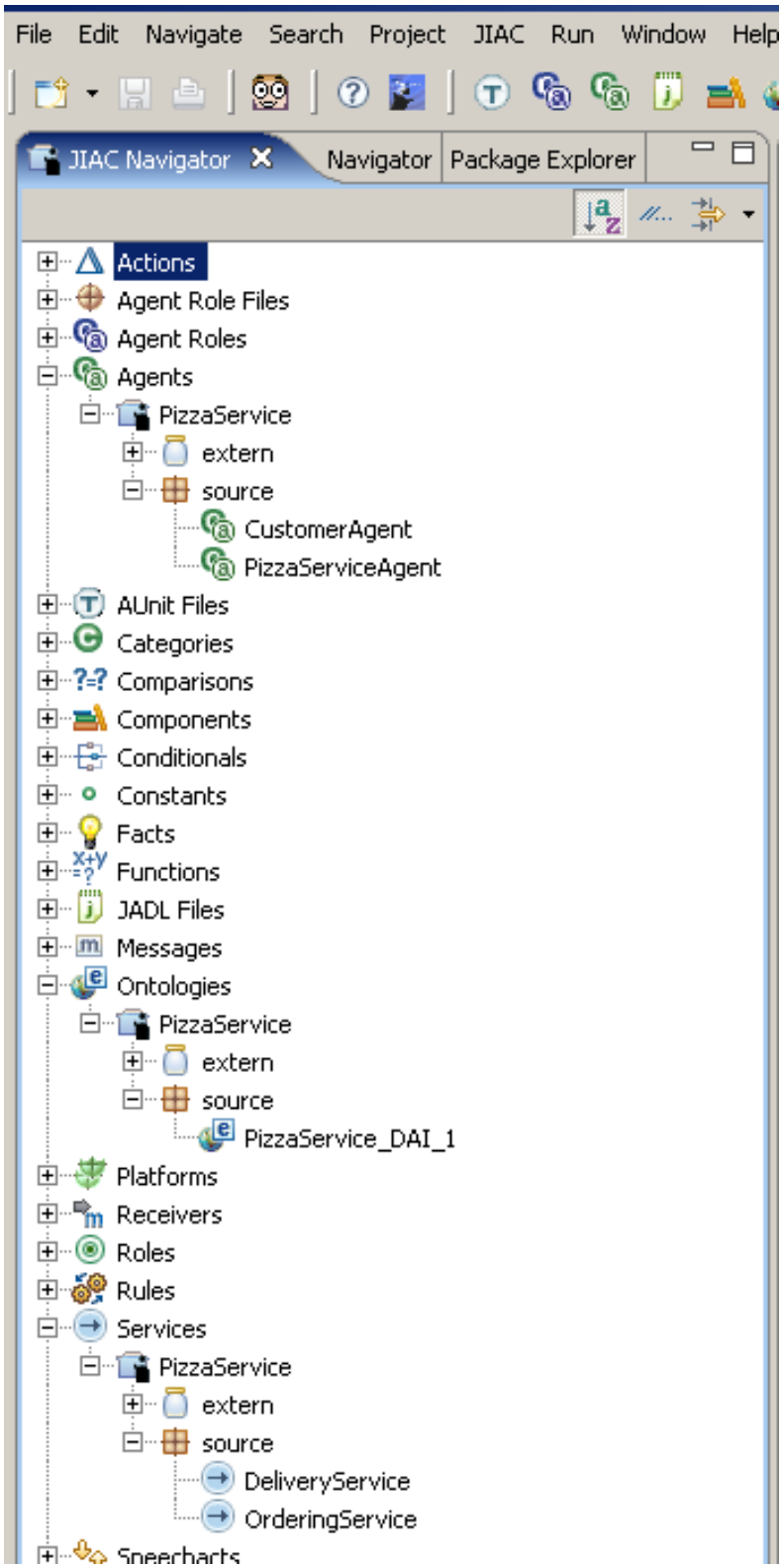

Figure 5.2: Navigator view showing project resources according to the JIAC meta-model.

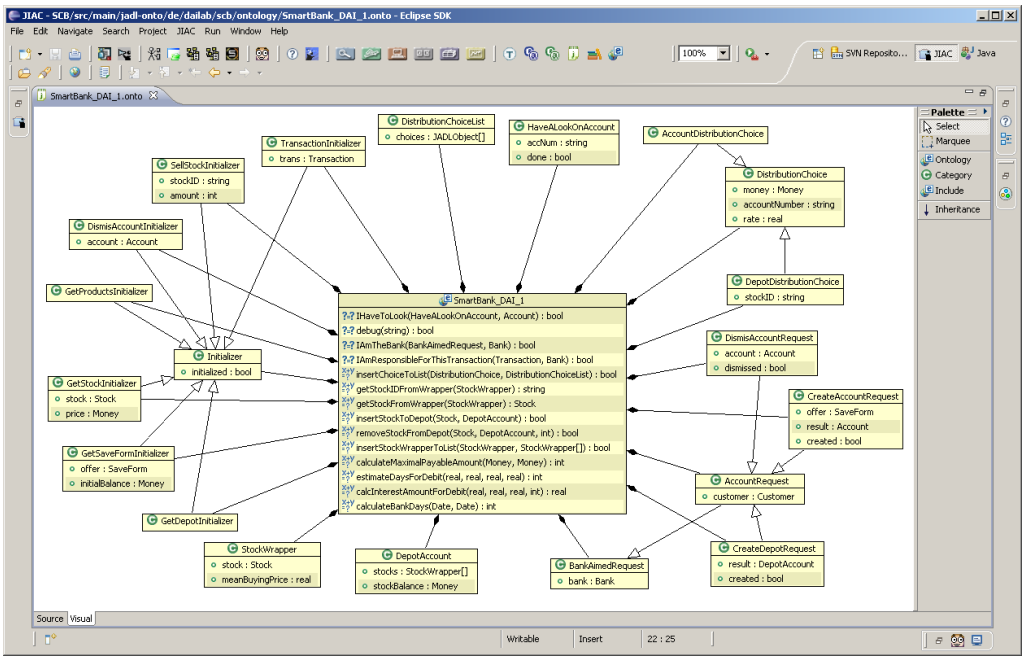

Figure 5.3: The visual ontology editor modelling the SmartBank ontology of the SCB scenario.

As opposed to other ontology languages such as OWL, it is not possible in JADL to define categories and their instances in the same file. Thus, the visual ontology editor does not support facts; however, facts can be modelled using the knowledge editor.

Furthermore, in order to support interoperability, Toolipse facilitates the development of ontologies with an import wizard, which currently translates OWL Lite ontologies into JADL.

5.1.4 Knowledge

Agent knowledge is described by JADL facts, reaction rules and plan elements. To support the agent developers in creation and maintenance of agent knowledge, Toolipse includes a knowledge editor, which comprises a source code editor and a visual editor. The visual editor is a multi-page editor, which contains a page for each knowledge type. Most of these pages are only form pages which create, edit and delete agent knowledge elements; however, the page for plan elements visualises JADL services, protocols and scripts as a flow chart and provides graphical editing functionality. Figure 5.1 shows the knowledge editor together with the fact form page,

editing two facts of the Service Centric Bank (SCB) scenario.

Moreover, the IDE provides two wizards for the creation of agent knowledge files: the JADL file wizard and the service wizard. While the former is used to create arbitrary JADL knowledge files, the latter is more specific, in that it creates a service description file as well as user and provider script files for the service protocol implementation.

5.1.5 Testing

In order to enhance the quality of an application and to detect errors early and continuously, it is essential to test [Bec02]. For this purpose, JIAC provides an agent unit testing framework called AUnit, which can test plan elements. In Toolipse, an AUnit test can be created with the AUnit wizard and processed with any Extensible Markup Language (XML) editor. Although there are some XML editors available as Eclipse plugins, we have added a simple XML editor to our IDE, which supports syntax highlighting and launching of AUnit tests. After launching and running through AUnit tests, the results are shown in the AUnit results view.

5.1.6 Agent Beans

Agent beans are usually used to implement agent core components or to wrap non-agent environment using Java APIs. They are exchangeable at runtime. The artifacts to create and modify here are agent beans, bean roles and bean messages. The creation of these artifacts is supported in Toolipse by a number of wizards, while the implementation support of agent beans is left to JDT. For testing agent beans we rely on the JUnit framework, which is also supported by JDT.

5.1.7 Deployment

In the last step of the agent development, the developers configure agent roles and agents, deploy them into agent platforms and launch the platforms. This development step is supported by wizards for creation of configuration files and by multi-page editors, each of them consisting of an XML editor and a visual editor. While the visual editor for agents and agent roles visualises hierarchical relationships between agents and agent roles, the visual platform editor graphically represents a platform into which agents can be inserted. Both visual editors provide a set of form pages with which properties of agent roles, agents and platforms can be manipulated. Toolipse

includes also a platform launcher, which can start a platform directly from the platform editor.

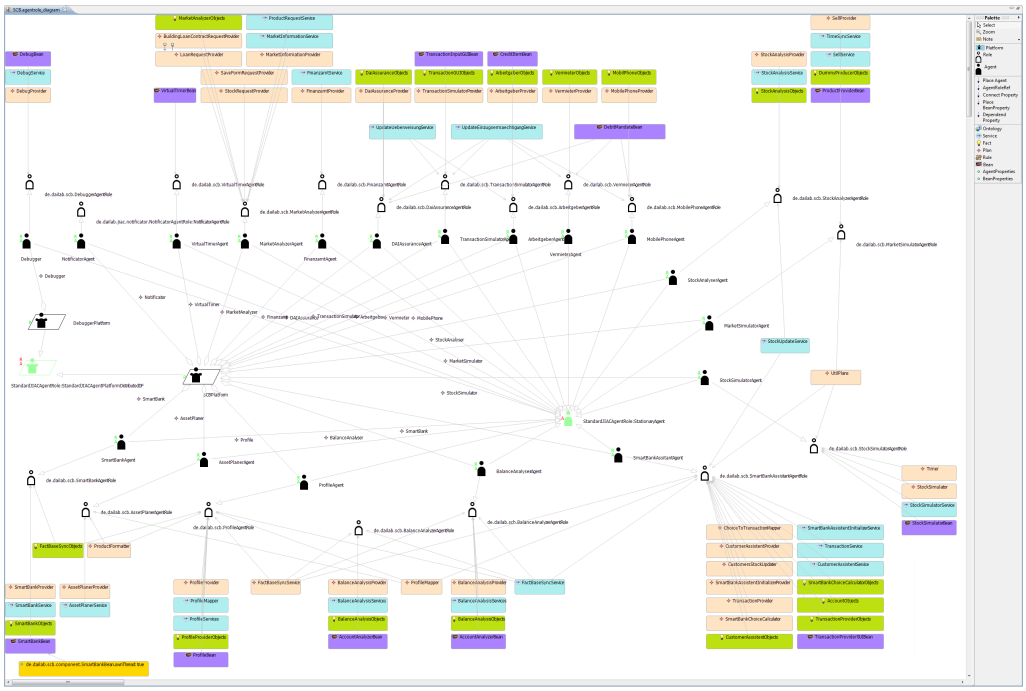

Figure 5.4: The agent role editor shows agent roles, agents and agent platforms together with their relationships of the Service Centric Banking scenario.

We have then prototyped a new agent role editor, which shows the overall structure of an agent society and allows different views on it. Figure 5.4 shows the new editor visualising agent roles, agents, platforms, their relationships and assembly from our SCB case study.

5.1.8 Helpers and Guidance

To familiarise the developers with our agent framework and its IDE, we have implemented a number of Toolipse help components, some of which extend the help system of Eclipse, while others are realised as separate components.

First of all, the IDE extends the welcome site of Eclipse with its own customised welcome site, which overviews the development steps, namely analysis, design, implementation, testing and deployment of an agent application. With each development step, a set of Toolipse and standalone

tools is associated; the developers can start these tools directly from the help site.

Beside the customised welcome site, it adds so-called cheat sheets into the help system of Eclipse. These cheat sheets are interactive how-to's, which demonstrate how a JIAC concept is created with a wizard and then is manipulated with an editor.

The next help component is the user guide, which guides the developers through the development steps similarly to the Toolipse welcome site. While this welcome site is designed to be displayed in full screen mode after the installation and gives an initial overview of Toolipse, the user guide, which is realised as a view and is displayed next to the editor area, is intended for guiding the users through the entire development process. Other than the user guide, the JIAC guide gives the users only a short description of the currently selected tool, providing constant supporting information.

Furthermore, the IDE provides interactive tutorials such as a pizza delivery service, with which the developers can create a full JIAC project interactively.

5.1.9 Comparison

Although we concentrate on the needs of JIAC users in the first place, we have also evaluated a number of other agent development tools, in particular tools that help creating real-life applications. We have found the JDE [Age05] very inspiring concerning the design of an agent-based application. Based on a clear visual notation, the Design Tool and the Plan Editing Tool allow modelling an application from different views and support code generation. Together with the plan tracer and agent interaction display for runtime monitoring, the JDE is a complete toolbox for easy and fast agent-based development. We are also looking forward to test the CaFnE tool [JTPW06], which allows domain experts to develop or modify agent applications. The description and demonstrator promise a good step forward. A good standard toolkit for Jason applications has been delivered with the Jason IDE [BHW07], which provides project management and AgentSpeak source code editing together with a number of wizards and debugging capabilities. The Cougaar IDE [Cou08] has also been realised as a number of Eclipse plugins and provides the management and running of Cougaar projects. Whitestein created a Development Suite [RCK05] for their Living Systems (LS) Platform. It consists of two parts: a number of Eclipse plugins for creating, re-use, debugging and monitoring of LS applications and a Modeler, realised as plugin for a UML tool, which allows the modelling of agents and their behaviour using AML [Whi04b].

5.1.10 Lessons Learned

Toolipse has been realised, a fully featured IDE prototype for the fast and efficient development of JIAC agent applications, based on the Eclipse platform. Eclipse has been chosen as integration platform for JIAC agent tools in compliance with best tool builder practice. The IDE supports both experts and beginners at the same time. While it provides experts with standard text editor functionalities such as syntax highlighting, warning and error marking, and code completion, beginners can create an agent application solely by using visual editor functionalities, which include visualisation, zooming, graphic layouts, creating, editing and deleting JIAC artifacts visually. Additionally, beginners and advanced learners are assisted in Toolipse with IDE and framework documentations, guides, how-to's and interactive tutorials.

The JIAC Toolipse IDE has been used and tested in teaching and by a number of projects in different domains. The feedback from the early testers was mainly positive. The main deficiency, which the testers pointed out, was lack of a code refactoring capability. They also missed a possibility to edit higher level interaction protocols, which is one of the most challenging topics. Additionally, some testers wanted a feature which supports modelling agent services with standard process modelling notations such as BPMN [OMG08] using predefined basic services.

The next release has been planned, which came with revised and enhanced text editor functionalities. The agent role editor has also been reworked (see Section 5.2.1), which now allows to visualise all artifacts of a JIAC-based application in one diagram as well as filtering diagram information to view different aspects of the application. The next tool we were then working on was a visual service design tool [Küs07] (see Section 5.2.3), which can model services with BPMN and transform them into executable service languages. It also became a part of Toolipse, and supports a transformer from BPMN to JADL and thus can be used as agent service modelling tool.

Toolipse can be downloaded at `http://www.jiac.de` and the different features can be tried directly. While we think that we have provided a fully featured and powerful toolsuite, there is always work to be done and we hope that not only will you give it a try but also let us know any further improvements that we could make.

5.2 Advanced Tooling

5.2.1 Agent World Editor

Over the last decade, the AOSE has gained attention as a suitable AOSE methodology for providing quality assurance within software development processes [BGZ04]. AOSE controls and manages software life cycles while supporting the developer with a high degree of flexibility and expressiveness. It has been shown to be a successful method for developing complex software systems.

However, there are at least as many agent methodologies as there are agent frameworks, and each has their own drawbacks and advantages. At present, the DAI-Labor Berlin has three derivatives of its agent framework JIAC in use, each with varying degrees of tool and methodological support. This condition is far from perfect, since tool support is recognised as an important element of AOSE in general, and plays a vital role in the success of the different systems.

While the different JIAC versions are different in their goals and implementation, they have certain characteristics in common. For example, agents always consist of different components that together constitute the agent. The components can be Java beans, plan elements, goals or rules, depending on the target framework and goal of the agent. The different components are then *configured* to form an agent, and the agents together constitute the multi-agent system or the agent-based application.

This section describes the Agent World Editor (AWE), that allows to design MAS and translate the results into different agent frameworks, namely JIAC IV [SA02], JIAC V (see Section 3.2), and *MicroJIAC* [PT09]. This section is based on the diploma thesis of the first author [Lüt09].

The MAS Design in JIAC is done with the AWE [LKHH09]. AWE allows for the visual creation and editing of multi-agent systems via drag and drop and supports concepts such as agent-nodes, -types and -roles or components. The applied notation is simple but comprehensive and represents the entire MAS in one single diagram as directed graph (see Fig. 5.5). Access to the file system allows to reference existing Agent Beans or JADL services just like the ones created with the VSDT.

The Agent World Editor allows for the generation of executable code from the visual MAS design. The code can be understood directly by the JIAC runtime and triggers the execution of a MAS according to the design specification.

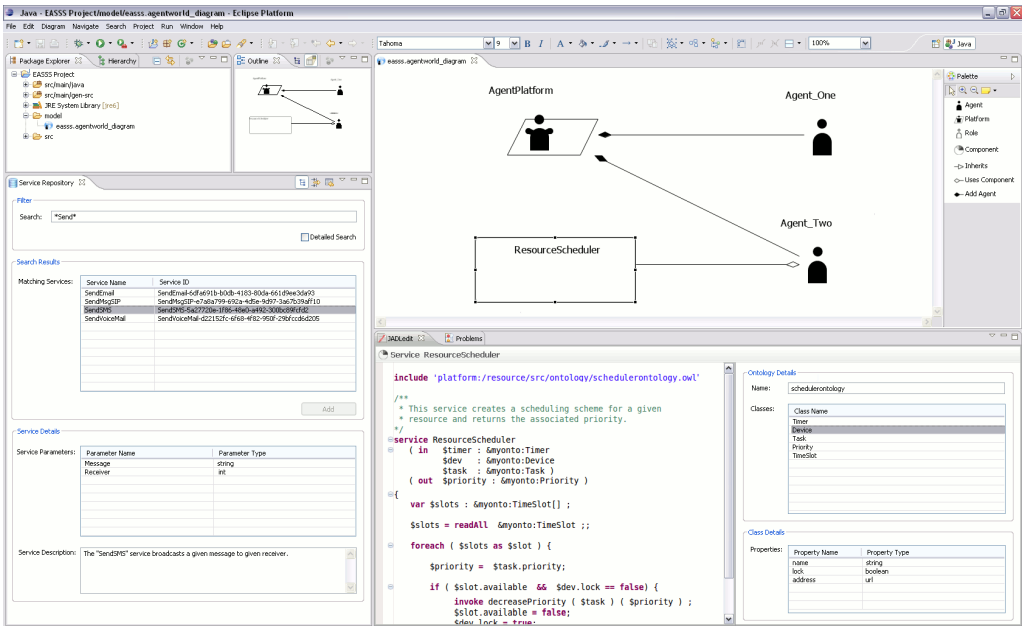

Figure 5.5: Agent World Editor and JADLedit as a part of JIAC's tool suite.

5.2.1.1 AWE Overview

The AWE has been developed using the GMF. In AWE, complex multi-agent systems can be conveniently represented in a single diagram, showing the various agent types and their relationships. An example is provided later in Figure 5.9.

Formal background to these diagrams is a highly generic domain model, which covers the entire JIAC scope and is thus capable of carrying valid configuration instances of each JIAC derivative. The MAS design process is mainly accomplished by selecting elements from a palette. Detailed settings are provided by property sheets, which are available for each diagram element type. The property sheets are customised to provide better usability than the generated ones, while still conforming to the general look-and-feel of Eclipse. After the MAS has been configured, executable code for different frameworks can be generated with a single click. Using a modular approach, the transformations for each framework to be supported, being capable of translating the current instance of the generic domain model to the syntax of the selected target framework, can be implemented with little effort, as has been done already for the three derivatives of the JIAC family. The principles of this translation process are illustrated in Figure 5.6.

AWE is entirely based on a plug-in architecture, where each plug-in

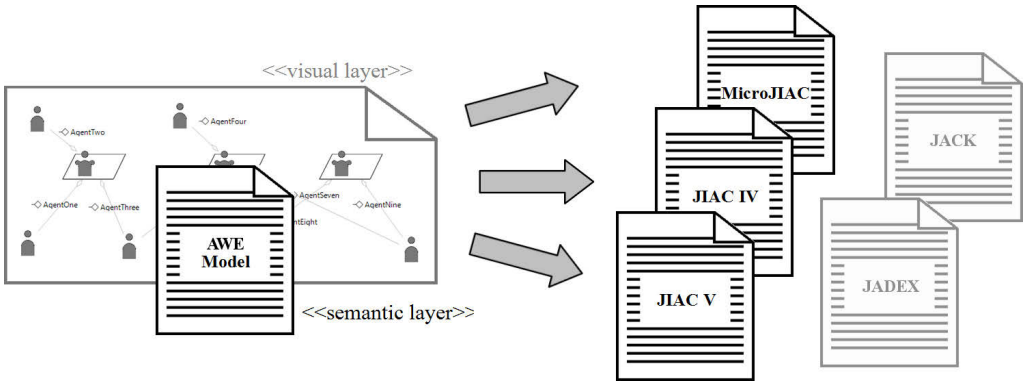

Figure 5.6: The Editor's Semantic Foundation and its Associations to the Framework Dependent Configurations

realises a distinct part of functionality. This approach allows to easily exchange even system relevant components, such as the domain model or the entire visual representation and thus ensures comfortable maintenance as well as easy enhancement. Each framework is encapsulated by a single plug-in. The modules are loosely coupled to the base application and thus provide optional and customisable framework support. The standardised structure of each extension plug-in furthermore encourages the development of tailored solutions and mainly consists of the definition of a customised translation routine, which produces the desired executable configuration code from AWE's domain model scheme. A detailed examination of this model is given below.

5.2.1.2 AWE Domain Model

The Domain model was designed to at least encompass the configuration metamodels of the three JIAC derivatives. In order to do so, it has to provide unified solutions for each framework's agent types, as well as components and library support. While the diverging agent types, e.g. agents, roles, nodes and platforms, can easily be mapped on a uniform set of types within AWE's domain model, namely agents, roles and nodes, the remaining challenges require more elaborate mechanisms, which we describe below.

Components JIAC's basic operation principle is to support the developer with a set of default agents which are capable of agent-elementary functionality (like communication or environment awareness) and allow him to define specific behaviour by appending closed and reusable components. Although the types and the architecture of this be-

haviour attachment differ through the JIAC family, the general concept remains component oriented. Utilising this condition, the domain model's *Component* class serves as abstraction in this matter and provides a final type specification by attribute.

Library Support In order to provide support for library elements of different agent frameworks, we extend the AWE's domain model with the *Concept* class, which is used to abstractly describe a library element's functionality by means of a conceptualisation rather than pointing to its concrete implementation. These abstract concepts are specified within the AWE base application, and used by each installed framework extension, which individually defines a mapping from the conceptual element to a corresponding implementation and thus ensures a framework associated execution. The mechanism is illustrated in Figure 5.7. The selection of concepts is geared towards the requirements of JIAC, and they define a set of default types, which can be referenced by custom developments, ensuring an accelerated and safeguarded development process. We support the MAS developer with concepts like standard agency, which implies communication facilities and service awareness or more advanced concepts like mobility, which extends the previously mentioned characteristics with the ability to migrate from platform to platform. The modular implementation of the concept mechanism comfortably allows to extend this as needed.

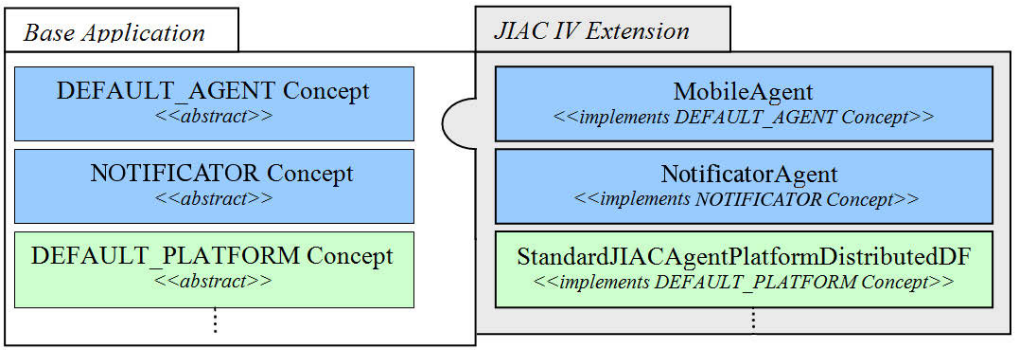

Figure 5.7: Concepts and their respective Implementations

5.2.1.3 Development Features

We provide a number of usability features with the editor. For example, AWE supports the developer not only with comfortable modelling accelera-

tors, but with error avoidance and representation enhancement mechanisms as well. A brief explanation of the most prominent features is given below.

Modelling Support Since we employed directed graphs as MAS representation, most of the modelling part is determined by connecting nodes with edges. This can lead to a cumbersome process, particularly regarding large multi-agent systems. We designed AWE to increase its modelling efficiency by simplifying recurring tasks like connecting a set of components to an agent. Instead of connecting these elements one by one, we provide elaborate mechanisms which abbreviate these tasks to just a few clicks.

Error Avoidance AWE supports error detection on a structural as well as on an attribute level. On-the-fly validation accompanies the actual element creation and thus prevents inconsistent states like inheritance cycles. Attribute values are evaluated against the framework specific constraints and, if necessary, marked as faulty. AWE's modular architecture allows an easy extension to individual constraints.

Representation Enhancement Based on the evaluation of related tools, we developed a simple but expressive visual notation for the occurring entities. We included as much expressiveness as possible into each employed figure, in order to reduce the complexity of the overall diagram. In this course, we reflect certain standard inheritance associations merely within the figures and provide an optional *unfold* feature. Since AWE shows a complete representation of the MAS, we provided an *accentuation* feature to enhance readability. This feature is capable of highlighting individual agents and their networks and thus provides insight into subordinate MAS fragments.

Furthermore, AWE inherits many useful features from GMF, such as unlimited undo and redo, auto-arrange, snap-to-grid, modeling assistance, a graphical outline, picture export and many more.

5.2.1.4 Framework Extensions

In the Agent World Editor, support for the actual agent frameworks is realised using Eclipse's plug-in mechanism, so that tool support for additional agent frameworks can be added to AWE with minimal effort. First and foremost this includes code generation, but also the provisioning of framework specific components. To this end, AWE includes an *Extensibility Plug-In*,

acting as the counterpart to a number of *Extension Plug-Ins* for the various agent frameworks.

The Extensibility Plug-In acts as connection between the framework extension plug-ins and the base application. At startup, each installed extension plug-in registers itself at the Extensibility Plug-In, providing an individual set of library elements, i.e. concrete implementations available for that agent framework for the various abstract concept elements (e.g. communication, mobility). The library elements of all installed framework extensions are categorised and collected in the property sheets, from where they can be used for creating a new agent configuration.

Implementing Framework Extension Plug-ins Plug-ins extending the Agent World Editor with framework specific library element, modeling support and code generation can easily be created. To this end, the plug-in's *Bundle Activator* class has to implement a certain interface, making its functionality available for the base application. Firstly, each framework extension has to define agent roles implementing the several concepts, including a role to be extended as the default. Upon code generation, these roles are used as a substitute for the abstract concepts. Optionally a set of custom figures to be used for those elements can be supplied, too. Secondly, code generation for the respective agent framework has to be provided, being available via an entry in the diagram's context menu.

The Code Generation The major component of each framework extension plugin is the transformation of the agent world model created in the editor to the respective framework specific model, e.g. some declarative XML file. In the course of this work, EMF has been used for describing these models, but any other technique can be used as well. The transformation procedure is started on selecting the respective export functionality from the context menu of an individual agent or the diagram as a whole.

After the diagram has been tested for cosistency (e.g. no cyclic inheritance), the export procedure itself is very straight-forward. As each connection in the diagram has an unambiguous "owner", the whole diagram can be covered without redundancies by iterating over the several agents, roles and nodes constituting the agent world diagram.[1] For each agent, the respective counterpart of the selected agent framework is created and configured according its relations to components and other agents; the re-

[1]Components, that are not connected to any agent, do not have to be regarded in the transformation.

sulting model can then be serialised in a format according to the targeted agent framework.

In the course of this work, framework extension plug-ins have been realised for the three frameworks of the JIAC family: JIAC IV, JIAC V and MicroJIAC. In the following we will exemplarily give a detailled description of the mapping to JIAC V using a simple example.

Transformation to JIAC V With the JIAC V configuration files being based on Spring, the process starts with the creation of a *beans* object, serving as container for the other agent instances. Subsequently four more *bean* objects are added to the container, representing the platform, the two agents and the component, and their attributes are set accordingly, whereas the *parent* attribute is set to this framework's default parents, i.e. *PlatformWithRegistry* and *SimpleAgent* respectively. Finally, the agents are added to the platform-bean's *agents* list and the component to the agent's *agentBeans* list. Finally, the configuration is serialized to XML, as depicted in Figure 5.8, and given that the component exists, it can be executed by the JIAC V runtime.

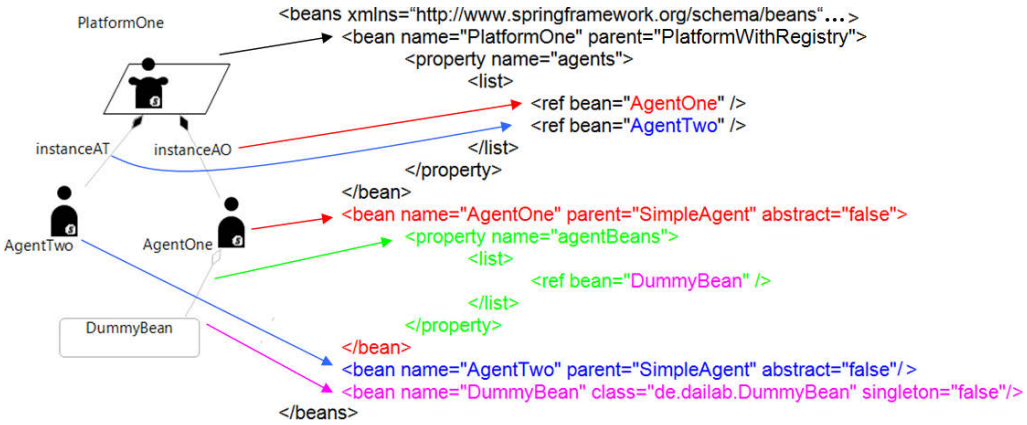

Figure 5.8: JIAC V Mapping Example

5.2.1.5 Example

Having introduced the Agent World Editor, we will use it for recreating the agent configuration of a recent project.

The SCB scenario has been developed in a doctoral thesis [Müc08]. Its objective was the demonstration of principles of the agent oriented approach

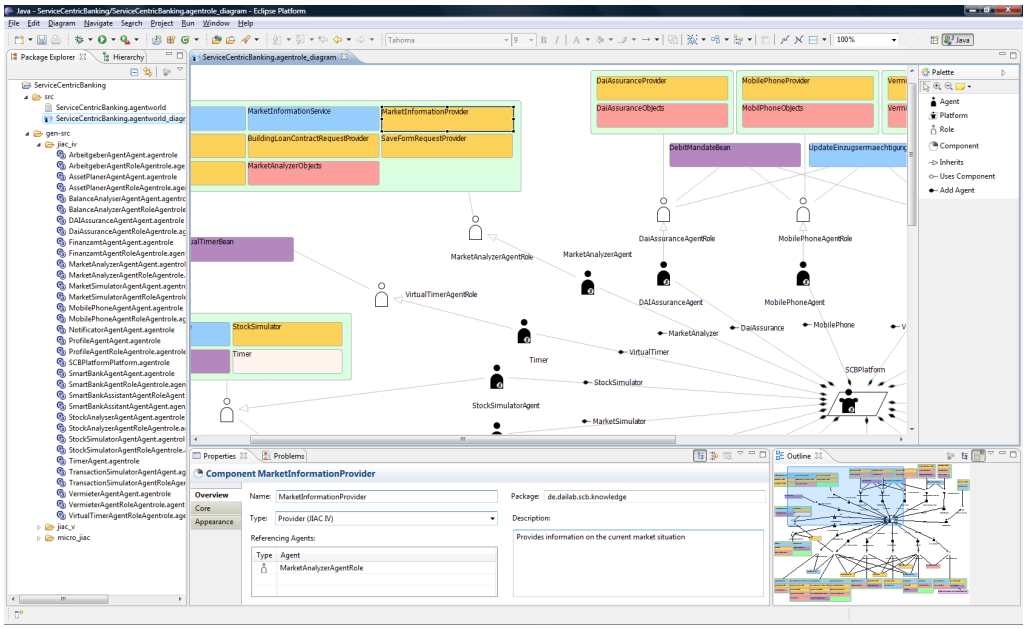

Figure 5.9: Agent World Editor showing the *Service Centric Banking* scenario

by representing a customer within an agent oriented banking context. Several agents are involved in order to simulate the investment- or the stock market and to represent the customer herself. Based on a personal profile reflecting her current living situation and investment preferences, a dynamic supervision of her account development and an analysis of the current market, the banking agents provide her with a customised investment plan.

Figure 5.9 shows the scenario being recreated with the Agent World Editor.

The design starts with the creation of agents, platforms, roles, components, and the connections between them. Given the MAS's complexity, the editor's structuring features prove to be notably useful.

Next, the elements' attributes have to be set using the enhanced property sheets. In particular, the types of the components have to be selected. In this case, the JIAC IV components to be used in the SCB scenario are chosen; among others the *Notificator* concept is used, which is mapped to JIAC IV's *NotificatorAgentRole*. This at the same time determines the diagram to be executed. as a JIAC IV agent system. Selecting the corresponding entry form the menu triggers the generation of the respective JIAC IV agent configuration files.

5.2.2 JADLedit

JADLedit [BLM09] is a source code editor for JADL++. The tool facilitates the development of JIAC services with features such as syntax highlighting, code completion, and error marking (Fig. 5.5). Regarding the methodology, JADLedit is used for the implementation of previously generated service stubs.

As stated above, JADL++ services are using OWL as representation for complex data types. Thus, JADLedit allows for type safe editing of knowledge (facts). For this purpose JADLedit includes an ontology browser view, which can be used for browsing existing OWL ontologies, providing a visual representation of their classes and properties.

5.2.3 Visual Service Design Tool

Our methodology starts with defining use cases and the respective BPMN processes. This stage is supported by the Visual Service Design Tool (VSDT) [KH08], a BPMN editor, providing process analysis and validation tools, such as a process structure view, a process simulator, and the generation of natural language process documentation (see Fig. 5.10). The VSDT is also used for creating the use case diagrams, establishing the relation of participants to processes and connecting the individual business process diagrams to a complete system.

The VSDT includes an extensible transformation framework, which is responsible for grouping the nodes of the process graphs to blocks of structured programming languages (i.e. sequences, selections, and repetitions). Currently, the process diagrams can be exported to both BPEL and to JADL++, and more transformations can easily be added. Further, the VSDT uses a very simple expression language, which can be validated and interpreted at design time, and translated to the expression languages used in the executable languages, such as JADL or BPEL's XPath. The use case diagrams are translated to the agent role model and the relation of the roles to the generated services. These models then serve as input for the Agent World Editor.

5.3 Streamlined Tooling

The design of large, distributed applications involving several communicating partners can be extremely challenging. Besides the prevailing SOA solutions, agent-oriented systems are regarded as a promising approach in this domain [LMSW05]. Today, a variety of multi-agent systems provide

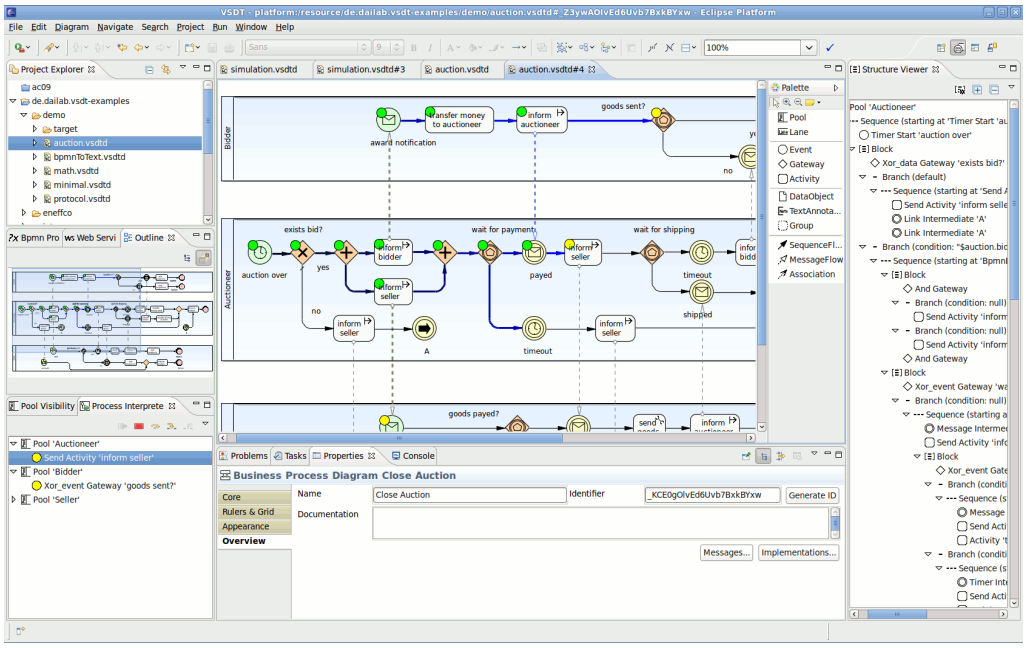

Figure 5.10: The Visual Service Design Tool

means for facilitating the implementation of such systems, such as libraries and special languages for discovery, messaging, and event handling, and some provide advanced concepts like pro-activeness and autonomy. However, some general problems of distributed systems remain, such as the alignment of sending and receiving messages in a complex communication protocol.

The developer sees just a part of the problem: One file of source code, one agent's set of rules, one side of the communication. What is needed is a holistic, integrated view on the system, showing the involved partners and their processes as they are orchestrated in the system. Additionally, everything has to be well-documented, especially in cross-enterprise projects, and technological progress is demanding shorter and shorter development cycles.

One possible solution to this problem is the application of techniques from business process design to agent engineering. Over the last years, a number of industrial projects have investigated business process design as a way of closing the gap between analysis and development [GH05]. Many ideas from agent theory, such as roles, rules, and communication, can be found in the BPMN, too, which has already led to some efforts of mapping BPMN diagrams to multi-agent systems, as we will see later. We will have

a closer look at BPMN in Section 5.3.1.

Still, there are other aspects, for which BPMN is not intended, such as modelling organisation, complex algorithms, or data. While BPMN describes business partners, i.e. roles, it does not define how these are implemented by agents, let alone their physical distribution. And while process diagrams are in general very well suited for displaying all kinds of control flow, modelling each single algorithm in BPMN would not be adequate. Finally, while BPMN can refer to data types, e.g. for properties and assignments, it does not provide means for defining them.

In order to apply business process design in agent engineering, it needs to be combined with other techniques and additional notations. Therefore, in Section 5.3.2, we introduce a methodology, in which BPMN is used together with ontology engineering, multi-agent system design and service engineering, combining the strengths and balancing the weaknesses of the individual approaches. While the methodology is applicable for other agent frameworks, too, we will illustrate it using the JIAC V framework, providing a number of useful tools and editors (see Section 3.2). In Section 5.3.3 we will describe the mapping from process models to JIAC agent systems. Thereafter, in Section 5.3.4, we will apply the methodology to a simple online auction example, and discuss some open issues in Section 5.3.5.

5.3.1 BPMN

The Business Process Modeling Notation [OMG09] can be seen as a combination of UML's Activity Diagrams and Sequence Diagrams. It can be understood at three levels of abstraction:

1. The diagrams are made up of a few easily recognisable node types, i.e. Events, Activities and Gateways, connected by control- and message flow.

2. These basic elements are further distinguished using sets of marker icons, e.g. Message, Timer, and Error Events, or parallel and exclusive Gateways.

3. Each element contains a number of additional attributes, which are hidden from the diagram, but contain all the information that is necessary for automated code generation.

Thus, BPMN is easily understandable by all business partners, even those who have great knowledge in their domain but do not know too much about programming and multi-agent systems. At the same time, BPMN diagrams

provide enough information for generating executable program code from them.

One problem with BPMN which is often brought up is that most of the semantics are derived from the mapping from BPMN to the BPEL, so for those elements that are not covered by this mapping the semantics are not always made clear, let alone formally defined. Still there is an increasing number of approaches describing the semantics of BPMN using e.g. Petri nets [DDO08, EHKA07, RPU+07], and version 2.0 of the specification makes things clearer, too. But why not use Petri nets in the first place? The answer is simple: While Petri nets have very clear semantics, and basically everything can be expressed as a Petri net, some high-level constructs, that are directly supported by BPMN, would result in huge, incomprehensible Petri nets.

BPMN diagrams, on the other hand, have a variety of notational elements, making them well suited for the design of distributed systems in general and multi-agent systems in particular. The process diagrams are subdivided in pools, each representing one participant in the process, which maps to the concept of agent roles. Using message flows for communication between pools, even complex interaction protocols can be modelled clearly. The "ad-hoc" process even resembles goal-oriented behaviour. Further, the notation supports features such as event- and error handling, compensation and transactions.

But as already mentioned, there are other aspects of a system that can not be designed very well using BPMN alone. These issues and how to deal with them are discussed in the next section.

5.3.2 Methodology for Process Oriented Agent Engineering

The core aspect of our approach is the application of business process design in an early stage of the development process in order to provide a holistic view of the entire distributed multi-agent system. As we have pointed out, BPMN alone is not sufficient for this task, but has to be embedded in a methodology that deals with the missing aspects, including (a) the declaration of data types, (b) design of agent organisation and distribution, and (c) textual programming for low-level algorithms. Further, as one BPMN diagram shows only one business process, while a complex system might consist of a number of processes, we also include (d) use case diagrams, providing an overview of the processes belonging to the system and their relation to the several participants. Finally, no methodology can be complete without

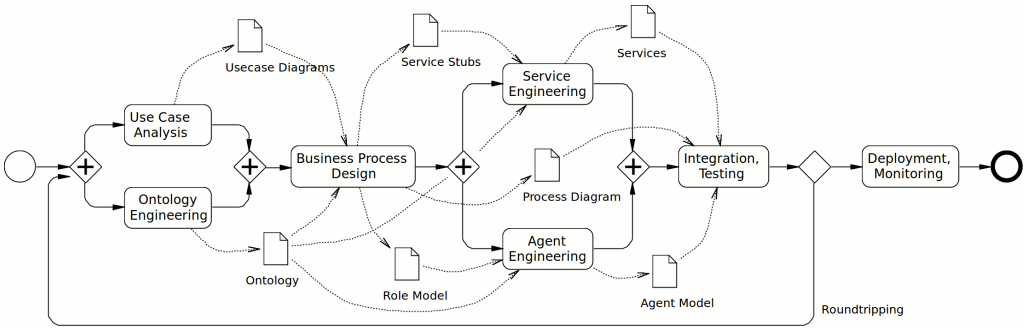

Figure 5.11: Process diagram showing the stages emphasised in this work within the complete methodology.

iterative development, testing and maintenance.

The "big picture" is shown in Fig. 5.11: The development process starts with use case analysis and ontology engineering. Then, for each use case a business process diagram is created. From the processes, the role model and a number of behaviours/capabilities are derived. Both are then further refined using tools more suitable for this task than BPMN. Finally, the agent organisation model and the components are integrated and tested, and the process can go into another iteration, until the system is eventually deployed. In the following, we will describe the relevant steps in more detail.

5.3.2.1 Use Case Analysis and Ontologies

The methodology starts with the identification of actors and use cases of the system under design. For this task we adopted the use case notation from UML. Here, each actor represents an agent role, with each use case being a business process in which the actor takes part. Apart from providing an entry point into the design, this also serves as a connection between the several business process diagrams, since one BPMN diagram alone does not suffice for describing a complete system.[2]

Another task that can and should be accomplished in this early phase is the (formal) definition of the domain vocabulary or ontology. However, in the context of this methodology it is not relevant whether this is done using UML class diagrams, Java classes, or sophisticated ontology languages, such as OWL.

[2]Version 2.0 of BPMN includes Conversation Diagrams, serving a similar purpose. We still prefer use case diagrams, as these are more intuitive to understand.

5.3.2.2 Business Process Engineering

Next, a number of BPMN diagrams are created based on the use case diagrams. Each use case corresponds to one business process, and each actor involved in the use case matches one pool in that process diagram. Here, the process diagrams are not used for modelling the entire life cycle of that actor. Instead, they depict only the workflow and the communication between the actors for this specific use case. In this regard they can be seen as a combination of activity diagrams and protocol diagrams. The process diagrams will serve as starting point for the implementation.

As a rule of thumb, the process diagrams should contain at least each activity and each branch related to the communication with the other actors, referred to as the 'public process' in the BPMN specification [OMG09]. The payload of the messages should also be specified using the formerly defined domain model. Of course, the diagrams can be further detailed to contain more of the actor's internal activities (the 'private process'), since they can also be used as documentation and for validating the final implementations against them. But as the diagrams' complexity quickly increases as one tries to depict each detail of an algorithm, the modeller has to find a practical compromise here. For example, while possible, it does not seem reasonable to detail a search algorithm like A^* in BPMN; instead a single, adequately named activity should be used.

5.3.2.3 From Business Process to Multi-Agent System

The business process diagrams provide a view on the system as a whole that can be used as groundwork for the following implementation of (a) the agents' behaviour, e.g. in the form of plans, rules, or services, and (b) the organisational model, e.g. roles, agents, and agent platforms.

Of course, the derivation of agent behaviours from the process diagrams is highly dependent on how behaviours are implemented for a given agent framework, e.g. imperative or declarative, as a set of rules, goal-oriented, or using hybrid approaches, and whether it is done automatically or manually. Generally speaking, for each pool in the different process diagrams, or more precisely, for each start event in the pools, one behaviour is derived, covering the workflow (branching, communication, etc.) from that starting point on. For instance, a pool with a message start event can be translated to a script for the content of the workflow and a rule for starting that script when the given message arrives.

The basic procedure of the mapping has been developed in our previous work. First, a BPMN *normal form* for facilitating the mapping has been

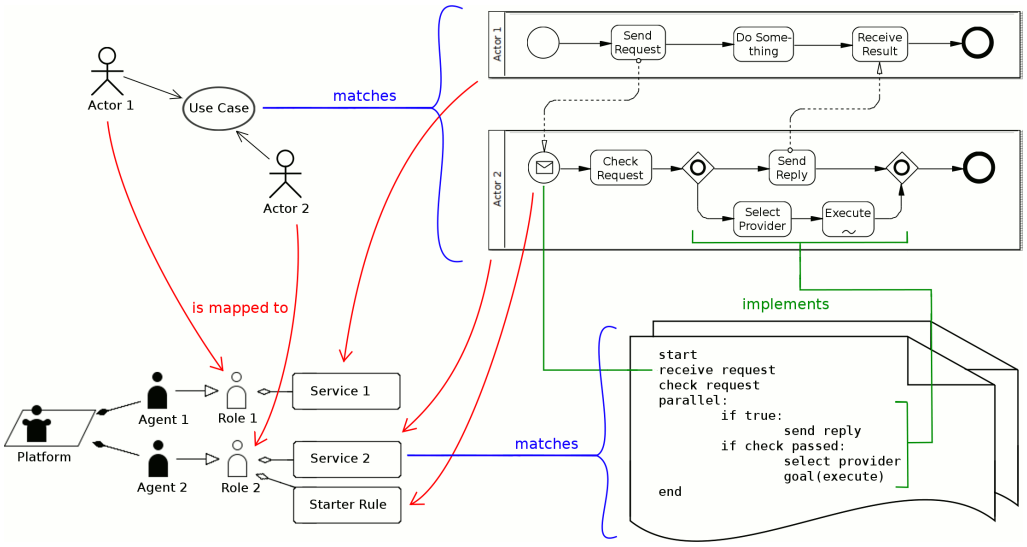

Figure 5.12: Overview of models and their dependencies. Use Cases, Process Model, Role Model, Implementation

investigated and formalised using Petri nets [EHKA07]. Then, the first steps of the actual mapping were specified, basically mapping pools to agents, processes and flow objects to the agents' plans and the control flow, and message flow to the exchange of messages between the agents [EKHA07]. Since then, the mapping has been adapted in order to provide better support for systems consisting of multiple BPMN diagrams, by taking the use case diagrams into account, too. The current state of the mapping, targeting the JIAC framework, is given in Section 5.3.3.

The use cases and actors are used for identifying the agent roles. For each actor one agent role is defined, offering the respective behaviours derived from the processes corresponding to that use case. Since the business process diagrams make no assertions towards the grouping of roles to individual agents and agent systems, this aspect has to be dealt with in the next step.

Fig. 5.12 shows an overview of the models and their dependencies. Both parts of the transformation can be done manually or automatically. We will introduce a BPMN editor that can be used for automatically generating services and a role model for the JIAC framework in Section 5.2.3.

5.3.2.4 Refinement

By now we have agent roles and their relation to a number of behaviours, derived from the use cases, and the behaviours implementing the workflow and the communication as designed in the BPMN diagrams.

As mentioned, the use of BPMN is not efficient when going into too much detail. Thus, in this step the behaviours are refined with everything that is too cumbersome to model in BPMN, e.g. complex calculations, or GUI calls. Further, the role model has to be completed with actual agents and agent platforms.

5.3.2.5 Integration, Testing, Deployment

After the different parts have been integrated, the system needs to be tested. Here, the process design can be of some help, too, as processes can be simulated, similar to a Petri net, and the behaviour of the final system can be compared to that of the process. Most notably, the use of model driven engineering ensures the consistency of the resulting multi-agent system with the formerly defined models. Depending on the applied tools, the system (or at least parts of it) can be generated from the models, and the models can be updated according to the implementation. However, the latter is still a hot topic of current research.

5.3.3 Mapping Process Models to JIAC V

The core aspect of the methodology — the transition from process models to agent systems — has been implemented for the JIAC framework as a diagram export plugin for the VSDT. The mapping from BPMN to JIAC V currently covers three aspects of multi-agent systems:

- Participants in the Business Process are mapped on *agent roles*,
- the Participants' individual Processes are mapped on *services* provided by these roles, and
- the Events triggering the Processes are mapped on *reaction rules*, invoking the services.

In the following sections we will give a detailed, yet informal description of these three aspects of the mapping, as due to the complexity of the BPMN standard a formal description of the mapping would go beyond the scope of this paper. Starting with the individual services at the bottom, we

will work our way up to the rules, triggering the services, and finally the agent roles integrating them.

5.3.3.1 From Process Diagrams to JADL Services

The creation of JADL services is the core aspect of the mapping from BPMN to JIAC. In principle, the entire logic can be specified in BPMN and translated to JADL. However, in praxis it is more reasonable to restrict the process models to the basic control flow and – most importantly – the communication between the several agents (the 'public process'), as this is where BPMN helps the most. When going into too many details, on the other hand, the use of BPMN can be very laborious.

Naturally, the mapping to JADL services is in many aspects similar to the mapping from BPMN to BPEL, as given in the specification. However, there are some interesting deviations, as well.

Pools and Services Usually, for each Pool in the business process diagram one JADL service is created. But this is not always the case, as we will see later when considering multiple start events. Further, a Pool can be subdivided into several Lanes. However, we decided against considering Lanes in the mapping, as their semantics are widely arbitrary; still, this aspect may be interesting for a future extension of the mapping. Each Pool in BPMN can have Properties (variables), which are mapped to the service's variables, while the workflow within the Pool is mapped to the service logic.

Structure Mapping / Control Flow Since the workflows in the Pools are directed graphs (which may also be cyclic), whereas JADL is a structured programming language, the first step in mapping the process diagrams to JADL services is the identification of structures in the process's workflow graph. For this purpose, a number of pattern matching rules have been defined [KH08]. This aspect will not be discussed in detail here, as these rules are largely the same as e.g. in the case of mapping BPMN to BPEL, for which sufficient literature exists (see for example Mendling et al. [MLZ05]).

The most basic rules are such for identifying (a) sequences, (b) decisions and parallel blocks (with n branches), (c) loops, with one or two branches, and (d) event handlers, where the execution of an activity is interrupted on the occurrence of a certain event. Further, there are a number of 'convenience' rules, which allow for fewer restrictions when modelling process graphs, such as interlaced blocks, and multiple start- and end-activities.

By applying these transformation rules, the BPMN model is restructured such that the sequence flow edges are removed and the several flow

elements are wrapped in structuring elements, such as sequences, conditionals, parallel blocks and loops (including the respective conditions). This also has the effect that in case the process graph can not be structured, this can easily be detected due to the leftover sequence flows in the model. The resulting tree-structured process model can then be transferred easily to an equally structured programming language.

Basic Language Elements BPMN's basic flow elements are Events, Activities (Tasks and Subprocesses), and Gateways. The Gateways, like the sequence flows, are consumed in the structure mapping, so the only elements left are Events and Activities, both of which can be further discriminated given their type, e.g. Message Events, Service Tasks, and the like. These are translated to specific elements of JADL services.

Activities Activities are what the process *does*. They are subdivided in Tasks (atomic) and Subprocesses (composite). The most important Task types are Service, Send, and Receive. Send and Receive Tasks are mapped to JADL's operations for sending and receiving messages, which we will discuss in detail in the part on Messaging later on. The Invoke Task is straightforwardly mapped to a call to another JADL service. The Script Task can be used to provide short snippets of JADL code, which will be validated and inserted at the given position of the resulting service. Subprocesses are 'inlined' in the current service, instead of creating a separate subroutine. This way they still have access to the variables declared by the calling service, as it is the case in the BPMN diagram.

Activities can be repeating. In this case, the results of the mapping are wrapped into a loop statement with the loop condition set accordingly. Further, Activities can have a number of Events attached to them, meaning that the Activity will be interrupted in case the Event is triggered and the control flow will continue after that Event. This is currently only supported for the case of an Error Event, which is mapped to a try-catch block.

Events Generally, Events are things that *happen* in the course of the process, but in fact their semantics can be quite diverse: Most importantly, depending on their context, Events can mean that the process is *triggered* by the given Event, that its execution is suspended *until* the Event is triggered, or that the process *itself* will trigger the Event. They can mark the beginning and the end of the process (Start and End Events), but they can also occur in the middle (Intermediate Events). Of the various Event types specified in BPMN currently only the Message, Timer, and Rule Events are

supported by the mapping, which can be seen as the most important ones. More mappings are currently under consideration.

What's special to Start Events is that for each Start Event a separate JADL service is created, holding the control flow from that Start Event onwards. In case the control flow of two Start Events merges at some point further down the process, a third 'private' service is created for the common part of the process, which is then called at the end of the individual starting services. Further, the Start Event determines the service's input variables. For a Message Start Event, the input variables are the message's payload, or for a Rule Start Event the subset of the process' variables that are being used in the rule. A service that is started by a Timer Start Event has no input variables. In a similar fashion, End Events determine the service's output variables.

When used as an Intermediate Event, a Message Event will behave just like a Send or a Receive Task, respectively, depending on the recipient of the message. An Intermediate Timer Event will cause the service to be suspended until the given time, and a Intermediate Rule Event suspends the service until the given condition evaluates to true.

Messaging What distinguishes BPMN from other process modelling notations is the possibility to model the communication between processes very directly and visually using Messaging edges and the respective Task- and Event types. Obviously, these are translated to communication in JIAC.

For this purpose the process model has been extended with a Message Channel class in addition to the Service Implementation descriptions known from the BPMN specification. These Message Channels can be used for specifying the channels to be used by Send and Receive Tasks and Message Events, while the Service Implementation is still used for Service Tasks. While the service names given in the Service Implementations are fixed at design time, the Message Channels can be composed and assigned to at runtime.

Service Tasks using Service Descriptions are mainly used for invoking existing services, e.g. services that are provided by other agents already running on the platform or the agent's basic services. In the mapping to JIAC's scripting language JADL, these Service Tasks are translated to invoke statements. In the more general case, however, JIAC messages are sent to message channels. Indeed, the generated services can be started by sending an appropriate message to a certain message channel, too, instead of directly invoking the service by its name. For this purpose, a number of

Service Starter Rules are generated (see below for details). In this case the service will be started asynchronously, whereas the invoke statement starts the service synchronously.

Perhaps the most valuable contribution of BPMN is the Event-based XOR-Gateway. These can be used for listening on multiple message channels or for messages from the same channel but with different payloads at once, also in combination with a timeout. This can be used well for clearly modelling complex interaction protocols, such as auctions, as shown in the Example in Section 5.3.4. The Event-based XOR-Gateway is translated to JADL's switch/case construct, which besides simple expressions also allows messages and timer events to be used for triggering cases.

Data Handling Assignments and calculations surely play an underpart in BPMN, which focuses much more on control flow than on data flow. Assignments are not represented by a separate type of Activity; instead each flow element (i.e. Events and Activities) can hold an arbitrary number of Assignments to be performed at the beginning or at the end of the execution of that element. While on first sight this may seem odd, as assignments make up the major portion of most programs, it is in fact beneficial, as otherwise the process graphs would be overly large and hard to overlook.

What's more: This way the assignments are packaged together with the actual activity they prepare or finalise, e.g. assigning values to the parameters of a service call and processing the result. In the mapping from BPMN to JADL, the assignments, together with the actual mapping for the flow element, are wrapped in a sequence, thus defining a separate variable scope. This way, the payload of a message to be sent or received can be declared as a temporary variable in this scope and used in the assignments without worrying about name clashes.

JADL's operations for accessing the agent's knowledge base can currently be modelled only with the help of a Script Task, allowing the insertion of arbitrary code snippets. In the future, BPMN's Data Objects and Data Stores will be used for this purpose.

Data types are not specified in the process diagrams, but only referred to by their full qualified name.

5.3.3.2 From Start Events to Starter Rules

Now that the JADL services have been generated, they have to be started somehow. Of course, each of the services can be explicitly invoked, but we want the services to be executed on the occurrence of the respective Start

Events, as it is described in the process diagrams. For this purpose, a number of Service Starter Rules are generated. JIAC agents integrate a Drools rule engine, which's working memory is synchronised with the agent's own memory, so the rules are effectively evaluated against everything that is currently in the agent's memory, such as services descriptions and intentions, incoming messages, and other facts that have been written to the memory.

The rules' general structure is as follows: The rule condition checks whether the service to be started by the rule is found in the agent's memory, plus an Event-dependent part. In the rule's effect, an intention is created for that service (with parameters determined in the Event-dependent part of the condition) and inserted into the memory, where the intention is eventually found and invoked by the agent's execution cycle.

Currently, three types of Start Events are supported in this regard:

- For Rule Start Events the case is quite trivial: The rule description, (an attribute of the Rule Event) is used as the Event-specific part of the rule condition. Further, the rule description is scanned for occurrences of the process' Properties (i.e. variables), which are then made to parameters of the service.

- For Message Start Events, the Event-specific part of the rule is just the Message to be received on the given message channel (being put in the agent's memory upon receipt) holding the given type of payload (which again is an attribute of the Message Event), and the service's input parameters are derived from the message's payload. Further, in the rule's consequence the Message object is consumed (removed from the memory).

- For Timer Start Event, the case is a bit more complicated. For Drools' Rete algorithm to work, there needs to be some 'Time' object in the memory, being regularly updated and thus triggering the re-evaluation of the rule conditions as needed. For that purpose, the agent has a special TimerBean which does nothing more than maintaining a 'TimerFact' in the agent's memory, regularly updating it with the current system time. This fact can then be used in the rule's precondition. The service is started with no parameters.

5.3.3.3 From Use Cases to Agent Configuration

Now we have a set of JADL services, corresponding to the several Pools in the different Business Process Diagrams, and a number of Starter Rules, triggering these services according to the original Start Events. What's

missing is the aggregation of these services and rules to a multi-agent system. For this aspect, we will take a look at the use case diagram-like overview diagrams, showing the several Participants and the Business Processes they are involved in.

Each of these Participants is interpreted as an agent role (since even if the participant represents a human being, there has to be an agent providing the interface to the other agents), providing the services and rules derived from the Participant's Pools and the respective Start Events. In this stage of the mapping, an agent configuration model is created, holding said agent roles and adequately configured JADLInterpreterBeans and RuleEngineBeans for the services and rules, as well as other essential agent beans, e.g. for communication and discovery, which we will refer to as the agent configuration model.

We decided against generating actual agents and agent nodes straight from the process diagrams, as this would require extending the BPMN diagram with information which simply does not belong there, such as how the participants are distributed across several computers in a network.

5.3.4 Example

In this Section we demonstrate how our methodology is applied using the example of a simple English online auction scenario.

First, the use cases are identified using the VSDT. In our example we have the use cases *Create Auction*, *Place Bid* and *Close Auction*, which are performed by the actors *Auctioneer*, *Seller* and *Bidder*.

The next step is ontology engineering. We have categories such as *Seller* and *Bidder*, holding e.g. their nickname, rating, and contact information, as well as categories like *Auction* and *Bid*. However, we will not go into more detail here and fast-forward to the process engineering instead.

Each use case corresponds to a business process, and modelling these is the main task of the VSDT. *Close Auction* is the most interesting, so we will have a closer look at it.

Each of the actors involved in the use case corresponds to a Pool: Auctioneer, Bidder, and Seller. The auctioneer starts the process when the time of the auction is over. He acts as trustee between seller and bidder, which are more or less just reacting to him. In case there are no bids at all, the seller is notified and the process terminates. Otherwise, both the seller and the awarded bidder are notified, and the auctioneer waits for the bidder to transfer the money. When the bidder informs the auctioneer that the money has been transferred, the auctioneer notifies the seller, who then ships the goods and informs the auctioneer. The auctioneer then transfers

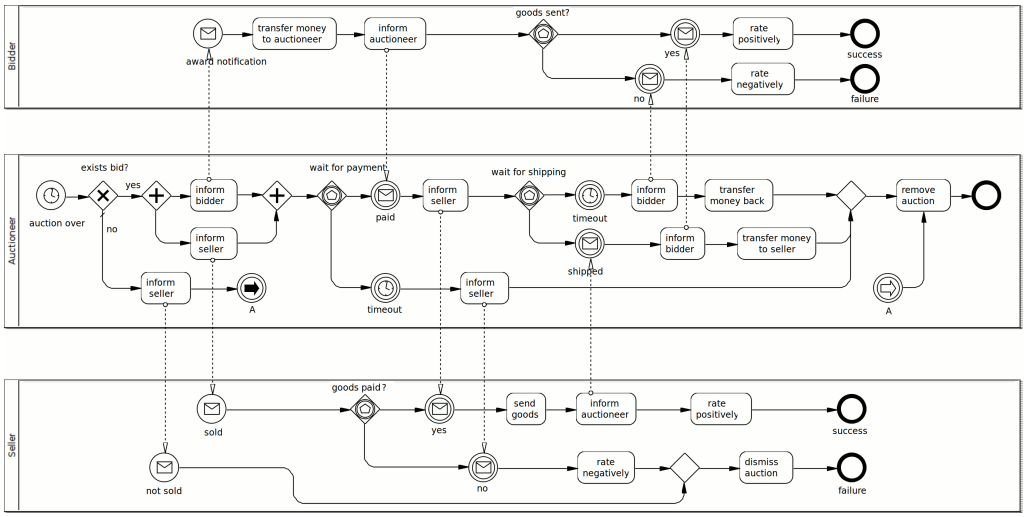

Figure 5.13: BPMN diagram for use case *Close Auction*

the bidder's money to the seller and finally removes the auction from its data store. The full process can be seen in Fig. 5.13. While this diagram shows more than just the "happy path", there are surely many more exceptions and *what-ifs* needed for a real auction, but for a small example this will do.

Next, the process has to be enriched with non-graphical elements and attributes, using the categories defined in the ontology, such as the messages sent, assignments made, and expressions used in branching conditions. While the VSDT does provide means for assisting this task, e.g. basic validation for assignment expressions and conditions, it is still relatively laborious, and we are currently investigating ways to simplify this part.

Next, the VSDT's export feature is used to generate agent roles and JADL services from the BPMN diagrams, which are then detailed further using AWE and JADLedit.

We enrich the role model with additional roles and components needed for the example, aggregate the roles to agents and place these agents on agent nodes. In our example, we have one auctioneer agent and some agents implementing the roles of both, seller and bidder, all situated on one node, as depicted in Fig. 5.14.

Note that while for each actor one role has been created, there are more components (in this case: JADL services) than there were use cases. For each Pool (i.e. each actor participating in a given use case) one service is created, which is reflected as a component in the agent world diagram. For the seller's Pool in the above diagram, a total of three services has been

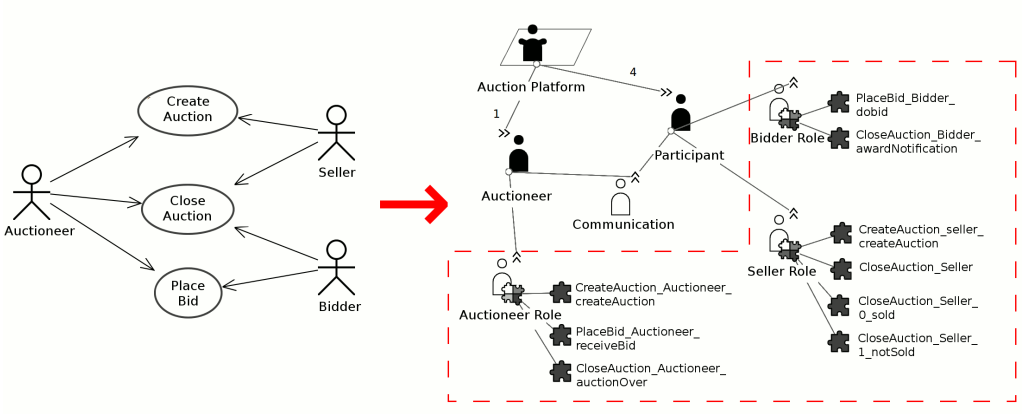

Figure 5.14: Left: Use Case diagram. Right: Agent World Diagram. The framed portion has been generated from the use case and process diagrams.

created, so we will take a closer look at this one now.

The seller's part of the *Close Auction* process begins with either receiving a message telling him that his item has been sold, or receiving a message telling him that no sale was made. This results in two JADL services being generated for the seller, being triggered by different Events and thus different starter rules. Further, as the 'sold' case can merge into the 'not sold' case if the bidder does not respond to the payment request, a third service is created, the common part of both services, which is called at the end of each of the former.

Making things further complicated, the seller's process does not have proper block structure, which is required for most programming languages, including JADL++. Thus, the activity *dismiss auction* is duplicated and moved to the end of the two branches, just before the merging gateway. (As a result, the third service, holding the common part, will be in fact empty.) Thus, the seemingly simple seller process results in three interacting services with some shared logic.

The generated JADL services, holding the necessary communication with the business partners and the basic branches and decisions, can now be further refined and extended using the JADLedit source code editor.

Finally, with the ontology, services, starter rules, agent roles and configuration being in place, the example can be completed with two JIAC agent beans, each starting a simple GUI for specifying auctions and bids and triggering the seller's Create Auction service and the bidder's Place Bid service, respectively, initiating the auction's life cycle and ultimately triggering the auctioneer's Close Auction service.

5.3.5 Discussion

As we have seen in the example, using the methodology presented in this paper, the mapping from BPMN to JIAC can be used to generate a good portion of an executable, meaningful multi-agent system and to easily fill in the gaps that can not be modelled sufficiently well using BPMN.

In this section we want do discuss some problems and open issues we still see regarding the methodology and the mapping. Especially, we will take a closer look at the coverage of the mapping, which parts of BPMN are as yet unused, and most importantly, which notions of multi-agent systems can not yet be modelled using BPMN.

5.3.5.1 Round-trip Engineering

Given a sufficiently detailed process model, the results of the mapping can be readily executable. Still, one has to complete the agent configuration model with actual agents, and as argued in our methodology some additions may still be necessary to the generated service scripts.

While this in itself is not a problem, it can lead to complications when working simultaneously with both the process model and the components generated from the model. While it is possible to merge changes in the process model into the generated files,[3] the other direction — updating the process model with changes made to the generated files — is as yet an open issue.

While the transformation from BPMN to JADL is unambiguous, the opposite is not always the case. E.g. in BPMN a repetition can be modelled either as a looping Subprocess, or by connecting a Sequence Flow to an upstream Activity. Thus, when translating a process model to JADL and back, it is not guaranteed that the resulting process model will look like the original one, although its semantics would be the same.

5.3.5.2 Coverage of the Mapping

On both sides — process modelling and multi-agent systems — there are still concepts that are not yet covered by the mapping.

BPMN: Complex Control Flow While most important features are supported, there are still some elements of BPMN that can not yet be mapped to agent systems, such as some of the more esoteric Event types,

[3] The agent configuration models are, on creation, automatically merged with existing files, while for the service scripts textual *Diff* tools have to be used.

or advanced features such as transactional behaviour, some of which are not even regarded by many BPM suites.

However, one feature that clearly deserves to be covered in more depth is that of Event Handlers. In BPMN the execution of a Subprocess can be interrupted by e.g. a Timer Event being triggered, diverting the process flow to an alternative path, this type of control flow is not yet supported in JADL, and even in Java this pattern would require a complex setup of event listeners and interruptible threads. Only the rather trivial case of an Error Event Handler can be mapped, which can straight-forwardly be mapped to a try/catch block.

Another aspect of BPMN that can not easily be transferred to a textual programming language — agent-oriented or otherwise — is that of unstructured workflows. In many respects, modelling workflows with nodes and arcs is similar to programming with GOTO-statements, and can lead do some hard to maintain processes, if not used carefully. Still, while generally process models should be designed in a structured way, making the processes much easier to maintain and to modularise, in some situations adding an 'unstructured' arc can in fact improve readability. Often enough, such workflows can still be mapped to structured program code, but this may require to introduce additional variables or to duplicate possibly large portions of the workflow, making the generating code harder to read.

Once these advanced control flow patterns — workflows that can easily be modelled in terms of BPMN, which map to complex control flow in structured programs — can be used in the mapping, the process-oriented approach to modelling will pose a real relief to programming.

Agent Systems: Dynamic Aspects At the same time, not all of the concepts used regularly in agent engineering are common to business process modelling, as well. Especially the concept of proactive, autonomous behaviour — often advocated as a unique selling point of agent-oriented software — can be said as being somewhat alien to process modelling. Still, there is in fact one element in BPMN, that could fill this gap very well: The Ad-Hoc Subprocess.

First, we will clarify what we understand by autonomous behaviour in this context: As *autonomous behaviour* we define the agent's ability to select a (series of) activities (a plan) from a set of possible actions so that the execution of the plan will help the agent to achieve a given goal in the current state of the world (or the agent's knowledge thereof).

The Ad-Hoc Subprocess is not given much room in the specification, and it is not supported well (if at all) by BPM tools, either. An Ad-

Hoc Subprocess is given a *completion condition* and contains a number of non-connected activities, which are to be executed in no fixed order and arbitrarily often until the completion condition is met.

Thus, Ad-Hoc Subprocesses are well suited for modelling autonomous behaviour in agent systems, especially when embedded into the context of a broader, fixed process, with the completion condition being the (achievement) goal to be met, and the Activities embedded inside the Ad-Hoc Subprocess being the actions available for fulfilling the goal. (Alternatively, the Ad-Hoc Subprocess may be left empty, which can be used for the case that the agent may use any available action — especially useful for taking other agents' actions into account, which are in general not known at design time.) Further, the Ad-Hoc Subprocess can be modelled to be a looping subprocess, resulting in a maintenance goal, or it can be given an Event Handler, resulting in the goal being dropped when some Event is being triggered, e.g. a timer.

The reason why autonomous behaviour using Ad-Hoc Subprocesses is not yet covered by the mapping is the lack of semantics in the process models. While the individual Activities have a name and a description, and in case of e.g. Service Tasks also lists of input parameters and return values, they lack a machine-readable description of the Activity's precondition and effect, being crucial for automated planning. Extending the process model so it supports such descriptions will be the next necessary step for enabling the mapping to cover autonomous behaviour.

5.3.6 Related Work

For our work, we evaluated a number of similar approaches which combine process modelling with agent technology.

The Agent-oriented Development Methodology (ADEM) [CT05] uses a combination of business processes and agent-oriented models for the overall software engineering, while the agent model is developed by using the Agent Modeling Language (AML) [CT07], an extension to the UML 2 notation. Goal-Oriented BPMN (GO-BPMN) is used for describing agent systems based on processes and the goals they fulfil [BACR08]. This seems like double work done by the modeller as they use goals as finite states of the processes (see for example [Whi09]). In our approach, we use domain ontologies to capture, analyse and model the "what". When using process modelling, we are more interested in capturing the idea and the expertise of a domain expert of "how" things are done or should be done, also through different levels of abstraction. The "how" is modelled in a process diagram, which can be directly executed by the VSDT. They are also transformed to

a JADL script and can be directly executed by JIAC agents. We have chosen the transformation approach here, because process models are not the only source for JIAC-based applications (see also [TKKH09, HKN+09]). So the runtime environment is the integration point for heterogeneous design methods.

The *Prometheus* Methodology [WP04] also integrates the advantages of process orientation into AOSE. The methodology comprises the three main topics System Specification, Architectural Design, and Detailed Design, and provides MAS development on the basis of several different diagrams. Prometheus applies processes for the detailed design of each single agent type. For the specification of these processes, an extension to the UML activity diagram notation is used. At this point, Cheong and Winikoff have shown in [CW06a] how Prometheus can easily be improved by different methods. They have used Hermes [CW06b] here to improve the design process regarding agent interactions, which is exactly the domain of BPMN, only for business processes.

Like Prometheus, the *Gaia* Methodology [ZJW03] has been designed to provide formally guided and comfortable MAS development. The methodology structures itself into Analysis, Architectural Design and Detailed Design. Each phase comprises the creation of various diagrams, each one highlighting particular aspects of the MAS. In comparison with other methodologies, Gaia exceedingly attends to organisational abstractions, with a particular focus on organisational rules and -structure issues. Gaia is not intended to commit to the adoption of a specific notation. With regard to the organisational structure, other notations can be adopted to describe and represent roles and their interactions. A respective appliance of Agent Unified Modeling Language (AUML) is described by Bauer et al. [BMO01].

Klaus Fischer and his research team at DFKI is working on the *DSML4MAS* Development Environment (DDE), a visual IDE for Jade [WHF09] based on the PIM4Agents metamodel [HMF09]. While they use processes for modelling agent plans, too, they use their own simple notation. One feature of DDE which is still missing in the JIAC tools is that of 'protected regions' in the generated code, in which the code can be manually edited and will not be overwritten on code regeneration.

Workflows and Agents Development Environment (WADE) is an extension to JADE, using XPDL [Wor] for modelling workflows, which are translated to Java classes, where each activity is reflected in a method stub [CGB08]. A nice effect of this approach is that workflows can inherit from each other, and that the control flow of the workflow can be revisited without conflicting with the manually written code.

The *Tropos* methodology [BGG+04] is model-driven approach to agent-

oriented software engineering and consists of a visual notation combined with guidelines for five development phases. It provides strong features for requirements analysis bases on the notions of actor, goal, plan, resource and dependency. For design activities it is combined with features from other software modelling techniques such as UML sequence and activity diagrams together with extension from AUML [OPB00] or it inherits notions from the agent framework that is used to implement the multi-agent system such as Jadex [MPP08]. While Tropos is very strong in analysing requirements we prefer the vocabulary and expressiveness of BPMN for capturing and modelling the procedural knowledge of how thing are done or ought to be done.

5.4 AgentStore

AgentStore is a mechanism and tool to support reuse by enabling users and developers to share, search and deploy agents. Web- and API-based interactions allow the integration in the common workflow of developers of multi-agent systems.

5.4.1 Introduction

In an ideal agent world, the agent is surrounded by many other agents that provide services that can be deliberately selected and used, or considered in plans in a more complex decision and execution process. In the real world, new projects will have an agent-based system as the core of the solution, but there are no agents and no services outside the specified system, or the agents cannot access other systems or services.

Inspired by the perplexing simplicity of the installation process of applications to Apple's consumer electronic devices iPod, iPhone and iPad, but also to a huge number of other devices based on the Android platform, we believe that Apple's App Store and Android's Market can be a good pattern to promote reutilisation of agents and agent-based solutions in agent-oriented software engineering. An Apple or Android device is usually delivered bare bone, with only basic applications pre-installed. The user can then go to the App Store or Market and download applications that enhance the basic capabilities with whatever is needed by the user. This can be as simple as a puzzle or as complex as a location-based social network or an augmented reality app.

Taking App Store and Market as prototype, the *AgentStore* creates a place where developers can upload their agents and other developers can

find and reuse them off-the-shelf. There are certain implications to the multi-agent infrastructure, development environments and process models, and also the willingness on the part of the developer and project managers to support the pattern, because *re-use* is most often out-of-scope in a single project.

The multi-agent infrastructure requires a number of services that allow to deploy and update agents remotely and request information concerning available runtime, installed libraries and user management. Development tools and build systems must be enabled to provide agents as well as reuse existing agents. Developers and project managers must form the habit of looking up existing agents and in turn provide the results of their work, so that they will profit from it in the medium term.

For example: At DAI-Labor we have an agent that is capable of sending e-mails to people and provides this capability to other agents as an agent service. While this agent has seen many upgrades of programming language, framework and APIs, the core implementation remained stable with marginal adjustments. Furthermore, this agent has been complemented with other agents with capabilities to post messages to different channels, such as SMS, SIP or Twitter, and extended with abilities to deliberately select the communication channel depending on which contact information is known, what the preferred channel is and what the most likely channel is to reach the addressee.

Now, having finished many projects with focus on multi-agent systems that require this functionality we discovered that we wasted time and effort just implementing this functionality time and again. If there had been a place to store the agent, a process that keeps it up-to-date and the awareness in the head of developers and project managers, this time could have been saved.

In the following, we describe the concept of AgentStore in more detail, propose a number of processes around it and inspire the community to "get social". For each aspect of the AgentStore we give the abstract term and then map it to the JIAC agent framework family [HKH09, PT09], which is our preferred technology of implementation. But concepts and processes used in this paper are easily adaptable and extendable with other frameworks.

5.4.2 Concept

The AgentStore is a set of tools that allows easy access to ready-to-run agents and scripts. Similar to Apple App Store and Android Market for smart devices, functionality can be incorporated in ones own multi-agent

systems. After installing and starting a new agent, it registers its services with the service directory (or directory facilitator). The services are immediately available and can be used by other agents. In addition to agents, we have scripts that represent plans or plan elements, which can be deployed to agents that are capable of interpreting these scripts. Once deployed the script is registered as a service as well.

AgentStore targets developers of multi-agent systems. Developers can take one or more of three main roles:

- *User* – select agents and scripts and deploy them to runtime
- *Developer* – develop agents and scripts and upload them to the AgentStore
- *Evaluator* – review, rank and comment on agent and scripts

In Figure 5.4.2 we give an overview about the AgentStore design. In brackets you find the technologies we have used to implement the concepts.

As its core, AgentStore is a developer portal with storage space. A web-based user interface allows to access the content and functionality provided by the AgentStore. In particular, the AgentStore offers functions to search and deploy agents to agent runtime environments, as well as to upload and modify them (see Section 5.4.4).

Additionally, AgentStore provides APIs for requesting AgentStore items via software clients such as build system, IDE or runtime management tools (see Section 5.4.5).

AgentStore can access the multi-agent environment. It can find agent nodes that are candidates for deployment of agents. For deploying scripts, AgentStore is capable of finding agents that can interpret scripts.

5.4.3 Metamodel

The metamodel is an instance of the project object model (POM) of the Apache Maven Project[4]. Deployable agents are unambiguously identified by *groupId*, *artifactId* and *versionNumber*. A human readable name and description is given. Especially the dependency management is used to also collect transitive dependencies. The obtained list of dependencies is matched during deployment with resources that are available on the target node. Any build system can be used here to build executables and package agents, as long as it provides needed agent parts and required metadata.

[4] `http://maven.apache.org`

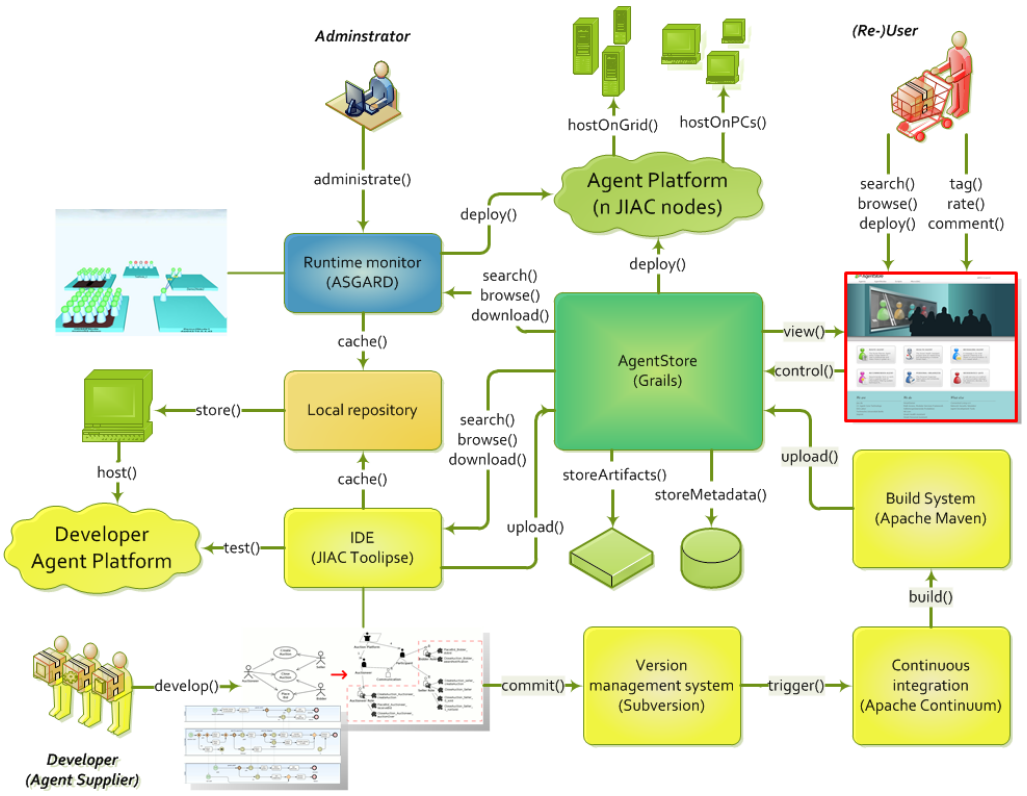

Figure 5.15: AgentStore concept overview

The only extensions we have made to this model are identifiers for the target agent framework and version. This allow us to filter agents based on the available platform nodes, and to use appropriate deployment tools.

Developers upload packaged agents. They also provide a semantic description of the agent's functionality and configurable parameters (filled with reasonable defaults). Agents can be labelled using pre-defined categories describing the intended application area. Additionally, agents can be tagged by users using custom key words. Icons can be provided to brand your agents and make them easily identifiable in the AgentStore.

5.4.4 Processes and Policies

Upload Developers can upload agents and scripts. The actual code is packaged (as a JAR file in our case). A description of the agent as well as a default configuration have to be provided. Then, the uploader's permissions are checked and, if authorised, the AgentStore will persistently store the

given meta data and files. Interested users can subscribe to new uploads and will be informed using their preferred notification channel.

Update Once an agent or script is uploaded, the author can make updates. Each update gets its own version number, and backward compatibility needs to be declared. This is important to ensure that updates do not break systems that rely on specific functions or service calls. Users that have already used earlier versions of this agent are informed about the change and can decide whether to update their agent instance or not.

Search AgentStore provides a number of means to search for agents and scripts. Besides browsing AgentStore entries, agents and scripts can be found by categories or user generated tags.

Deployment Once an agent is selected, the user can adapt the (default) parameters. Then the user can choose where to install the agent from a list of available agent nodes. This list comprises of nodes known to the agent store. Additionally, the user can also provide the URL of a running node — or download a node from the agent store which then can be executed locally. If nodes provide a load balancing functionality (see [TKK+09]), the task of selecting a node to deploy the agent is then delegated to the load balancing agent.

Delete Finally, developers can delete their agents and scripts. Users are notified of the deletion.

In principle, scripts and agents are treated the same. However, if the user downloads a script, an agent with the ability to interpret the script is needed on the selected node. If no such agent is found (or if the user so desires) an agent capable of running the script will be deployed at the same time.

5.4.4.1 Evaluator

The evaluator role has been introduced to allow the evaluation of agent and scripts.

Comment We provide room for user reviews of every item in the AgentStore. Users are encouraged to give feedback on the according agent as well as to make feature requests. In a future release of AgentStore it

is planned to synchronise comments and issue tracking tools in order to enable handling of user feedback in project management.

Ranking User may rate the agent according to its usefulness in a 5-star ranking. We show a Top-25 list of highest ranked agents at the portal page.

Statistics We also provide usage statistics for users and developers. We count page views and deployments and provide usage and update statistics to support the decision process when looking for useful agents.

5.4.4.2 Social networking

When agent developers talk about "social", they think about social capabilities and behaviour of their agents in a multi-agent system, such as speechacts, protocols, joint goals and intentions, or coalition formation strategies. Our aim is to socialise the developer itself by linking AgentStore to social networks.

The AgentStore provides a number of extended functions that deal with the social aspect of developing multi-agent systems. This includes the ability to tag agents, define profiles for automatic search and recommendation, analysis of user behaviour, as well as the ability for users to rank agents and give feedback to the developer. Developers can suggest other agents that are related to the uploaded one or should also be promoted by this developer.

RSS/Atom feeds AgentStore generates a public news feed from events in the store: agents that have been uploaded or updated, latest statistics such as top downloads, and agent ranking.

Promotion Developers can feature their agents in a number of ways. We offer post-to-Twitter and Facebook functionality for both users and developers. For developers this feature can be used to promote the latest agent uploads and updates. For users, this is a fine feature to suggest useful or cool agents to other users. This functionality has potential that has not been explored in depth yet. Twitter posts are automatically inserted into the feeds of people or topics using the "@" and "#" operators to reach certain developers or users. URLs that direct to the AgentStore items are added to the post using an URL shortener, providing additional information about usage statistics in the background. Of course, there is a Twitter agent in the AgentStore, too.

5.4.5 Tool Connections

The AgentStore provides an application programming interface (API) for uploading, searching and downloading agents and scripts. In the following, we will introduce a number of development tools for the JIAC multi-agent framework that already make use of this API.

5.4.5.1 Agent World Editor (AWE)

The AWE [LKHH09] allows to design multi-agent systems (MAS) visually using the concepts of agents, roles and components. The structure of a MAS is designed by drawing agent roles consisting of components (agents beans and scripts), aggregating them to agents and instantiating them on agent nodes. The AWE is capable of looking up agents and scripts in the AgentStore and adding them in the current design scenario. Newly designed agents can be preconfigured and described and then uploaded in order to provide them for reuse.

5.4.5.2 Visual Service Design Tool (VSDT)

The VSDT [KH08] allows analysis and design of workflows using the Business Process Modeling Notation (BPMN). It consists of a full featured BPMN editor, a workflow simulator and a powerful transformation framework that can produce programs from the workflows in a number of programming languages. The VSDT is able to browse agent scripts in the AgentStore and offer them for use as service calls in a workflow. New workflows can be transformed to agent scripts and uploaded to the AgentStore for later reuse.

5.4.5.3 JADLEditor

The JADLEditor [BLM09] is a tool for creating and revising agent scripts written in the JADL [HKBA10] agent programming language. Besides usual editor functionality such as syntax highlighting and code completion it allows using the AgentStore by browsing agent scripts for service calls and uploading newly written JADL scripts.

5.4.5.4 Agent Monitor (ASGARD)

The ASGARD monitor [TK10] allows monitoring and control of distributed MAS. Agents can be introspected and their lifecycle state can be changed.

One can also monitor inter-agent communication and interaction. ASGARD can browse the AgentStore entries. An authorised person can select agents from the AgentStore and deploy them on agent nodes using drag-and-drop.

5.4.5.5 Maven build system support

Apache Maven is a powerful software management and comprehension tool. It is based on the concept of what a project is consisting of (the project object model – POM), and manages the whole software project life cycle. It can publish project information and artifacts in many different ways, and allows sharing project artifacts across many projects. The project model can be extended easily by providing plug-ins that allow to define a custom set of project information and to integrate additional tasks in the process model.

We have built a plug-in that extends the POM with the notion of AgentStore in order to allow uploading and updating agents and scripts during automated builds. The upload/update by Maven is triggered when changes are made in the source codes of agents and scripts and committed/pushed to the version control system.

5.4.5.6 Local repository

When working in projects with government and industry partners, we are often faced with the problem that we are not allowed online access to software and services. Also, internet access is not always available, especially during travel. Therefore, all tools share a common local repository where agents and scripts are stored for design and deployment. The local repository follows the same conventions as the AgentStore does, which is an application of the default Maven repository layout.

5.4.6 Example

AgentStore is being developed as a senior thesis by Fabian Linges[5] together with members of the DAI-Labor of TU Berlin. All main aspects are up and running and the following example is already possible:

Developer A programs a Twitter agent that is capable of posting status updates to his own account on the known service, just for fun. He uploads the agent to the AgentStore. Developer B works in a traffic telematics

[5]Member of Team Brainbug in the Multi-Agent Programming Contest 2010 (see `http://www.multiagentcontest.org/2010`)

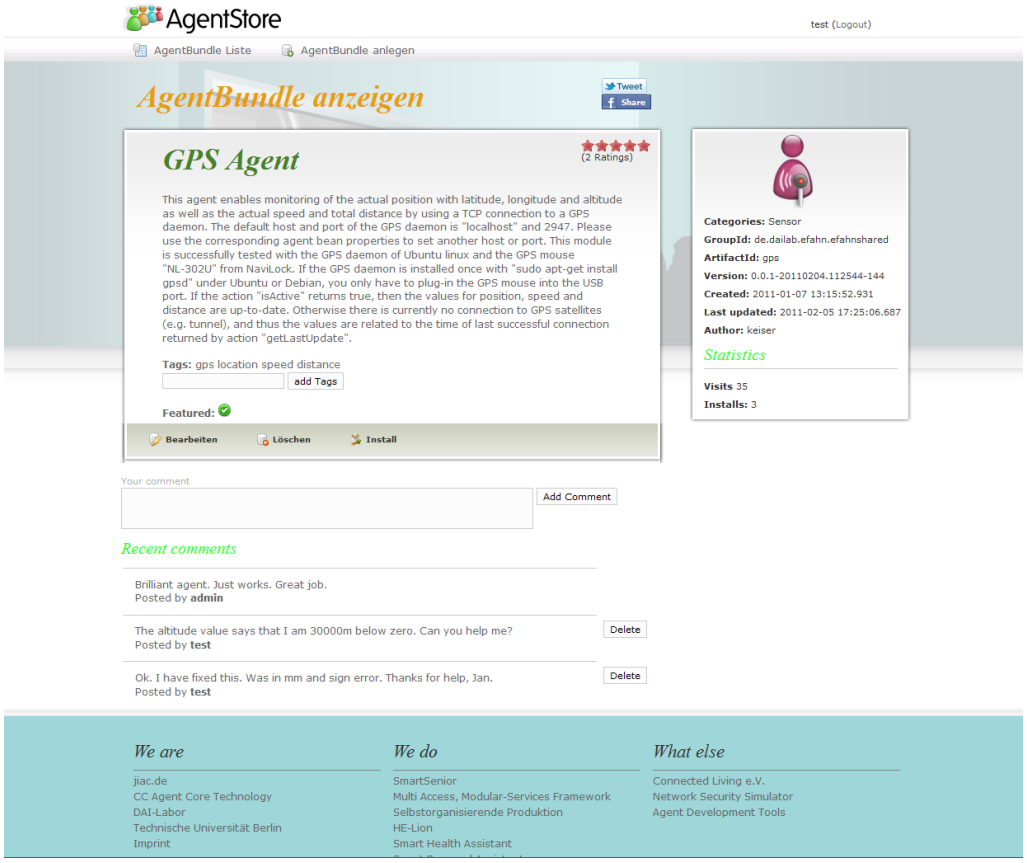

Figure 5.16: AgentStore item view - GPS agent

project and creates a GPS agent that wraps a GPS tracking device and provides agent services for reading position, speed and distance data (see Figure 5.4.6). He uploads the agent to the AgentStore. Developer C is browsing AgentStore entries and has the brilliant idea to twitter his actual position. No sooner said than done, he first deploys the Twitter agent, configured with his own account credentials. He borrows the GPS tracking device and deploys the GPS agent on his laptop. Then he writes a script that routes the position information to the twittering agent (without coping the details of the GPS or Twitter APIs) and deploys the script in an interpreter agent. And is done. His Twitter feed is now posting the position of his laptop. He uploads the script to the agent store and comments on his success story. Developer D reads the story in her RSS feed and can repeat the success in her own project.

Developer E also wants to use the solution. But he has problems with

the GPS agent. He informs B that there is something wrong. B fixes the bug and updates the GPS agent in the AgentStore. All users are informed about the update. Now C and D can decide whether to update their own running agent or not. Now developer A deletes his Twitter agent in the AgentStore because too many tweets are spamming his inbox with acknowledgements. This does not bother C, D and E because they have copy of the Twitter agent in their local repository. Developer F read about the deletion of the Twitter agent in his RSS reader and compensates the AgentStore entry with his social media agent.

5.4.7 Related Work

The AgentStore combines the concepts of an online store with agent oriented programming, and draws inspiration from both of those areas.

While today, most people think of Apple and Apple devices when confronted with a "something" store, the concept has been around much longer than iPhones and friends. However, the App Store and similar Android Market are the most famous instances.

The *Apple App Store*[6] is a program that runs on mobile devices such as the iPhone and iPad and provides access to the iTunes Store, which offers music, videos, and software. The App store focuses on software, and presents the available programs ordered by categories. User can search, browse, buy, and install and give feedback. The user interface is simple and intuitive. The *Android Market*[7] is an analogous program for Android devices, connecting to an application store provided by Google. As opposed to Apple, Android devices can make use of different stores apart from the Android Market. As a final example of mobile stores we want to mention the *Opera Mobile Store*[8] that provides a platform independent web based access to programs for mobile devices running Android, Symbian, and Blackberry devices. Common to these stores is the simplicity with which users can search for and install new functionality for their mobile devices. Often, programs are tightly integrated with the operating system, thereby allowing for genuine extensions of the system.

Somewhat related are web based stores for Perl and LaTeX, called *CPan*[9] and *CTan*[10] respectively. Like the AgentStore, they offer functional extensions that can be browsed by category or searched. However, the soft-

[6] `http://store.apple.com/`
[7] `http://market.android.com/`
[8] `http://mobilestore.opera.com`
[9] `http://www.cpan.org/`
[10] `http://www.ctan.org/`

ware needs to be downloaded and installed manually. CPAN also refers to a command line tool that automatically installs not only the chosen module but also resolves any dependencies, downloading required additional packages without user intervention. *Linux Package Managers* are also quite similar to the Agentstore in that they generally offer some streamlined interface to searching for programs, and support one-click download and installation.

Agentcities [WDB+01] was a large deployment of at its height over 100 FIPA compliant platforms, where agents could provide and look for services. Several national and EU-funded projects pushed the deployment of these platforms. Unfortunately, despite its large developer base and industrial backing, agentcities is defunct now. However, its core idea is related to the AgentStore, with the distinction that the AgentStore does not provide services but allows for the quick and easy installation of agents on ones own platform. Were agentcities alive it could serve as a target for agent deployments from the AgentStore.

5.4.8 Conclusion

In this paper we have described the AgentStore, a mechanism to support reuse by enabling users and developers to quickly and easily share, search and deploy agents and agent scripts. The main focus lies on the usability of the system. Web- as well as API-based interactions allow the integration in the common workflow of developers, thereby fostering reuse without requiring large changes to the normal flow of work. The store is set up such that also other artifacts, such as ontologies, can be stored and retrieved easily [TNNM10].

Rather than forcing developers that want to provide functionality via agents to run them on their own hardware, as cloud-based services or agentcities require, the agentstore merely stores the necessary artifacts, and allows users to configure the agent and deploy it on any platform he chooses, cloud-based or local. Thus, the AgentStore is a middle way between static programmer's libraries, requiring lots of manual work to download and reuse components, and directories of running services, which provide easy reuse of services but require permanent availability and potentially vast computing capabilities.

Apple's App Store and Android's Market also incorporate a simple but powerful business model. It is conceivable that this model can be applied to the AgentStore. We want to extend the AgentStore with self-healing mechanisms of our runtime environment, by enabling agent nodes to download

agents with required functionality from the AgentStore in order to support additional redundant strategies and to provide substitute services.

Chapter 6

Evaluation

> *While it is true that science, to the extent of its grasp of causative connections, may reach important conclusions as to the compatibility and incompatibility of goals and evaluations, the independent and fundamental definitions regarding goals and values remain beyond science's reach.*
> (Albert Einstein)

The essentials of the methodology described in this thesis have been carved out and applied in more than 30 projects, where an agent-based system is part of the solution. The most direct feedback is given in the *Multi-Agent Programming Contest*:

- The contest provides a simple enough to understand, complex enough to implement scenario and a clear goal to be achieved.
- The contest is a clear measure of the overall performance of the participating team.

In research and industry projects, especially with many partners, the evaluation of a methodology and accompanying framework and tools often turns out difficult. In almost any case project partners use their own methodology and have their own preferred technologies to be used. In almost any case only parts of the framework come into operation depending on the task or work package to be solved.

Two larger projects have been selected to illustrate the holistic but agile approach in order to not only achieve the project's objectives but also to apply, refine and enhance the agent framework, the development methodology and tools: the projects *Service Centric Home (SerCHo)* and *Multi-Access,*

Modular-Services (MAMS), which will be described in the second part of this chapter.

In each of the two projects, at least one aspect had a deeper impact on the project success. In the case of SerCHo, ontology engineering has been used for domain analyses to better understand the entities of the domain, their inner structure and their relationships with one another. In MAMS, the distributed architecture and the massively provided services were essential for self-construction of new services.

6.1 Multi Agent Programming Contest

A competition always shows the performance of the participants. We have developed the JIAC agent frameworks over years now and took the contest as a chance to see where we stand. The following sections describe the approaches to the contest scenarios from a software engineering point of view, i. e. how we would solve similar problems of complex and distributed nature.

The contest is also an excellent testbed for debugging and benchmarking. In preparation for the contest we found and fixed many bugs in the lifecycle and execution cycle of our agents and the interpretation of the world model. And, due to the high requirements of the contest, we tuned several core components concerning performance and reliability, making code easier and working more efficiently, which is a benefit also for maintenance and other projects that use JIAC: the JIAC-based *Nessi 2* simulator measured an overall performance gain of their application: improved speed by factor 8(!) after the contest.

Furthermore, the contest is an excellent platform for innovation, promotes sustainable developement and is an evident scenario for teaching AO principles. In preparation for the contest we implemented many features that make the life of the AO programmer easier (easier to learn, easier to use, easier to debug, easier to deploy). And, the contest was the first real application for JIAC TNG/V in 2008.

6.1.1 Agent Contest 2007

In 2007, the JIAC agent team has been prepared by members of the *Competence Center Agent Core Technologies* of DAI-Labor at Technische Universität Berlin. We used the JIAC IV agent framework with accompanying toolkit, which have been created in the course of several projects at DAI

Labor, intended for telecommunications and telematics services to be implemented quickly and effectively, and to be administered reliably.

6.1.1.1 System Analysis and Design

The JIAC IV framework comes with its own customised methodology and a number of tools integrated in the Eclipse IDE.

As shown in Section 4.1.1 (cf. Figure 4.2), the development process starts with collecting domain vocabulary and requirements, which then are structured and prioritised. Second, we take the requirements with the highest priority and derive a MAS architecture by listing the agents and create a user interface prototype. The MAS architecture then is detailed by creating a role model, showing the design concerning functionalities and interactions. We then implement plans, services and protocols, which are plugged into agents during integration. Agents are deployed to (one or more) agent platforms and the application is ready to be evaluated. Depending on the evaluation we align and amend requirements and start the cycle again with eliminating bugs and enhancing and adding features until we reach the desired quality of the agent-based application.

The JIAC methodology is based on the JIAC meta-model. JIAC has explicit notions of goal, rule, plan, service and protocol. Knowledge written in JADL and AgentBeans written in Java constitute agentroles, which are plugged into standard JIAC agents. The standard JIAC agent is already capable of finding other JIAC agents and their services, using infrastructure services and provides a number of security and management features.

For any part of the JIAC meta-model we provide an editor (source code as well as visual editor) in the JIAC IDE for easy agent and application development. Reuse is supported by a plugin that allows search and retrieval of components and solutions. A context sensitive help and a number of interactive tutorials complete the JIAC IV toolbox.

6.1.1.2 Single Agent Behaviour

We started collecting the simulation domain vocabulary and created the ontology containing such concepts as nuggets, gold-digger, grid cells, and so on. Listing 6.1 shows the *GoldDigger* category with its attributes. Furthermore, basic features such as the ability to communicate with the simulation server and a simple path-finding algorithm have been created.

```
    (cat GoldDigger (ext TemporalGridObject)
2     (name string)
      (currentPosY int (init -1))
```

```
4       (currentPosX int (init -1))
        (teammate bool)
6       (carriesGold int (init 0))
        (intention Intention))
```

Listing 6.1: Extract from GoldWorld ontology

In a further iteration, some higher level plans have been designed, embodied into special roles such as *Explorer* or *Transporter*. In particular, we created behaviours for finding gold, moving to a certain position, picking up gold, and scoring. The explorer role had capabilities to systematically search the terrain for gold, the transporter role brings the gold to the depot. Listing 6.2 shows the `makePoint` plan of the transporter role.

```
1   (act makePoint (var ?score:int ?self:GoldDigger ?x:int ?y:int ?world:GridWorld)
      (pre (and
3       (att score SIMULATION (fun Int_DAI_1.sub ?score 1))
        (att self SIMULATION ?self)
5       (att currentPosX ?self ?x)
        (att currentPosY ?self ?y)
7       (att world SIMULATION ?world)
        (att hasGold (fun getCellFromGridWorld ?world ?x ?y) true)))
9     (eff (att score SIMULATION ?score))
      (script (var ?depot:GridCell ?depotX:int ?depotY:int)
11      (seq
          // pick gold
13        (goal (and (att self SIMULATION ?self) (att carriesGold ?self true)))
          // goto depot
15        (eval (and (att depot SIMULATION ?depot) (att posX ?depot ?depotX) (att
              posY ?depot ?depotY)))
          (goal (and (att self SIMULATION ?self) (att currentPosX ?self ?depotX) (
              att currentPosY ?self ?depotY)))
17        // drop gold
          (goal (and (att self SIMULATION ?self) (att carriesGold ?self false)))
19  ) ) )
```

Listing 6.2: Higher-level plan “makePoint”

Furthermore, we exchanged our path finding capability with a generic A* implementation as our local search path finding algorithm did not work in labyrinths. In Figure 6.1 the principle control flow of a single agent is shown, which worked well with the simulation environment.

6.1.1.3 Software Architecture

JIAC IV is based upon the Component Architecture for Service Agents (CASA) BDI architecture described in [Ses02]. It combines a scalable component framework, a knowledge representation toolkit, a control architecture, and an agent infrastructure. Additional features are a runtime environment, system agents, tools, and libraries [FBK+01].

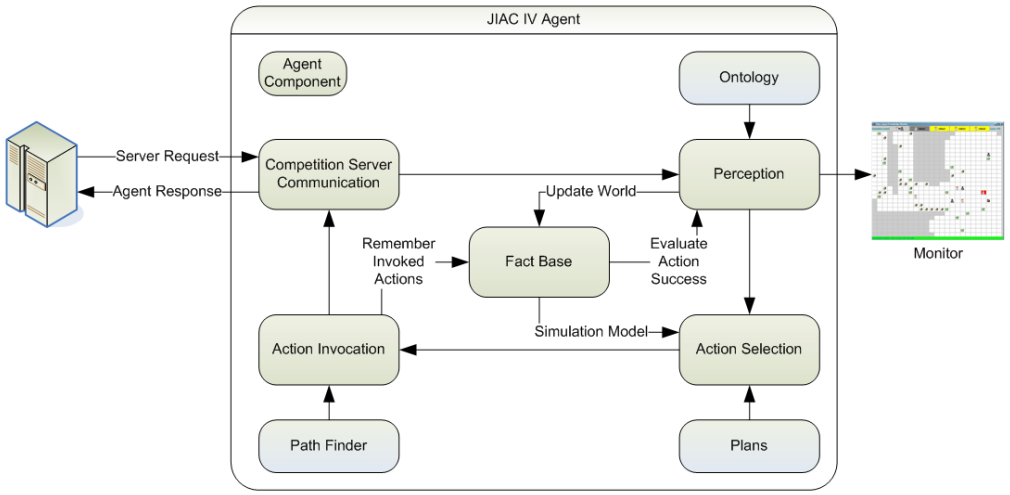

Figure 6.1: Customising the JIAC IV standard agent for the contest

JIAC provides the JADL programming language. Based on three-valued logic, it incorporates ontologies, FIPA-based speech acts, a (procedural) scripting part for (complex) actions, and allows to define protocols and service based communication. Rather than only relying on a library of plans, the framework also allows agents to plan from first principles [KHA06].

6.1.1.4 Agent Team Strategy

When dealing with MAS we always assume that the MAS must be worth more than the sum of its parts. We addressed this in a number of iterations dealing with communication, coordination, and cooperation.

Agents cooperate on a number of levels. First, they share their perception. Next, we enabled our agents to share their agent state (e.g. that the agent carries no gold items) and intentions (such as "I plan to pick gold at X,Y") as they may choose to go to the same unknown field or to pick the same gold item. Now every agent can appraise from what it knows if it will be better to leave the team member alone or to take the intention as its own when its more promising.

Our approach to communication and cooperation is fully decentralised. Each agent has the capability for finding the other agent on the network. It then directly tells every agent about its perception, agent state and intentions. There is neither a message broker nor a central instance which coordinates the contest agents. Every agent builds its own world model from what it is told by the server and the other agents. Every agent also plans

for itself, taking the states and intentions of its teammates into account.

6.1.1.5 Discussion

There are still issues left. Our agents behaved fair, in that they gave way for other agents even if they are opponents. We also assume that the current approach to coordination does not scale very well as it takes $n * (n - 1)$ connections with n the number of agents in the team. Observation of opponents would be a possible extension as well as to guess what they plan and then to crisscross it. There is evidence that this can be solved with some more iterations following the methodology.

Although the scenario of the contest seems quite simple we have found that it is not trivial to design and implement a well performing solution even using the agent-oriented approach, the agent framework and tools. The contest has shown that a practical comparison sometimes provides results that are not as obvious from theoretical deliberations.

6.1.1.6 Conclusion

We have shown that it is very easy and effective to solve the simulation problem using the JIAC IV agent framework. We first collect the domain vocabulary and basic requirements from the scenario description, derive a monitor GUI and basic agent architecture. Then we gather basic capabilities and interactions in roles and implement them. Implemented roles are plugged into the agents and deployed on the agent platform. After evaluating the performance of our agents we collect new requirements, bugs and feature enhancements and start a new iteration.

6.1.2 Agent Contest 2008

Another essential problem of mankind must be solved in this year's agent contest: cow capturing. We present the JIAC approach to this problem by applying the iterative and incremental JIAC methodology and JIAC tools. The solution will be designed and implemented using the next generation of the JIAC agent framework that provides easier way of agent construction, but that is in early beta state. We admire this contest as an evaluation platform for our developments (like our last year's MicroJIAC team).

The JIAC The Next Generation (TNG) agent team has been prepared by members of the Competence Center Agent Core Technologies of DAI-Labor at Technische Universität Berlin. We use the new JIAC TNG agent framework, the successor of JIAC IV [FBK+01], with accompanying toolkit,

which have been created in the course of last year's projects at DAI-Labor. The motivation to participate in the contest was to test the functionality and usability of this framework like our last year's contribution to the competition did with MicroJIAC [TP08]. In contrast to Agent Contest (AC)'07, the scenario of this contest is more complex and requires more coordination and cooperation. Thus, we are implementing a multi-agent system which addresses those issues more than our previous contributions [TP08][HHK08].

6.1.2.1 System Analysis and Design

We follow the iterative and incremental JIAC methodology (see Figure 6.2): First, we collect domain vocabulary and requirements, structure and prioritise them. For example, class *Cow* has attributes *colour*, *methane output* and *origin*. The overall requirement is to capture and keep as much of these cows as possible. We can find some necessary basic requirements such as the ability to communicate with the competition server as well as with other agents, to sense the world and to walk. The next step is to design the MAS architecture by naming agents that play a role in the scenario. We just skip this step because we follow the design of the organisers who have set a team of six agents into being. Then, we derive an agent role model. Each of our agents must play different roles according to the perceived situation and depending on what other agents, friends and foes, do or intend to do. We have found the following roles: *scout*, *herder*, and *smart opponent assistant*. The scout role has capabilities to systematically search the terrain for cows, the herder role is able to direct cows to the corral. The opponent assistant is in itself a complex role. In AC'07 [HHK08], we described our agents as fair, stepping aside when opponents come. This time we concentrated on observing opponents' behaviour and crisscrossing their plans. The agent role model (see Figure 6.3) also contains the interdependencies between roles.

Each Agent has its own perceptions, thus developing its own world model over time and also choosing its actions based on this world model. However, we believe that team communication and coordination plays a vital role in solving the contest scenario. Coordination strategies are specified in more detail in chapter 6.1.2.3.

While designing the MAS using JIAC tools, most of the implementation, integration and deployment artefacts are generated from the design, and are immediately ready to get evaluated, in order to generate new requirements or to change the old ones. The JIAC methodology is an agile development methodology with the outcome of real software agent-based systems. It does

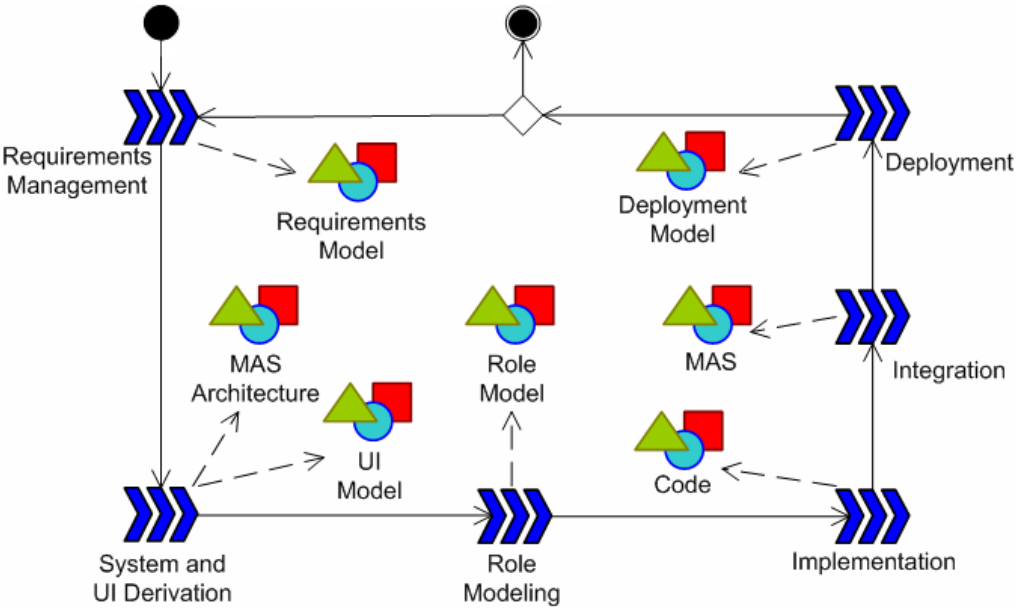

Figure 6.2: JIAC methodology - iterative and incremental process model in SPEM [Obj05] notation

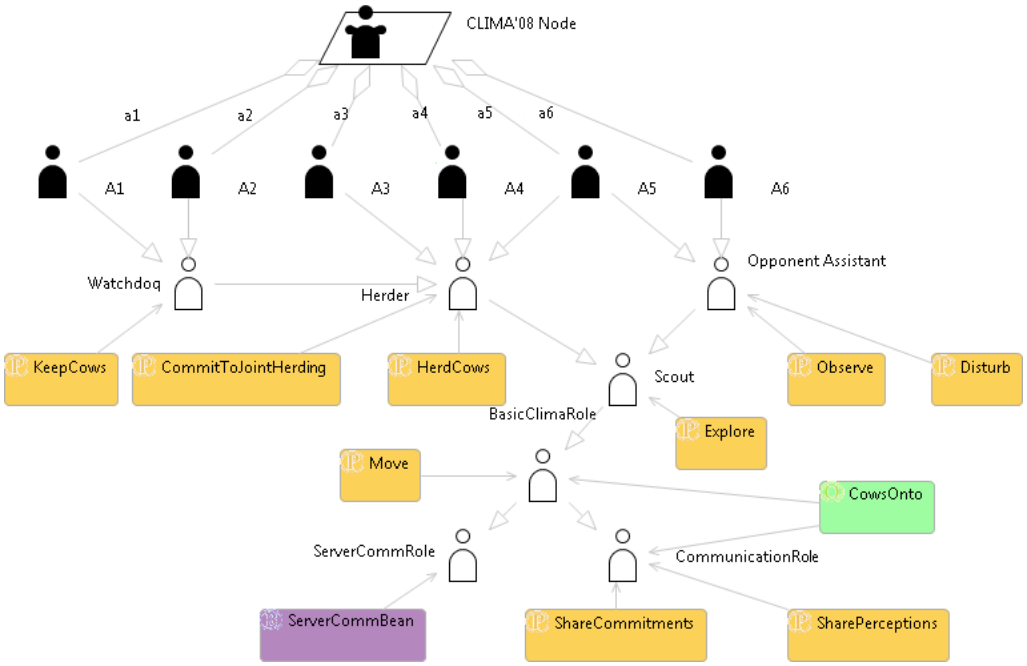

Figure 6.3: The agent role model created with the JIAC AgentRoleEditor tool

not iterate theoretical designs without cross-checking it with reality. As the design is an essential part in the development, it will not win the contest alone. When the competition server became available, short iterations and prioritised increments ensured a strong competitive solution.

6.1.2.2 Software Architecture

Our contribution is realised using the JIAC TNG agent framework which we are currently developing as the successor of JIAC IV. It is aimed at the easy and efficient development of large-scale and high-performance multi-agent systems. It provides a scalable single-agent model and is built on state-of-the-art standard technologies. The main focus rests on usability meaning that a developer can use it easily and that he is supported by the right set of tools depending on what he is doing. Like its predecessor JIAC IV and its smaller brother MicroJIAC, JIAC TNG is implemented in the Java programming language.

The aforementioned roles are implemented with agent components which are the behavioural structures of the agent. They access and modify the agent's state, generate knowledge and trigger the actions. We also use two sensor/actuator components. One component, the standard communication component of the framework, is used for the information exchange between our agents. The other component gathers the perception messages from and delegates the action messages to the competition server (see Figure 6.4).

6.1.2.3 Agent team strategy

A team is always worth more than the sum of its participants. We assume that this must also hold true for the MAS. We addressed this in several iterations dealing with communication, coordination, and cooperation.

Our approach to communication and cooperation is fully decentralised. Each agent has the capability of finding other agents on the network and communicating to them, no matter where they physically reside. Every agent builds its own world model from what it is told by the server and the other agents. Every agent also plans for itself, by taking the intentions of its teammates into account, and also, of course, what it thinks what opponents intend to do.

Our agents cooperate on a number of levels. First, they share their perceptions. Next, we enabled them to share their intentions (such as "I plan to direct cow C to (X,Y)"). This prevents agents from going to the same unknown field or even exploring the same region of the world and thus wasting precious steps. Every agent can appraise from what it knows if it

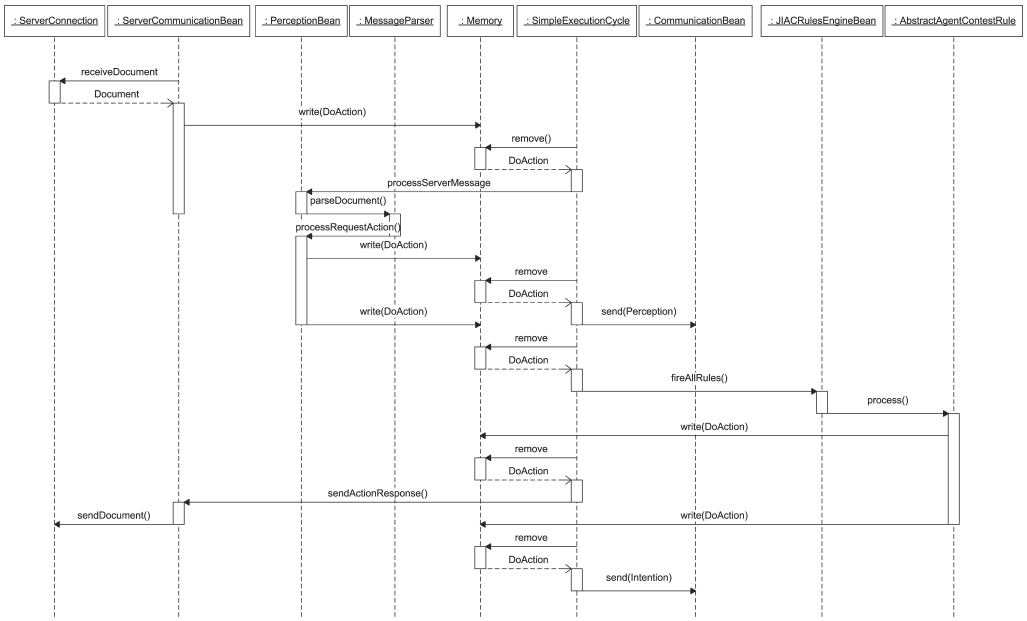

Figure 6.4: The single agent control flow realising general cowherd behaviour

will be better to leave the team member alone or to take the intention as its own when it is more promising.

We did not investigate different coordination strategies. However, we have observed emergent behaviour concerning herd driving: 3 or 4 agents collectively driving a larger number of cows, just arising from communication of perception and intention.

The agents navigate using the A* algorithm. We identified two different cases for its application. First, if an agent wants to explore the area or come to help, it calculates the path between its current position and the destination. Second, if the agent wants to drive a cow, it calculates the path between the position of the cow and the corral. So it knows where to position itself to be always behind the cows. The navigation algorithm treats opponents as obstacles so our agents will not block them explicitly.

Our agents possess two recovery mechanisms. The agent tries to reconnect, whenever the connection between an agent and the server breaks during the simulation. Furthermore, if an agent crashed and must be restarted, the agent requests other agents after the restart to send their actual world states.

We followed the discussion about unmoral agents and decided to let the agents make their own behavioural decisions. As a matter of fact, they are

capable of helping the opponent team scattering the cow herds. Also they avoid the opponents' corral by treating it like fields with obstacles.

6.1.2.4 Discussion

After our success in last year's agent competition we are happy for once more having the possibility to show the maturity of our new agent framework.

Clearly this year's competition was a much greater challenge that it was last year, since now the agents have to cope with "moving targets", making the world a lot more open and dynamic than it was before. There are much more parameters to think of and a greater variability of possible behaviours. When choosing our strategy we tried to consider as much as possible. We also were very curious regarding the other teams strategies.

It is still a challenge and adventurous to use existing agent technology and frameworks. While testing new ideas in this Agent Contest we try to bring forward our understanding of what agents can achieve. And it is always a pleasure to meet up with other teams in a competitive but friendly manner.

6.1.2.5 Conclusion

The JIAC TNG team solved the problem of capturing as much cows as possible. We used the contest as evaluation platform for our new agent framework. The greatest pleasure was the emergent team behaviour, which we did not foresee. It was not clear to us that sharing perceptions and intentions between agents is such a powerful concept. We also appreciate the higher scenario complexity.

6.1.3 Agent Contest 2009

When developing MAS, we follow the iterative and incremental methodology MIAC as described in Chapter 4. During preparation for the Multi-Agent Contest 2009 we used it for the first time together with JIAC V in a teaching project. Therefore it had to be condensed into a 30 minutes presentation and had been used and adopted throughout the semester by students preparing the contest contribution.

The methodology itself has been developed in the context of industrial and teaching projects and reflects more then 10 years of experience in agent-oriented software engineering. It has undergone several iterations itself until we came up with this practical guide to agent development, which covers the necessary aspects of the development lifecycle, and which is flexible enough

to be adopted to organisational structures and cultures. The methodology can be extended and supplemented by sophisticated methods from software engineering, modeling or quality assurance. Here we applied MIAC in a TUB course.

The following sections present the short version of MIAC as theached in the course. A detailed description of the JIAC V MAS will follow in Sections 6.1.3.2ff.

6.1.3.1 The short MIAC Methodology

In MIAC, the development process starts with collecting domain vocabulary and requirements, which then are structured and prioritized. Second, we take the requirements and derive a MAS architecture by listing the agents and create a user interface prototype. The MAS architecture then is detailed by creating a role model, showing the design concerning functionalities and interactions. We then implement the several agent beans, which are plugged into the agents during integration. Agents are deployed to agent nodes and the application is ready to be evaluated. Depending on the evaluation we align and amend requirements and start the cycle again with eliminating bugs and enhancing and adding features until we reach the desired quality of the agent-based application (see Figure 6.5).

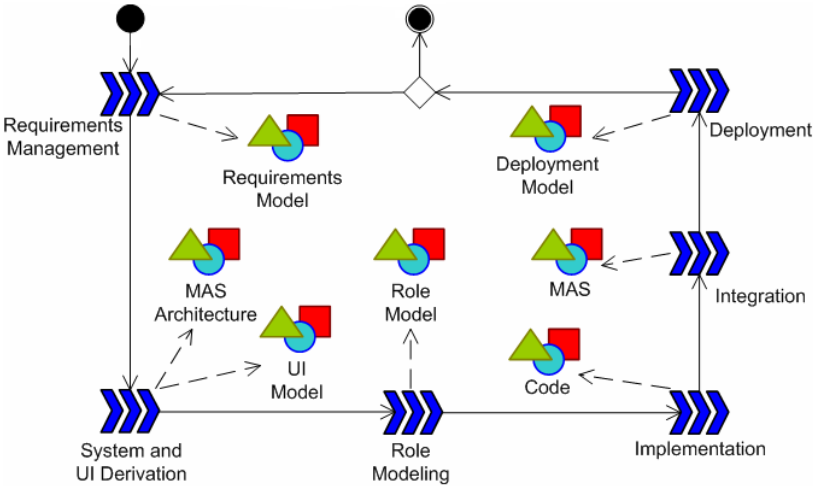

Figure 6.5: JIAC methodology – iterative and incremental process model in SPEM [Obj05] notation

6.1.3.2 Role Modeling

As mentioned above, this year our contribution to the contest has been implemented by students of a university course. All students took intuitively a role-based approach to analysis and design, a role meaning the aggregation of functionality and interactions regarding a certain aspect of the domain. The following roles have been identified. (Note that we name and explain the roles regardless of whether they have been implemented or not in the end.)

First of all, the agents need to explore their environment in order to get to find cows, find all obstacles in order to calculate the best way to their own corral, and find the opponent's corral. This is what is subsumed in the *Explorer* role. We then need to drive one or more cows to the corral, the *Herder* role. We also assume that cows may escape from the corral when someone is using the fence switch, so we presumably need a *Keeper* role. The additional feature of this year's scenario, the switch, leads to another role: the *ButtonPusher*, although we expect that this is not a full time job.

We also identified implicit roles which all agents must be capable of: connect to the server, receive perceptions from and send actions to the server: the *ServerConnector* role. An agent must also be capable of parsing the server message and update its world model, the *Perception* role. The perception role also notices if actions failed or not. Furthermore, each agent should be capable of talking to all other agents of its own team to share its perception and its intention, the *TeamCommunicator* role.

A third group of roles that has been identified concerns the opponent: analyze opponent behavior in the *OpponentAnalyzer* role. Based on the analysis the agents can then interfere with opponent agents' actions, and the *TroubleMaker* role is born. If applicable to the situation, the own agents may try to steal cows from the other corral. These capabilities and interactions can be concentrated in the *Thief* role. We also must take into account that the opponent has the same skill, so we will have to expand the Keeper role with the ability to prevent that opponent agents steal our cows. When discussing the roles, there was no consensus on the Thief and TroubleMaker roles. Some stated that it is not worth to have these extra roles, because when making trouble and stealing these agents could have driven cows to one's own corral.

Last but not least, we identified the necessity to analyze the behavior and performance of our own team, the *TeamAnalyzer* role.

For modeling the several roles, the AWE has been used [LKHH09] (see Figure 6.6). These roles were then aggregated into agents and set on the agent node. The role model was used for generating the agents' XML

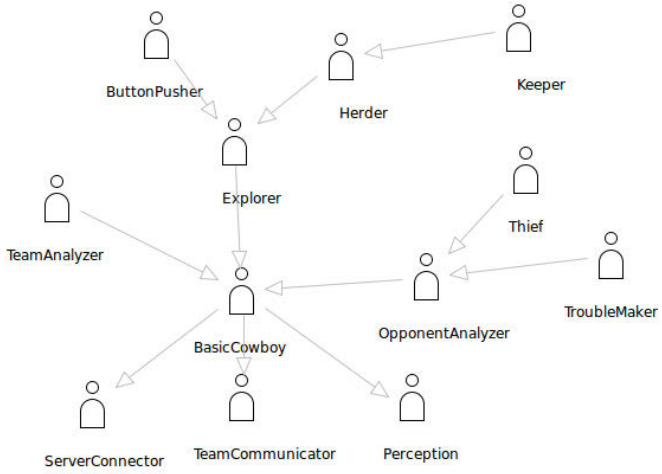

Figure 6.6: Role Model

configuration files. Each role corresponds to one or more agent beans and may include other roles, each one providing one aspect of the roles behavior by providing respective actions, or implementing observers, listening for changes to the agent's memory.

6.1.3.3 System Design and Architecture

Given the roles and functionalities that were identified during the analysis phase, the design of the agents consisted of a number of components that allowed to fulfill the requirements. Recall however that since JIAC follows an agile and incremental approach, requirements are prioritized and can be changed even late in development to gain a competitive advantage[1].

The JIAC methodology creates graphical representations of the agents and associated roles and components, as well as an agent configuration which is in principle executable by the framework. This rapid prototyping supports the incremental design approach taken by the JIAC methodology with early and continuous delivery of a valuable MAS.

The first iteration defined an execution cycle and a number of components used by the agents, namely:

- server connection

[1]See also: "Principles behind the Agile Manifesto". `http://agilemanifesto.org/principles.html`

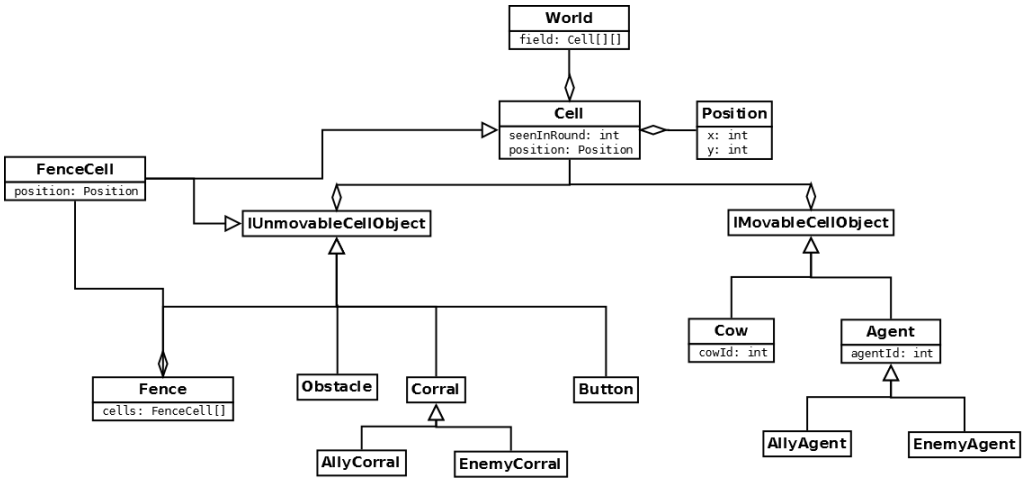

Figure 6.7: Initial world model

- message parser
- world model
- decision component

While the first two components are necessary to connect to the contest server, they are not really interesting in the context of this article. Instead we will focus on the world model and the decision component.

6.1.3.4 The World Model

The world model reflects the agent's (incomplete, possibly outdated) view on the world. It is instantiated from the ontology, which defines the concepts and their relationships. Within the execution cycle, a number of actions are taken. First, messages from the environment are parsed, then, the agent's world model is updated, and finally, a decision on the next move is made.

Figure 6.7 shows the initial world model. In it, the main actors of the environment are reflected, divided into movable and unmovable objects. Agents are distinguished as belonging to the own or opponent team, as are the corrals. It turned out however that the model as initially designed was too heavy on some details and the performance of the systems was also not good.

Therefore, in another iteration, the world model was simplified and all objects were made to be directly accessible from the main world object. Ba-

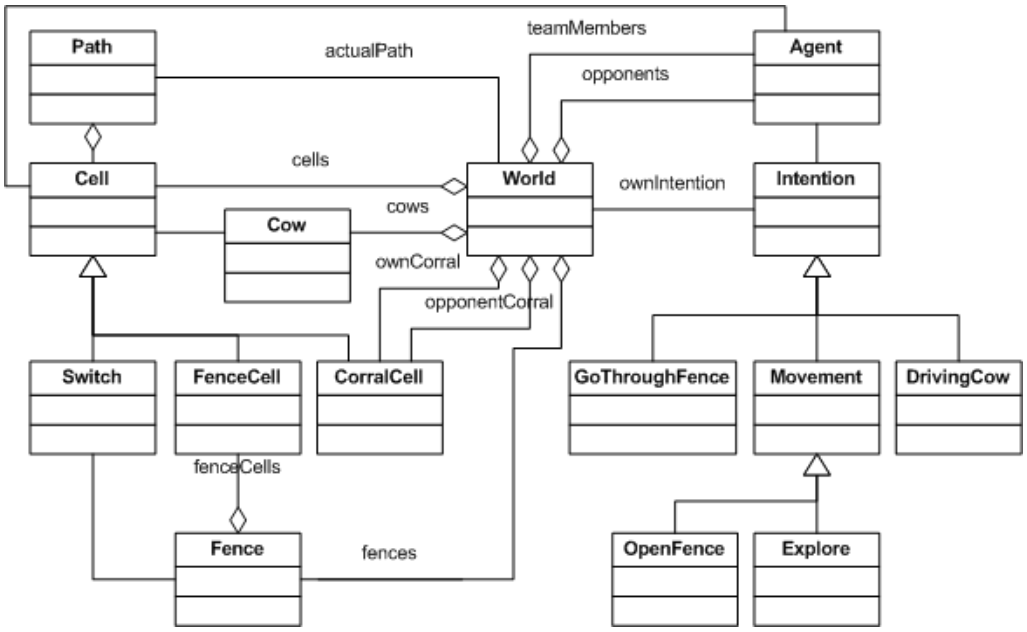

Figure 6.8: Final world model

sic intentions were added to the model, too. Figure 6.8 shows the resulting world model (without attributes for readability).

A *World* object holds information about the grid *cells*, *cows*, *fences*, *corrals* and *agents*, the computed *actualPath* and the *ownIntention* of the agent holding this world object. There are special cells, the *Switch* and the *FenceCell*, which together constitute a *Fence*. *CorralCell*s form *corrals*.

Perceptions are not part of the world model. They are designed straightforward from what is given in a server message and directly update the agent's world model. They are also sent to the team members in order to update their world model, too.

*Intention*s in the world model store additional, domain specific information about what the agent wants to do next and are the outcome of the decision component. An intention here is not just "I want to move west". The specialisation of intention in the world model roughly reflect the roles that are taken by the agent:

- **Explore:** contains the movement information and the number of unknown cells, which the agent expects to uncover there

- **DrivingCow:** reveals the exact cow the agent wants to drive

- **GoThroughFence:** just says that the agent wants to go through the fence given in this intention
- **OpenFence:** the agent wants to open the fence given in this intention

An intention updates the agent's own world model and is also sent to the team members, so that they can take into account the intention of the sender in their decisions. The *ownIntention* is then converted into a server action (e.g. "north"). In a next simulation step, the agent can check whether the action was successful or not, and he will decide on whether to stick to the current intention or reject it.

6.1.3.5 The Decision Component

Again, following the incremental and iterative approach, the initial decision component is just a random move. This allows us to make sure the execution cycle runs reliably. Once the "outer shell" of the agent is implemented, we then add incrementally more complex behavior.

In the first extension, a behavior for finding the next unknown cell is implemented. This is then again extended by adding an A^* algorithm to find the shortest path to this cell. If cows are found, we calculate a path from the closest, to the agent, cow to the corral and drive the cow by positioning the agent behind the cow, i.e. the opposite direction of the cow path.

Once this basic behavior is functional, team communication is added to the system. Each agent broadcasts its perceptions as well as its intentions to the others. The decision component now not only takes into account the private perceptions, but decisions are based on a "global" team world model. Each decision bean now calculates for the single agent on which agent should drive identified cows, based on the distance of agents to cows and cows to the corral. Since the intentions of other team members are known, it is made sure that no agents try to drive the same cow.

Finally, roles are implemented. Following the priorities, we have first implemented the roles *Explorer*, *ButtonPusher* and *Herder*. These behaviors are selected by simple reaction rules based on the current state of the world model, as shown in the following pseudo code.

```
IF no free cow known
   THEN adopt Explorer role
IF free cow known and not herded by other agent
   THEN adopt Herder role
IF cow and herding agent close to fence and self closest to fence
```

```
THEN adopt ButtonPusher role
```

6.1.3.6 Agent Team Strategy

Due to the similarity of the scenarios, large parts of the strategy could be adopted, which we had developed for the contest in 2008 (see Section 6.1.2). To clear this upfront, we did not use any sophisticated, explicit team concept, neither shared plans, commitments and plan alignments, nor peer-modeling techniques. Our approach is more similar to a decentralized team AI as described in [vdS02]:

> "...interactions between squad members, rather than a single squad leader, determine the squad behavior. And from this interaction, squad maneuvers emerge. This approach is attractive because it is a simple extension of individual AI..."

Firstly, every agent builds its own world model from what it is told by the server (i.e. its own perceptions) and its team mates. Therefore, each agent broadcasts its perceptions and intentions (see 6.1.3.4) to the rest of the team. Every agent also plans for itself, without a central point of control, by taking the intentions of its team mates into account. Thus, by sharing both their perceptions and their intentions, redundant actions could be prevented to a large extent. Further, it allows for quickly re-entering the simulation in case an agent should need to be restarted.

Just like 2008, the agents navigated using the A^* algorithm, which was used for both, calculating its own path and for calculating the path a cow should take and where to position itself to drive the cow in this direction. Thus, this algorithm provides basic capabilities for navigation, obstacle avoiding, cow herding, and exploration. The algorithm is slightly adopted using higher costs for neighbouring cells of obstacles (you cannot drive a cow away from an obstacle). Movable objects in the scenario, such as cows and agents, are seen as obstacles as long as the algorithm terminates with a path. This change results in the agent behavior of collision avoiding. Agents start exploring the field, until they find a cow, which they will then try to drive to the corral. The nearest agent takes the job of the herder and sends its intention to its team mates. The other agents keep exploring the field until they find their own cow or everything is discovered.

This year, due to the improved cow algorithm on the server side, single cows are more difficult to drive by a single agent, while it is easier now to drive smaller flocks and at least possible to drive even huge, compact herds. Just like last year, we could observe some emergent behavior here, as the agents will pick one of the cows closest to themselves and to the corral,

which, since cows seldom stand alone, automatically leads to a group of agents driving the flock of cows closest to the corral, which works better than a single agent driving a cow or a flock.

While we planned to have one agent to exclusively watch that no cows escape the corral, a trial simulation showed that this behavior will be covered emergently, too: Whenever a cow escaped the corral, the agent closest to the corral will drive it back inside, while the other agents can go on with their assignments. The basic cow driving behavior made this agent the 'Keeper'.

Also, there was much debate among the JIAC-team developers about whether to implement a 'thief' role, or not. In the end, we decided not to implement a thief, but not to prohibit such behavior, either. Thus, when it turns out that the cows in the enemy corral appear to be those that are easiest to drive to the own corral, our agent would not hesitate 'stealing' them – in fact, they were fully ignorant of them being in the enemy corral in the first place – but they will not do so to damage the enemy. Since we had an issue with 2008's opponents blocking our agents, we decided not to implement such kind of behavior, and are happy that our agents did not emergently develop such behavior, either.

One feature of the scenario made changes in the cowboy agents' behavior necessary: fences. We solved the problem by introducing two new intentions: *OpenFence* and *GoThroughFence*. With these two intentions our agents tell their team mates if they just open a fence for others to drive cows through or if they just want to go through the fence in order to explore the world and find new cow herds. Thus, whenever an agents wants to go through a fence, for exploration or when driving cows, it will broadcast its intention to go through the fence, and the closest allied agent will open the fence for him. This led not only to teams of herders, autonomously driving flocks of cows through fences, but also to teams of two explorer agents, helping each other passing through the fences while exploring the map.

For future editions of the contest, we plan to extend our team's coordination capabilities to further reduce redundant actions and to optimize paths walked by the agents. Further, we think about analyzing the opponent, e.g. which cows the enemy agents are likely to engage next.

6.1.3.7 Technical Details

Our development infrastructure, both for the contest and for the development of the JIAC family of agent frameworks in general, includes version control with Subversion, the Apache Maven build system, a Continuous Integration Server, bugtracking software and a Wiki for capturing ideas.

All of our agents ran on a single multi-core Windows machine which was dedicated for this job during the week of the contest. During the games, the agents were very stable, and due to the constant synchronization of the several agents' knowledge and intentions a crash could quickly be compensated, as the crashing agent's knowledge would not be lost, allowing him to quickly re-enter the running game. However, we experienced some problems *between* the games, when switching from one map to another. Here, our agents repeatedly crashed and had to be restarted on the new map. Apart from this, our agent framework was very stable.

6.1.3.8 Discussion and Conclusion

In general, we have at least three very positive results concerning the contest:

- The contest is an excellent testbed for debugging and benchmarking our JIAC agent frameworks.
- The contest is an excellent platform for innovation and promotes sustainable development of multi-agent systems, frameworks and tools
- The contest is an excellent scenario for teaching AO principles.

Let us amplify this: In preparation for the contest we have found and fixed many bugs in the lifecycle and execution cycle of our agents, and in the updating and interpretation of the world model. Due to the high requirements of the contest we tuned several core components concerning performance and reliability, making code easier and working more efficiently, which is a benefit also for maintenance and other projects that use JIAC (e.g. the $NeSSi^2$ project [DAI], a JIAC-based simulator, measured the overall performance of their application: improved performance by factor 8!). The contest was the first real application for JIAC TNG/V in 2008. And, finally, in preparation for the contest we implemented many features that make the life of the JIAC programmer easier (easier to learn, easier to use, easier to debug, easier to deploy).

Although we have won the contest, we still think there is room for major improvement in single algorithms, single agent behavior, and, of course, in team behavior. The greatest pleasure was, again, to see emergent team behavior while agents are driving cows, keep cows in the corrals and "steeling" cows from the other team. We have reproduced this behavior in this year's contest and we have now a clue how emergent behavior can be "engineered". But for all that, we would like to compare emergent behavior to

an explicit team concept with explicit roles, dynamic role assignments and changing groups following a common, shared goal.

6.1.3.9 Short guide to be competitive

At the very end, we want to give some hints to newcomers. We here sketch what we did every time in our three contest participations, and it seems to be successful. This is our personal point of view and may not be generalized. It is more a rough procedure than a scientific methodology. We think we also can generalize with caution that this procedure is independent from concrete programming language, agent framework or development methodology, although choosing familiar ones may help during implementation.

- In every contest, we took part in, a single agent consisted at least of five components: *Server Connection component, Message Parser component, World model, Decision component, Team Communication component.* So we assume that this is a valid modularization. These components also helped us to distribute the implementation work over more programmers if there were any.

- There seems to be a suitable control cycle that is similar to the stateful agent model (model-based reflex agent [RN03]): *Receive perception, Parse, Update world model, Decide, Send action.* In our case, the cycle is triggered by the server message, except for the initial server registration process.

- We always start development with implementing this control cycle with dummies of parser, world model, and decision component, sending a random move as action.

- We then continue with implementing the parser and ontology, which makes up the world model.

- When the cycle runs reliably, we add real behavior: find next unknown cell. We then add A* to find the shortest path to next unknown cell. There are several implementations of A* available in any language.

- If our agents now find a cow, they calculate the path from the cow to the corral. We place the nearest agent behind the cow. If it is in corral the agent will find the next cow and so on.

- If one does not have a centralized approach to agent team (meaning that the agents share *one* world model) the agents should share their

perception (what they got from the server) and intention (what they want to do). This will improve the basis of decision-making for every single agent and will also boost the team performance tremendously.

We see the above bullet point as necessary steps to be competitive, and beginners following this procedure will reach a position already in middle field of the contest competitors.

- When we come to this point, we start adding details and fine-tuning: *Open fences, Go through fences, What is the best unknown cell (to explore the world efficiently), Learning the cows' behavior, Special tactics.* We collect the ideas, prioritize them and then follow this list while improving our agent team implementation.
- Finally, we make sure that our world model is cleaned up when one simulation ends and the other one starts in order to survive more then one simulation in a row.

6.1.4 Agent Contest 2010

Our team is called "Brainbug" and has been developed in the course "Multi Agent Contest"[2] by the following B.Sc. students: Fabian Linges, Sönke Sieckmann, Erik Stürmer, Jonathan Ziller, supervised by Dipl.-Inform. Axel Hessler (main contact). For the development we mainly use Java and an agent framework called JIAC V [HKH09]. We have invested approximately 320 man hours until the tournament.

6.1.4.1 System Analysis and Design

One of the first things we had to do was to decide on the architecture of our agent system. We had to decide whether we favor a more centralized or decentralized approach. We have decided on a centralized approach with a manager agent collecting the perception of the agents, maintaining one world model for all agents and coordinating the behavior of the team. We have chosen this approach mainly for two reasons: we expect that a centralized approach is more straightforward to implement and to reduce the amount of messages sent between agents to linear complexity, sending only one perception message to the manager and one instruction message to the single agent.

The outcome is some kind of supervisor, an outer force behind the agents on the field. Therefore we have implemented the MasterAgent, an agent

[2]Project 0435 L 774 at TU Berlin, Germany

that runs additionally to the normal agents, which we will call field agents from now on. The MasterAgent is responsible for deciding the overall strategy of the field agents and could be called the "brain" of the team. Furthermore all agents (field agents and MasterAgent) share a common model of the world. What happens every turn is that the field agents update the shared world model with their perceptions.

Based on the updated world model the MasterAgent then decides what needs to be done next. It tries to identify cow clusters in the world, decides if fences need to be opened, and which cows should be captured next. However, we wanted to keep at least some degree of independence in the field agents, so we chose to implement a role system. Every turn each field agent gets a role assigned by the MasterAgent accompanied with a task. Tasks can reach from simple *goto* orders to general *explore all* commands. After the MasterAgent has decided what has to be done he sends each field agent a message with the new role and task. Then the agents decide for themselves what has to be done in order to accomplish their set goal.

6.1.4.2 Software Architecture

We use Java and XML as programming language and eclipse as our main IDE. This decision was rather easy, because JIAC [HKH09] uses Java and XML, too. As for agent-oriented concepts the nearest one to our concepts is that of *delegation*, in principle, a very simple agency without self-interested agents.

6.1.4.3 Agent team strategy

The main navigation algorithm we use is A*. Every role may have a modified version (it has different things to avoid or it uses different weights). So th use of A* is bound to the role the agent plays and depending on which variation of A* is used, agents, cows and fences will be avoided. The team coordination works in a centralized way. The communication works with messages from the MasterAgent to the field agents and the other way around, but every round the number of messages is constant. All Agents share the same world model, so everyone knows what everbody else sees. The only background processing that we do is saving the actual world model to a file in case the program crashes.

6.1.4.4 Roles

In the following part we describe in detail the roles every field agent can take on.

Explorer

- Explorer - an agent that is responsible for exploring unknown points of the grid world
- MasterExplorer - assigns a point to explore for each explorer

An Explorer tries at first to explore directly the cells that he can explore with one step. For everyone of the eight directions he calculates which would explore the most cells at once. If the agent can not explore an unknown field directly within one step, it looks up the nearest 'explore point' and goes there. But as soon as the agent comes in sight of that point he switches back to direct explore. If it needs to pass a fence the agent enregisters itself at the MasterSwitcher which organizes another agent as door opener for the time the agent needs to go through this area.

The MasterExplorer has a list of points to explore. These explore points are calculated at the beginning of each round. They are either unexplored or have an unexplored neighbour. These points are uniformly distributed at the map with a configurable gap of steps. Cells that are explored but which are last seen over fifty steps ago become 'forgotten'. Therefore the knowledge of the world is up to date.

Switcher

- Switcher - Agent that is responsible for opening one specific switch
- MasterSwitcher - decides which agent will be the switcher for which switch

Whenever an agent has decided on his path to his goal, he looks up if there are any fences on his way. If that is the case it will tell the MasterSwitcher that he wants to pass that fence. Normally it will do that only for the next fence on his path. The Masterswitcher memorizes which agent wants to pass which fence. Whenever an agent has passed a fence it deregisters and tells the Masterswitcher about it. If an agent changes its goal it deregisters itself from any fence he wanted to pass.

The Masterswitcher looks up which agent has the possibility to become switcher for a specific fence. Every agent that is not already a switcher can be assigned this role. Whenever an agent is the only one that wants to pass a specific fence it is not considered as a possible switcher. This has the effect that agents help each other to pass a fence. For example, when a group of five agents wants to pass a fence and one of them is chosen the switcher for that fence, it will open the fence. The other four agents pass

the fence and if possible one of these four will hold the fence open from the other side so the first switcher can pass the fence too. If everything works fine no agent has to stay behind because of a fence. The MasterSwitcher choses the best switcher by sorting the possible switchers by their distance to that fence given by the maximum norm. Then the MasterSwitcher tests if that agent can reach and open that fence. If that is not the case it will test the next possible switcher.

When a switcher has been selected, the MasterSwitcher sends it a message to play the switcher role and the task to open the according fence.

The path finding algorithm for testing, whenever a possible switcher can find a way or for a switcher to find his way, avoids paths through this fence it has to open. So a switcher should not pass the fence it has to open. Other fences are not avoided but require another switcher. So, if a switcher has to pass a fence to reach its switch another agent will open the fence it has to pass for it.

Shepherd

- Shepherd - Agent that is responsible for herding a specified cluster of cows
- MasterShepherd - determines, which cow clusters are most rewarding and assigns the role Shepherd to field agents

Cows are grouped in clusters. A cow is part of a cluster if the cow is in a configurable radius of another cow that is already in the cluster.

The MasterShepherd sorts the cluster after their distance to the corral and then choses the agents that are the shepherds for that cluster. The number of shepherds for a cluster depends on the size of the cluster. If a cluster decrease or increase more or less shepherds are assigned to that cluster. An agent for a cluster is chosen by its distance to the cluster center much like the switcher is chosen by its distance to the switch. Every agent is a possible shepherd that is not already a switcher.

Every cluster has an ID and shepherds are told the ID so they know, which cow cluster to drive.

For the assigned cluster, the path to the center of the corral is calculated by chosing a number of random cows and taking the average direction. Then we apply a 'boundary' to the cluster that includes every cell directly besides the cluster. Along the back of this boundary (related to the direction to the corral) are a number of possible 'drive' points calculated. Where the back of the boundary is the opposite direction of the next step on the path to the corral. The shepherds then go there and thus drive the cluster towards

the corral. A drive point is a cell from which a shepherd can force a cluster to move. From these possible drive points the final drive points are chosen for each shepherd.

The fences along the path of the shepherds and the cluster will be registered so they can pass without problems. Each shepherd choses the drive point by itself, which is the nearest to its position if that point is not already taken by another shepherd for that cluster. In short, a group of shepherds tries to position itself behind the cow cluster to force it to move on the path to the corral.

6.1.4.5 Monitors

To get an impression of our debugging tools, please, have a look at Figure 6.9 and 6.10. The *Info monitor* (Figure 6.9) shows information about switches (left hand side), cow clusters (right hand side) and field agents (beneath).

The switch info screen presents identified switches, which agents want to pass which switch and which agent has been selected as a switcher.

The cluster screen shows spotted cow clusters and assigned to this cluster cows. It also shows field agents that have been selected as shepherds and the assigned cluster.

The lower screen of the Info monitor represents the actual agent positions and the intended action that has been sent to the contest server.

The *World monitor* (Figure 6.10) has two perspectives. The first perspective gives an overview of the map as seen by the selected field agent. The grid can overlay the view. Known areas are colored white. Own agents are shown as little white men, opposing agents are black. The visible area of the selected agent is colored yellow. Areas that have been explored and actually seen area are shown in different shades of gray, the darker the longer they have not be seen. Additional information can be shown, such as the target field of the selected agent, the selected cow to drive and the precalculated paths of both the agent and the cow. The second perspective shows the simple herd recognition, using a simple blur filter. The whiter an area appears on the monitor the more cows are standing in the region and the more compact the herd is.

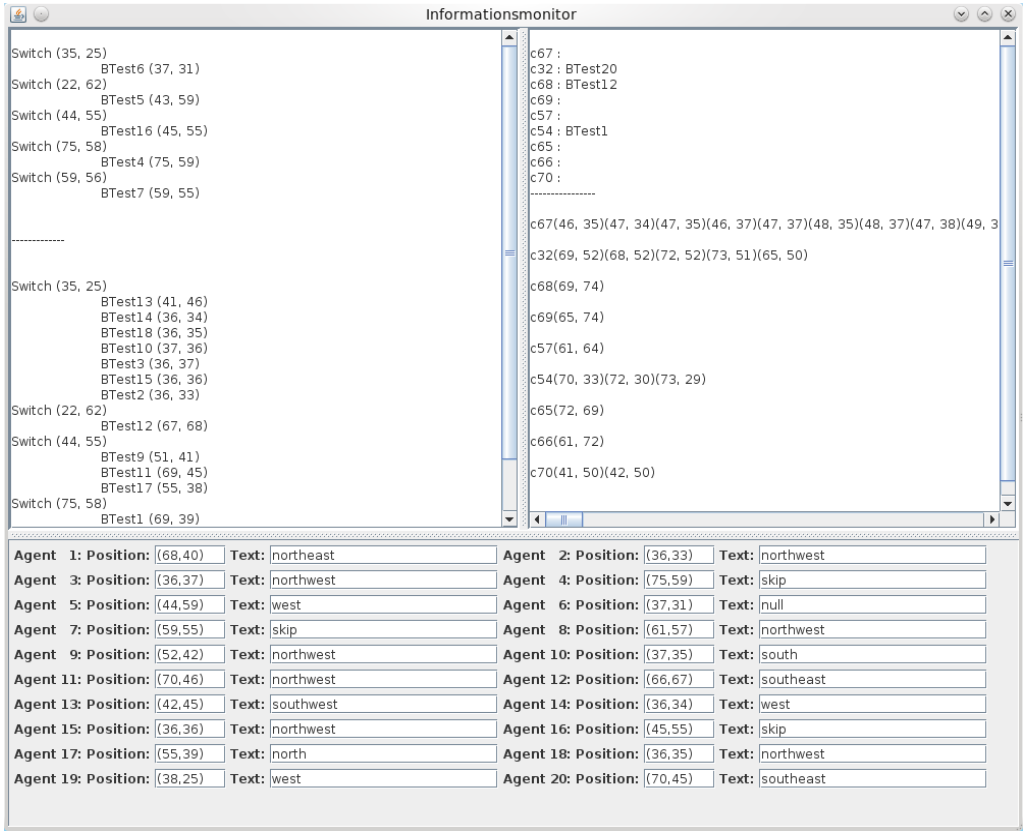

Figure 6.9: Info monitor

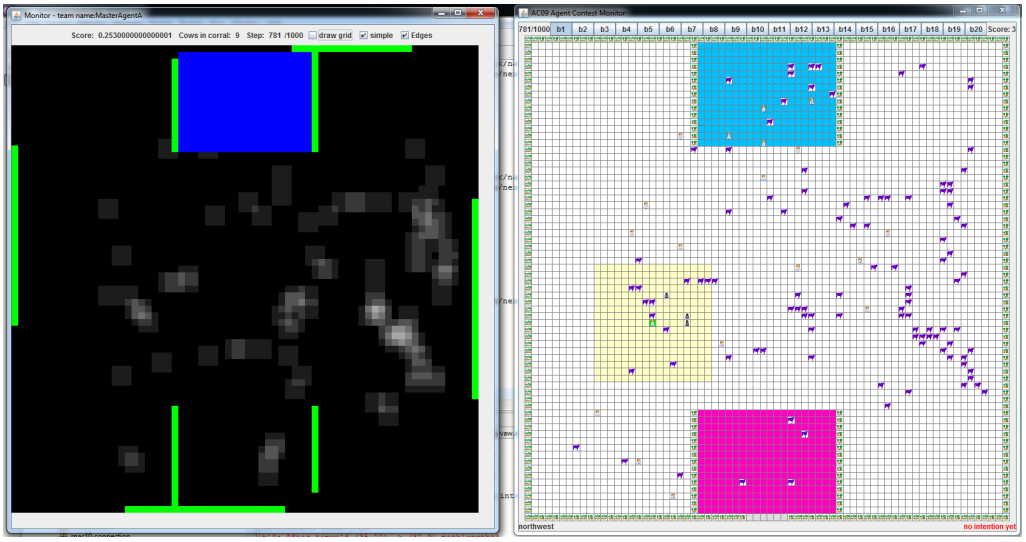

Figure 6.10: World monitor: On the left the herd recognition can be seen, while on the right the corresponding cows are visible.

6.2 SerCHo - Service Centric Home

Corresponding to the research areas of the DAI-Labor the SerCHo project has been focused on three major topics:

- a reference architecture for service provisioning
- hosting and execution for multi-access, multi-modal user interfaces
- easy service creation process and tooling

The first and the last topic is of major interest in this thesis. The second topic has been covered in [Feu08]. The idea behind service engineering using the agent-oriented paradigm has been scatched in [HKHA06].

6.2.1 SerCHo architecture

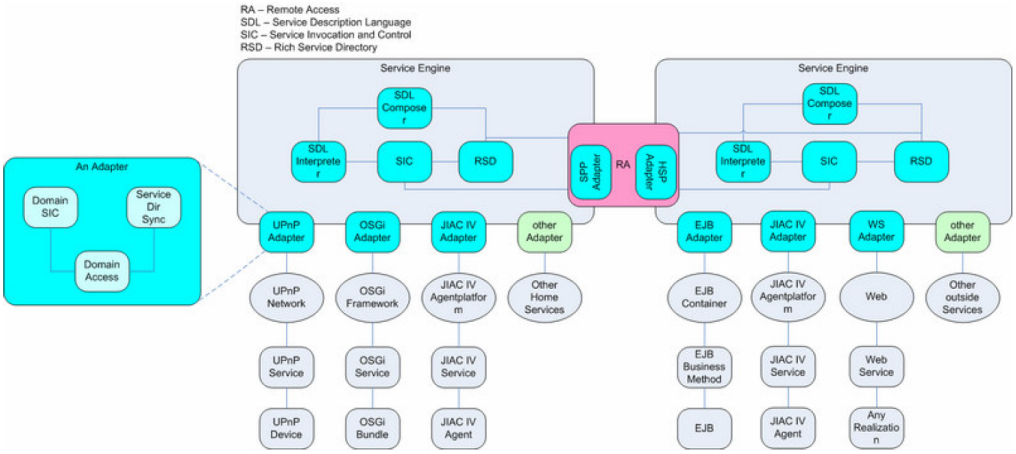

Figure 6.11: The SerCHo service overlay

The SerCHo architecture faces a two-fold challenge: enabling services on the operator side *and* at home at the same time. For this, the SerCHo platform consists of two parts: the Home Service Platform (HSP) in the home network and the Service Provider Platform (SPP) in the operator network. Both must work together seamlessly and secured.

The SerCHo architecture is a service-oriented architecture as it provides the fundament on which service applications could be deployed and can interoperate with each other. It is indeed a distributed platform where several servers are interconnected by transport, security and messaging protocols (although there are only two in the model: HSP and SPP; see Figure 6.11).

It is also a semantic service overlay over a number of service-oriented technologies. The goal was to unify service discovery and the service execution and control of a number of service-oriented technologies and to enrich according service descriptions semantically. This enables a number of possibilities in a dynamic, open ubiquitous and heterogeneous service-oriented environment:

- Technology spanning service registration and discovery
- Technology spanning service invocation and control
- Dynamic service composition and execution

Figure 6.11 gives an overview about the service overlay architecture. It shows two *Service Engines*, which are in fact two JIAC nodes in their own JVMs. Each node has four core components:

Rich Service Directory (RSD) for dynamic service registration and semantic service discovery

Service Invocation and Control (SIC) for bridging service invocation in different service technologies

SDL Interpreter for the execution of complex services descriptions in the form of scripts

SDL Composer for dynamic creation of complex services on the basis of planning algorithms and existing services

The Service Decription Language (SDL) is an extension of OWL-S with AAA and QoS concepts.

Additionally, a Service Engine provides a number of *Service Adapters* for integration of different service technologies:

UPnP adapter for discovery and control of Universal Plug'n'Play (UPnP) devices at home

OSGi adapter to access the services of an Open Services Gateway initiative (OSGi) device where a platform may be embedded at home

EJB adapter to access Java Enterprise technology, i.e. the services of *Enterprise Java Beans* on the service provider side

Webservice adapter for the intergration externally deployed services

The HSP- and SPP-adaptors allow to synchronize registered services between the two platforms and grant secured service access.

6.2.2 SerCHo development process

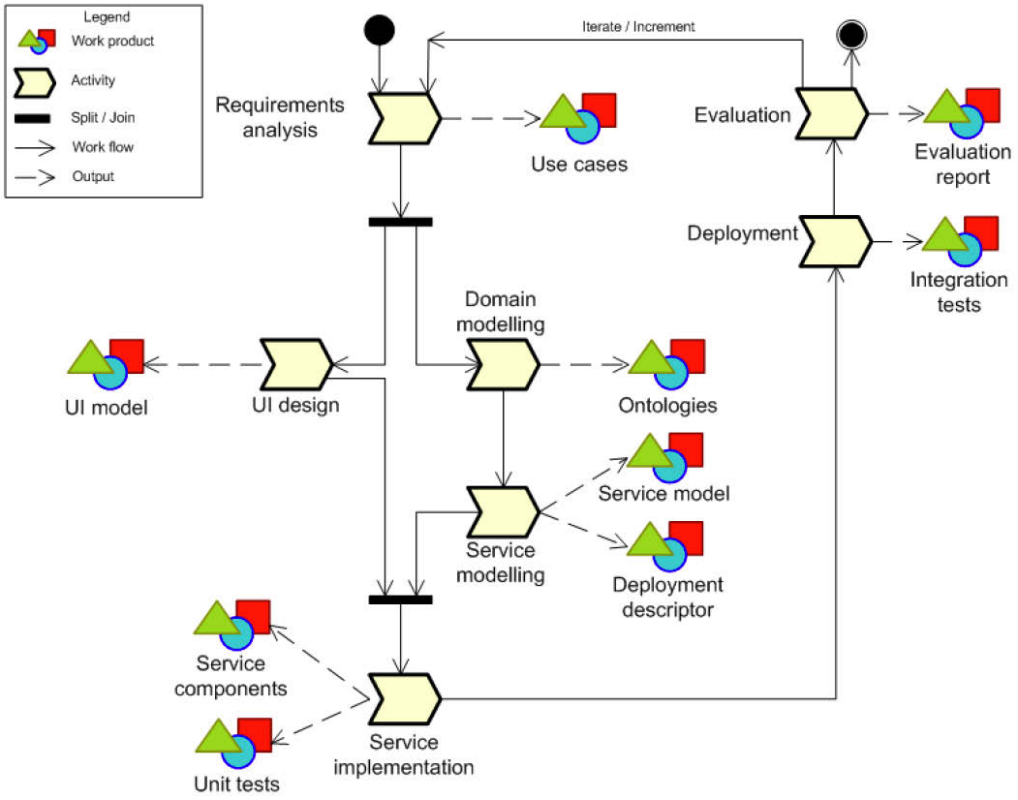

Figure 6.12: The SerCHo development cycle in SPEM notation [Obj05]

Since SerCHo had a Service-Oriented Architecture in place, a Service Overlay and a number of basic services already deployed as infrastructure in the SerCHo architecture, it could then concentrate on the service logic at application level (see Figure 6.12).

During *requirement analysis* the service developer expands the service idea while he/she is identifying stakeholders in a service scenario, their goals and needs, how they interact with each other, and what the service-to-be can do to achieve the goals and meet the needs. The outcome of the process step is a number of use cases, scenarios and a list of detailed and prioritised requirements which guide the next steps toward the running service. *UI design* utilitizes Concur Task Trees (CTT) [Pat99] with an won development process for rapid prototyping of multi-modal user interfaces (see [Feu08]).

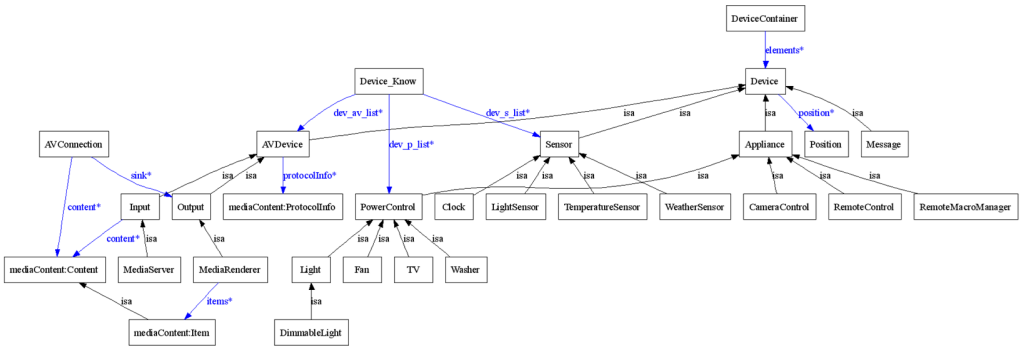

Figure 6.13: Device ontology developed for the SerCHo project

6.2.3 Ontology engineering in SerCHo

In the SerCHo project a working definition of what an ontology is has been drafted for pragmatic use in the SerCHo methodology:

> An **ontology** is a formal explicit description of concepts in a domain of discourse (**classes** (sometimes called concepts or categories)), properties of each concept describing various features and attributes of the concept (**properties** (sometimes called roles or slots)), and restrictions on slots (restrictions (sometimes called role restrictions or facets)). An ontology together with a set of individual instances of classes constitutes a **knowledge base**. In reality, there is a fine line where the ontology ends and the knowledge base begins.

Figure 6.13 shows the *Device ontology* developed in the SerCHo project. The class *Device* represents all devices. Specific devices, the *Jenoptik Digital Camera C 3.1 LI* and the *Siemens Oven HB 38056H* are instances of this class. Interesting in this case that, strictly speaking, the Jenoptik Camera is a *DeviceContainer* that consists of three devices: a *PowerControl* (to switch the camera on and off), *CameraControl* (to access the functions of the camera such as focus, shutter, flash, etc.) and *MediaServer* (to access photos and videos made by the camera).

6.2.4 Process modeling in SerCHo

A major challenge to any software developer is to capture the business idea and proceedural knowledge of the client. Service engineering in SerCHo targets small or medium enterprises. One cannot assume that all business

owners know methods or techniques of Object-Orientation (OO) such UML diagrams or Rational Unified Process (RUP). So we decided to use a very simple, common vocabulary and restricted ourselves to a basic modelling technique, BPMN, to gather the business ideas and how things can be accomplished.

The approach has been supplemented by the VSDT (see also Section 5.2.3) that allows visual modelling of the ideas and to use and integrate existing components into the solution. During service development the developer can use several extensions to validate and simulate his design. Finally, the VSDT transformation framework allows to translate the business diagrams into JADL scripts and deploy them on the HSP or SPP.

6.3 MAMS - Multi-Access, Modular-Services

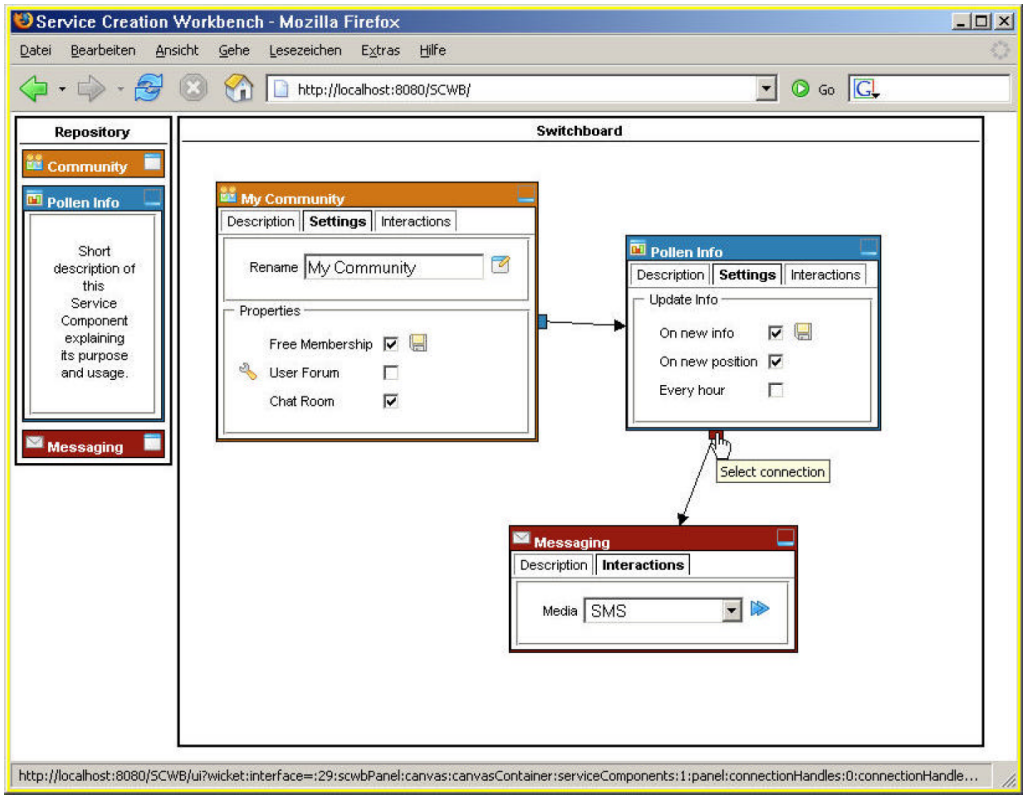

Figure 6.14: Graphical service composition in the MAMS project using the Service Creation Workbench

The Multi-Access, Modular-Services (MAMS) project has been accomplished in two phases over three years between 2006 and 2009. The DAI-Labor used the JIAC framework to provide an open distributed service platform in order to deploy and run user-generated services [FSDM07, TKK+09, TKKH09].

The MAMS objectives were to deliver a framework for service providers consisting of a distributed runtime environment, a repository of service components (basic services) and a visual service composition tool usable for non-IT experts.

The outcome of the MAMS project was an architecture and a prototypical implementation of the Open Distributed Service Delivery Platform (ODSDP), a repository containing about eighty basic services divided into eleven categories and a Service Creation WorkBench (SCWB), where users can compose services graphically (see Figure 6.14). Seven show cases in the health care/prevention and business communication have been realised to show the power of the approach.

6.3.1 The MAMS architecture

Core of the MAMS architecture is a distributed platform consisting of several JIAC V agent nodes with deployment management and load balancing – the ODSDP. The number of nodes needed is calculated by the load balancing solution and scales using Intelligent Service Oriented Network Infrastructure (ISONI) to guarantee unobstructed operation.

The ODSDP provides some features that are not part of a standard JIAC platform (but became optional components):

- load measurement
- load balancing at start time
- load balancing at runtime
- a grid interface (for the use grid infrastructure)

Additionally, the ODSDP encapsulates proprietary services that form the so-called *Network Abstraction (NA)* and the IP Multimedia Subsystem (IMS). It also provides interfaces to manage and deploy services to the SCWB.

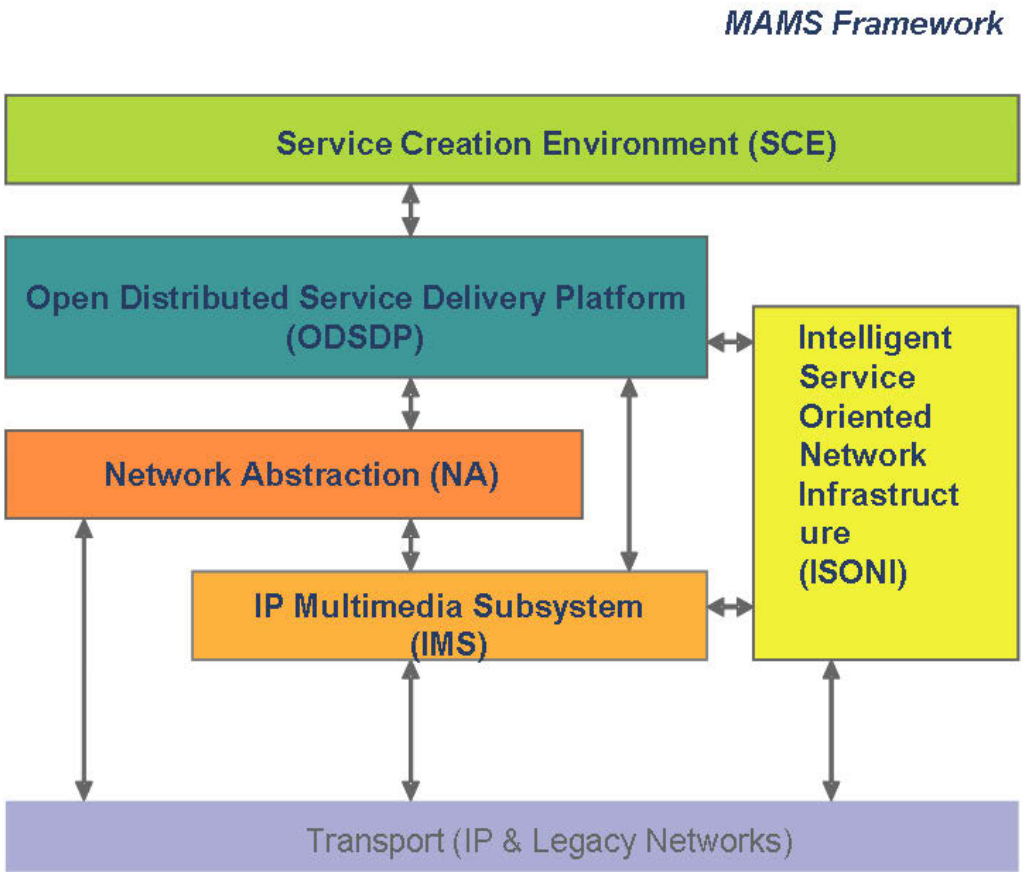

Figure 6.15: The MAMS architecture

6.3.2 The MAMS process model

The MAMS project has used the MIAC process model to develop about 80 basic services. During development each basic service runs throught MIAC with only touching the *integrate* activity implicitly. Integration is done in a separate process model called *4plus1 phase model* (see Figure 6.16). In this process model, non-experts select basic services from the repository and compose services using the visual SCWB. The composed services can be deployed to a testing ODSDP and, if positively evaluated, deployed to real-life operation. Feedback is given during selection and composition phase regarding missing basic services or change requests of existing services. Experts then create or adopt respective basic services.

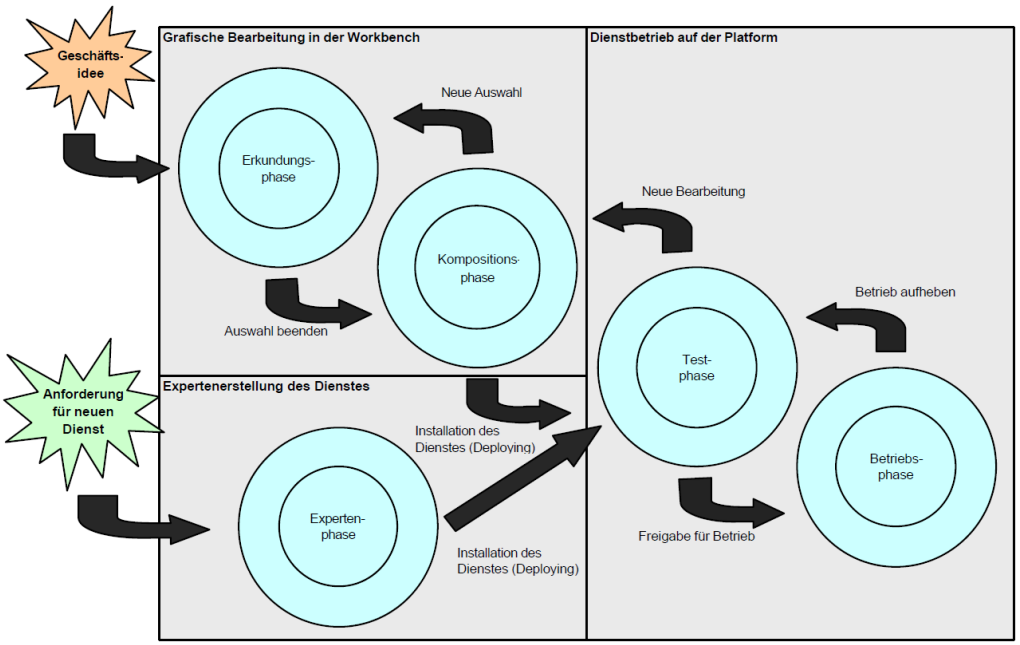

Figure 6.16: The MAMS 4plus1 process model

Chapter 7

Conclusion

The best way to predict the future is to invent it.
(Alan Kay)

7.1 Summary

This thesis reflects on the application and evolution of *Agent-Oriented Software Engineering* in practice. The outcome of this thesis is an agent-oriented methodology, a framework for multi-agent systems and a tool environment for the develoment of agent-based systems. The evolution is driven by the practical experience of projects, which have incorporated multi-agent systems and services as a result. Not only we see iterative and incremental development essential for software development as such, but also for methodology, framework and tools driving the engineering process and making it faster and more efficient.

7.2 Future Work

The world is changing permanently and many ideas from agent-oriented software engineering have found their way into everyday life like *active objects*, *service engineering* and *cloud computing* are only some examples. However, the AOSE workshop at AAMAS 2011 conference pointed out that future AOSE must

> "... find a way out of the increasing fragmentation and fuzziness on software engineering in AOSE..."

At DAI-Labor, the team of *Agent Core Technologies* is currently working in many applicational areas, such as the *home and energy domains*, *electro-*

mobility, *production control*, *cyber-physical systems* and *cloud computing*, in order to customize methodology, framework and tools to concrete problems to be even more efficient when creating solutions in the respective fields. Applications in these areas create new ideas of what is really needed and so there exist expectations to find more results of what can be generalized and provided as building blocks for AOSE.

Index

Glossary

Eclipse
: open source tool development platform, `http://www.eclipse.org/`. 76, 79–82, 86–88, 91, 94

Java
: programming language, `http://www.oracle.com/technetwork/java/index.html`. 19, 22–24, 27–29, 36, 40, 41, 64, 70, 133, 139, 153

JUnit
: open source unit testing framework for Java, `http://junit.sourceforge.net/`. 86

Maven
: Apache project developing a comprehensive build system, `http://maven.apache.org/`. 83

OWL-Lite
: OWL-Lite uses only some of the OWL language features and has some more restrictions on the use of these features than DL and Full, `http://www.w3.org/TR/2004/REC-owl-features-20040210/`. 31

OWL-S
: Semantic markup for web services, upper ontology for describing services, including subontologies for profiles, processes, and groundings. 22, 27, 29, 31, 35–38, 40, 41

Acronyms

AAA
Authentication, Authorisation, and Accounting. 21, 160

AC
Agent Contest. 137

ADEM
Agent-oriented Development Methodology. 116

AI
Artificial Intelligence. ii

AML
Agent Modeling Language. 116

AMS
Agent Management System. 13, 24

AO
Agent-Oriented. ii

AOP
Agent-Oriented Programming. i, 1

AOSE
Agent-Oriented Software Engineering. i, 1, 2, 5, 90, 117

API
Application Programming Interface. 24, 29, 64, 86

AUML
Agent Unified Modeling Language. 117, 118

AWE
Agent World Editor. 90–94, 143

BDI
Belief Desire Intention. 1, 16, 22, 34, 40, 134

BPD
Business Process Diagram. 71

BPEL
Business Process Execution Language. 34, 65–70, 72–78, 101, 106

BPM
Business Process Management. 65, 76

BPMN
Business Process Modeling Notation. 10, 24, 65–68, 70–77, 89, 99–109, 111, 112, 114, 115, 117, 118, 163

CAFnE
Component Agent Framework for domain-Experts. 19

CASA
Component Architecture for Service Agents. 134

CI
Continuous Integration. 7–9

CLDC
Connected Limited Device Configuration. 41

CTT
Concur Task Trees. 161

DAI
Distributed Artificial Intelligence. 1

DC
Dublin Core. 65

DF
Directory Facilitator. 13, 24

EMF
Eclipse Modeling Framework. 70, 81

EMT
Tiger EMF Model Transformation. 70, 77

EPC
Event-driven Process Chain. 65

FIPA
Foundation for Intelligent Physical Agents. 12, 13, 27, 41, 135

FOAF
Friend Of A Friend. 65

GEF
Graphical Editing Framework. 81

GMF
Graphical Modeling Framework. 68, 91, 94

GO-BPMN
Goal-Oriented BPMN. 116

GUI
Graphical User Interface. 6, 113, 136

HSP
Home Service Platform. 159, 163

IAF
INGENIAS Agent Framework. 19

IDE
Integrated Development Environment. 7, 79, 80, 83, 86–88, 133

IDK
INGENIAS Development Kit. 19

IMS
IP Multimedia Subsystem. 164

IOPE
Input parameters, Output parameters, Preconditions and Effects. 36

ISONI
Intelligent Service Oriented Network Infrastructure. 164

J2ME
Java 2 Platform Micro Edition. 41

JADE
Java Agent DEvelopment Framework. 19, 117

JADL
JIAC Action Description Language. 27, 28, 80, 82, 83, 85, 86, 106–110, 112–114, 116, 133, 135, 163

JADL++
JIAC Action Description Language. 22–24, 26–29, 31, 34, 35, 40, 41

JDE
JACK Development Environment. 19, 88

JDT
Java Development Tools. 82, 86

JIAC
Java-based Intelligent Agent Componentware. ii, 21–25, 27–29, 34–42, 67, 75, 79–83, 86, 88, 90–93, 96–98, 100, 104–106, 108, 110, 113, 114, 117, 133–137, 139, 141, 142, 144, 149, 150, 152, 160, 164

JMS
Java Message Service. 41

JMX
Java Management Extensions. 41

JVM
Jave Virtual Machine. 14, 160

LHS
Left Hand Side. 70

MAMS
Multi-Access, Modular-Services. 164

MAS
Multi-Agent System. 2, 14, 19, 22, 23, 35, 75, 90, 91, 93, 94, 97, 117, 133, 135, 137, 140–142, 144

MDA
Model Driven Architecture. 10

MDE
Model Driven Engineering. 10, 11

MIAC
Methodology for Intelligent Agent Componentware. ii, 141, 142, 164

MTS
Message Transport System. 13

NAC
Negative Application Condition. 70

ODSDP
Open Distributed Service Delivery Platform. 164, 165

OO
Object-Orientation. 163

OOP
Object-Oriented Programming. 64

OSGi
Open Services Gateway initiative. 160

OWL
Web Ontology Language. 22, 27, 29, 31, 33–36, 39, 41, 64, 85

PDA
Personal Digital Assistant. 41

PDT
Prometheus Design Tool. 19

PIM
Platform Independent Model. 10

PSM
Platform Specific Model. 10

QoS
Quality of Service. 36, 37, 160

RDF
Resource Description Framework. 65

RFID
Radio-Frequency IDentification. 74

RHS
Right Hand Side. 70

RSD
Rich Service Directory. 69

RUP
Rational Unified Process. 163

SCB
Service Centric Bank. 86, 87, 96, 97

SCM
Software Configuration Management. 5

SCWB
Service Creation WorkBench. 164, 165

SDL
Service Decription Language. 160

SerCHo
Service Centric Home. 67, 74, 159, 162

SOA
Service Oriented Architecture. 66

SPP
Service Provider Platform. 159, 163

STP
SOA Tool Platform. 67, 76

STRIPS
STanford Research Institute Problem Solver. 27

SUMO
Suggested Upper Merged Ontology. 62

SWRL
Semantic Web Rule Language. 39

TDD
Test-Driven Development. 7

TNG
The Next Generation. 136, 139, 141, 150

TUB
Technische Universität Berlin. 67, 142

UI
User Interface. 83

UML
Unified Modeling Language. 10, 83, 116, 118, 163

UPnP
Universal Plug'n'Play. 160

VSDT
Visual Service Design Tool. 67–70, 77, 90, 105, 111, 112, 116, 163

WADE
Workflows and Agents Development Environment. 117

WSDL
Web Service Description Language. 73, 74

WSIG
Web Service Integration Gateway. 13

XML
Extensible Markup Language. 86, 95, 96, 143, 153

XPDL
XML Process Definition Language. 65, 117

XSD
XML Schema Definition. 31, 67

Bibliography

[AEU07] T. Abdelaziz, M. Elammari, and R. Unland. A Framework for the Evaluation of Agent-Oriented Methodologies. In *4th International Conference on Innovations in Information Technology (IIT'07)*, pages 491 – 495, 2007.

[AG98] Sahin Albayrak and Francisco J. Garijo, editors. *Intelligent Agents for Telecommunication Applications*, volume 1437 of *Lecture notes in artificial intelligence*. Springer, 1998.

[age] The agenttool iii project.

[Age05] Agent Oriented Software Pty. Ltd. *JACK Intelligent Agents — Design Tool Manual*, 5.3 edition, June 2005. `http://www.aosgrp.com/documentation/jack/DesignTool\_Manual.pdf`.

[AIS78] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1978.

[Alb92] Sahin Albayrak. *Kooperative Lösung der Aufgabe Auftragsdurchsetzung in der Fertigung durch ein Mehr-Agenten-System auf der Basis des Blackboard-Modells*. PhD thesis, Technische Universität Berlin, 1992.

[Alb98] Sahin Albayrak, editor. *Intelligent Agents for Telecommunications Applications*, volume 36 of *Frontiers in Artificial Intelligence*. IOS Press, 1998.

[Alb99] Sahin Albayrak, editor. *Intelligent Agents for Telecommunication Applications*, volume 1699 of *Lecture Notes in artificial intelligence*. Springer, 1999.

[Apa10] Apache Software Foundation. *About Maven*, Mar 2010.

[Ast03] David Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003.

[AW98] Sahin Albayrak and Dirk Wieczorek. JIAC - an open and scalable agent architecture for telecommunication applications. In S. Albayrak, editor, *Intelligent Agents in Telecommunications Applications - Basics, Tools, Languages and Applications*. IOS Press, Amsterdam, 1998.

[AWV+07] Sahin Albayrak, Stefan Wollny, Nicolas Varone, Andreas Lommatzsch, and Dragan Milosevic. Agent Technology for Personalized Information Filtering: The PIA System. *Scalable Computing: Practice and Experience. Scientific International Journal for Parallel and Distributed Computing*, 8(1):29–40, March 2007.

[BACR08] Birgit Burmeister, M. Arnold, Felicia Copaciu, and Giovanni Rimassa. BDI-Agents for Agile Goal-Oriented Business Processes. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 37–44, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[BCPR03] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE - A White Paper. Technical report, Telecom Italia Lab, 2003.

[Bec02] Kent Beck. *Test Driven Development. By Example.* Addison-Wesley Longman, 2002.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, Genova, Italy, October 2006.

[BG88] Alan H. Bond and Leslie G. Gasser, editors. *Readings in Distributed Artificial Intelligence*. M. Kaufmann, 1988.

[BGG+04] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. TROPOS: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004.

[BGZ04] Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli. *Methodologies and Software Engineering for Agent Systems. The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2004.

[BHS+04] Arthur Barstow, James Hendler, Mark Skall, Jeff Pollock, David Martin, Vincent Marcatte, Deborah L. McGuinness, Hideki Yoshida, and David De Roure. OWL-S: Semantic Markup for Web Services, 2004. `http://www.w3.org/Submission/OWL-S/`.

[BHW07] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley-Blackwell, October 2007.

[Bis07] Judith Bishop. *C# 3.0 Design Patterns*. O'Reilly Media, December 2007.

[BLM09] Michael Burkhardt, Marco Lützenberger, and Nils Masuch. Towards Toolipse 2. Tool Support for the Next Generation Agent Framework. *Computing and Information Systems Journal*, 13(3):21–28, October 2009.

[BMO01] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Revised Papers*, volume 1957 of *LNCS*, pages 91–104. Springer-Verlag, 2001.

[Boa08] JADE Board. JADE Web Service Integration Gateway (WSIG) Guide. Technical report, Telecom Italia, 2008.

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - a FIPA-compliant agent framework. Internal technical report, CSELT, 1999. Part of this report has been also published in Proceedings of PAAM'99, London, April 1999, pp.97-108.

[Bra87] Michael E. Bratman. *Intentions, Plans, and Practical Reason*. Havard University Press, Cambridge, MA, 1987.

[BRHL99] Paolo Busetta, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. JACK — components for intelligent agents in java. Technical report, Agent Oriented Software Pty, Ltd., 1999.

[bug10] bugzilla.org. *The Bugzilla Guide*, Jan 2010.

[Cal04] Touku Tcheumadjeu Louis Calvin. Entwicklung eines werkzeugs zur unterstützung des agent-oriented software engineering (aose) prozesses. Master's thesis, Technische Universität Berlin, 2004.

[CGB08] Giovanni Caire, Danilo Gotta, and Massimo Banzi. WADE: A software platform to develop mission critical applications exploiting agents and workflows. In Michael Berger, Bernard Burg, and Satoshi Nishiyama, editors, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008) — Industry and Applications Track*, pages 29–36. International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org), May 2008.

[Cha05] Autonomous computing, Nov 2005.

[CJB99] B. Chandrasekaran, John R. Josephson, and V. Richard Benjamins. Ontologies: What are they? why do we need them? *IEEE Intelligent Systems and Their Applications*, 14(1):20–26, 1999. Special Issue on Ontologies.

[Cla07] Ian Clatworthy. *Distributed Version Control Systems – Why and How*. Canonical, December 2007.

[com99a] Common Criteria, part 1: Introduction and general model, version 2.1, Aug 1999.

[com99b] Common Criteria, part 2: Security functional requirements, version 2.1, Aug 1999.

[com99c] Common Criteria, part 3: Security assurance requirements, version 2.1, Aug 1999.

[Cop04] James O. Coplien. *Organizational Patterns of Agile Software Development*. Prentice Hall International, Sep 2004.

[Cou08] Cougaar Software Inc. Cougaar IDE Manual. Technical report, Cougaar Software Inc., 2008.

[Cow09] Taylor Cowan. Binding java objects to rdf. *semanticweb.com*, 2009. `http://semanticweb.com/binding-java-objects-to-rdf\_b10682`.

[CSFP04] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion.* O'Reilly Media, June 2004.

[CT05] Radovan Cervenka and Ivan Trencansky. Agent-oriented development methodology (ADEM). Technical report, Whitestein Technologies, February 2005.

[CT07] Radovan Cervenka and Ivan Trencansky. *The Agent Modeling Language — AML. A Comprehensive Approach to Modeling Multi-Agent Systems.* Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser Basel, 2007.

[CW06a] Christopher Cheong, , and Michael Winikoff. Improving Flexibility and Robustness in Agent Interactions: Extending Prometheus with Hermes. In Alessandro Garcia, Ricardo Choren, Carlos Lucena, Paolo Giorgini, Tom Holvoet, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications*, volume 3914 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 2006.

[CW06b] Christopher Cheong and Michael Winikoff. Hermes: Implementing Goal-Oriented Agent Interactions. In J. G. Carbonell and J. Siekmann, editors, *Programming Multi-Agent Systems. Third International Workshop, ProMAS 2005, Utrecht, The Netherlands, July 26, 2005, Revised and Invited Papers*, number 3862 in Lecture Notes in Artificial Intelligence. Springer Berlin/Heidelberg, 2006.

[DAI] DAI-Labor, TU Berlin, Germany. $NeSSi^2$: Network Security Simulator. `http://www.nessi2.de/`.

[DC98] A. Drogoul and A. Collinot. Applying an Agent-Oriented Methodology to the Design of Artificial Organizations: A Case Study in Robotic Soccer. *Autonomous Agents and Multi-Agent Systems*, Volume 1, Number 1, March:113–129, 1998.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Formal semantics and analysis of BPMN process models using Petri nets, 2008.

[DeL06] Scott A. DeLoach. Engineering organization-based multiagent systems. In Alessandro Garcia, Ricardo Choren, Carlos Lucena, Paolo Giorgini, Tom Holvoet, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV*, volume 3914 of *Lecture Notes in Computer Science*, pages 109–125. Springer Berlin/Heidelberg, 2006.

[dKLW98] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In M. Singh, M. Wooldridge, and A. Rao, editors, *Intelligent Agents IV: Agent Theories, Architectures, and Languages*, volume 1365 of *LNAI*. Springer Verlag, 1998.

[DMG07] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. The Signature Series. Addison-Wesley Professional, July 2007.

[DPF+05] Li Ding, Rong Pan, Tim Finin, Anupam Joshi, Yun Peng, and Pranam Kolari. Finding and ranking knowledge on the semantic web. In *Proceedings of the 4th International Semantic Web Conference*, number 3729 in LNCS, pages 156–170. Springer, 2005.

[eet07] Rapid Subversion Adoption Validates Enterprise Readiness and Challenges Traditional Software Configuration Management Leaders. *EETimes*, May 17, 2007.

[EHKA07] Holger Endert, Benjamin Hirsch, Tobias Küster, and Sahin Albayrak. Towards a mapping from BPMN to agents. In Jingshan Huang, Ryszard Kowalczyk, Zakaria Maamar, David Martin, Ingo Müller, Suzette Stoutenburg, and Katia P. Sycara, editors, *Service-Oriented Computing: Agents, Semantics, and Engineering*, volume 4505 of *LNCS*, pages 92–106. Springer Berlin / Heidelberg, 2007.

[EKHA07] Holger Endert, Tobias Küster, Benjamin Hirsch, and Sahin Albayrak. Mapping BPMN to agents: An analysis. In Matteo Baldoni, Cristina Baroglio, and Viviana Mascardi, editors,

Agents, Web-Services, and Ontologies Integrated Methodologies, pages 43–58, 2007.

[FB07] Dominic Greenwood Fabio Bellifemine, Giovanni Caire. *Developing Multi-Agent Systems with JADE*, chapter The JADE Web Services Integration Gateway, pages 181–205. John Wiley & Sons, Ltd, 2007.

[FBK+01] Stefan Fricke, Karsten Bsufka, Jan Keiser, Torge Schmidt, Ralf Sesseler, and Sahin Albayrak. A Toolkit for the Realization of Agent-based Telematic Services and Telecommunication Applications. *Communications of the ACM*, 44(4):43–48, April 2001.

[FBLA06] Sebastian Feuerstack, Marco Blumendorf, Grzegorz Lehmann, and Sahin Albayrak. Seamless home services. In *Developing Ambient Intelligence (AmID'06)*, 2006.

[Feu08] Sebastian Feuerstack. *Eine Methodik zur nutzerzentrierten und modellbasierten Entwicklung von interaktiven Anwendungen*. PhD thesis, TU Berlin, 2008.

[FKH02] Stefan Fricke, Jan Keiser, and Axel Hessler. *Demo-Storyboard for the JIAC IV Agent Development Environment*. AAMAS Demonstration Session, Bologna, Italy, July 15-19 2002.

[FN71] R. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fog03] Karl Fogel. *Open Source Development with CVS*. Paraglyph Press, July 2003.

[Fou02a] Foundation for Intelligent Physical Agents. FIPA Agent Communication Language Specifications, 2002.

[Fou02b] Foundation for Intelligent Physical Agents. Interaction Protocol Specifications, 2002.

[Fou04a] Apache Software Foundation. Apache Continuum. `http://continuum.apache.org/`, 2004.

[Fou04b] Foundation for Intelligent Physical Agents. FIPA Agent Management Specification, March 2004.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Object-Oriented Software Engineering. Addison-Wesley Longman, Nov 1996.

[Fow00] Martin Fowler. Continuous integration. `http://martinfowler.com/articles/originalContinuousIntegration.html`, 2000.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, Nov 2002.

[Fow06] Martin Fowler. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, May 2006.

[FSDM07] Behrend Freese, Horst Stein, S. Dutkowski, and Th. Magedanz. Multi-access modular-services framework — supporting smes with an innovative service creation toolkit based on integrated sdp/ims infrastructure. In *ICIN 2007 Bordeaux*, 2007.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GH05] Pamela Garretson and Paul Harmon. How Boeing A&T manages business processes. Whitepaper, BPTrends, November 2005.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[GKP04] Tim Geissler and Olaf Kroll-Peters. Applying security standards to multi agent systems. In *Proceedings of the First International Workshop on Safety and Security in Multiagent Systems, Sasamas'04, AAMAS'04*, 2004.

[GMGFF09] Iván García-Magariño, Celia Gutiérrez, and Rubén Fuentes-Fernández. The ingenias development kit: a practical application for crisis-management. In *The 10th International Work conference on Artificial Neuronal Networks (IWANN2009)*, 2009.

[GMGSA08] Ivan García-Magariño, Jorge J. Gómez-Sanz, and José R. Pérez Agüera. A Complete-Computerised Delphi Process with a Multi-agent System. In *Proceedings of the Sixth International Workshop on Programming Multi-Agent Systems, Estoril, Portugal*, pages 187–202, 2008.

[GSFF03] Jorge J. Gómez-Sanz and Rubén Fuentes-Fernández. *Multi-Agent Systems and Applications III*, chapter Agent Oriented Software Engineering with INGENIAS, pages 394–403. Springer Berlin/Heidelberg, January 2003.

[GSP06] Jorge J. Gómez-Sanz and Juan Pavón. Defining Coordination in Multi-Agent Systems within an Agent Oriented Software Engineering Methodology. In *Proceedings of the 2006 ACM symposium on Applied computing, Dijon, France*, pages 424–428. ACM, April 2006.

[GTP07] Pau Giner, Victoria Torres, and Vicente Pelechano. Bridging the gap between BPMN and WS-BPEL. M2M transformations in practice. In *Proc. of the 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007)*, Como, Italy, July 2007.

[HBvdHM99] Koen V. Hindriks, Frank S. De Boer, Wiebe van der Hoek, and John-Jules Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[HFKP08] Benjamin Hirsch, Stefan Fricke, and Olaf Kroll-Peters. Agent programming in practise — experiences with the JIAC IV agent framework. In *Proceedings of AT2AI 2008 — Agent Theory to Agent Implementation*, 2008.

[HHK08] Axel Heßler, Benjamin Hirsch, and Jan Keiser. JIAC IV in Multi-Agent Programming Contest 2007. In M. Dastani, A. El Fallah Segrouchni, A. Ricci, and M. Winikoff, editors, *ProMAS 2007 Post-Proceedings*, volume 4908 of *LNAI*, pages 262–266. Springer Berlin / Heidelberg, 2008.

[HHK10] Axel Heßler, Benjamin Hirsch, and Tobias Küster. Herding cows with JIAC V. *Annals of Mathematics and Artificial Intelligence*, pages 1–15, 2010.

[HHKA11] Axel Heßler, Benjamin Hirsch, Tobias Küster, and Sahin Albayrak. Agentstore — a pragmatic approach to agent reuse. In *12th International Workshop on Agent-Oriented Software Engineering (AOSE 2011)*, 2011.

[HKBA10] Benjamin Hirsch, Thomas Konnerth, Michael Burkhardt, and Sahin Albayrak. Programming service oriented agents. In Monique Calisti, Frank P. Dignum, Ryszard Kowalczyk, Frank Leymann, and Rainer Unland, editors, *Service-Oriented Architecture and (Multi-)Agent Systems Technology*, number 10021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[HKF+04] Axel Heßler, Jan Keiser, Sebastian Feuerstack, Karsten Bsufka, and Stefan Fricke. *Demo-Storyboard: An Agent-based Framework supporting Rapid Application Development for Telecommunication Applications.* AAMAS Demonstration Session, New York, USA, July 19-23 2004.

[HKH09] Benjamin Hirsch, Thomas Konnerth, and Axel Heßler. Merging Agents and Services — the JIAC Agent Platform. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 159–185. Springer, 2009.

[HKHA06] Benjamin Hirsch, Thomas Konnerth, Axel Heßler, and Sahin Albayrak. A serviceware framework for designing ambient services. In Antonio Maña and Volkmar Lotz, editors, *Developing Ambient Intelligence (AmID'06)*, pages 124–136. Springer Paris, 2006.

[HKK+09] Axel Heßler, Jan Keiser, Tobias Küster, Marcel Patzlaff, Alexander Thiele, and Erdene-Ochir Tuguldur. Herding agents - jiac tng in multi-agent programming contest 2008. In Koen V. Hindriks, Alexander Pokahr, and Sebastian Sardina, editors, *Programming Multi-Agent Systems. 6th International Workshop, ProMAS 2008, Estoril, Portugal, May 13, 2008. Revised Invited and Selected Papers*, volume 5442 of *Lecture Notes in Artificial Intelligence*, pages 228–232. Springer, 2009.

[HKN+09] Axel Heßler, Tobias Küster, Oliver Niemann, Aldin Sljivar, and Amir Matallaoui. Cows and Fences: JIAC V - AC'09 Team Description. In Jürgen Dix, Michael Fisher, and Peter Novák, editors, *Proceedings of the 10th International Workshop on Computational Logic in Multi-Agent Systems 2009*, volume IfI-09-08 of *IfI Technical Report Series*. Clausthal University of Technology, 2009.

[HMF09] Christian Hahn, Cristián Madrigal Mora, and Klaus Fischer. A platform-independent metamodel for multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, 4 2009.

[Hol05] Steve Holzner. *Ant: The Definitive Guide*. O'Reilly Media, 2005.

[HPGP09] Jens Hartmann, Raul Palma, and Asuncion Gomez-Perez. Ontology repositories. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks Information System, pages 551–571. Springer Berlin Heidelberg, 2009. 10.1007/978-3-540-92673-3_25.

[HTW04] A. Helsinger, M. Thome, and T. Wright. Cougaar: a scalable, distributed multi-agent architecture. In *IEEE International Conference on Systems, Man and Cybernetics*, 2004.

[HW05] Aaron Helsinger and Todd Wright. Cougaar: A Robust Configurable Multi Agent Platform. In *IEEE Aerospace Conference*, 2005.

[JGC+05] Michael C. Jaeger, Rojec-Goldmann Gregor, Liebetruth Christoph, Mühl Gero, and Geihs Kurt. Ranked matching for service descriptions using OWL-S. In *Kommunikation in Verteilten Systemen (KiVS) 2005*, pages 91–102. Springer, February 2005.

[JTPW06] Gaya Jayatilleke, John Thangarajah, Lin Padgham, and Michael Winikoff. Component Agent Framework for domain-Experts (CAFnE) Toolkit. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan*, pages 1465–1466. ACM, May 2006.

[Ken98] Elizabeth A. Kendall. Agent roles and role models: New abstractions for intelligent agent systems analysis and design, 1998.

[KFS06] Matthias Klusch, Benedikt Fries, and Katia Sycara. Automated semantic web service discovery with OWLS-MX. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, *AAMAS*, pages 915–922. ACM, 2006.

[KGM+11] Jan Keiser, Juri Glass, Nils Masuch, Marco Lützenberger, and Sahin Albayrak. A distributed multi-operator W2V2G management approach. In *Proceedings of the 2nd IEEE International Conference on Smart Grid Communications, Brussels, Belgium*, pages 291–296, October 2011.

[KH04] Jana Koehler and Rainer Hauser. Untangling unstructured cyclic flows - a solution based on continuations. In R. Meesman and Z. Tari, editors, *CooplIS/DOA/ODBASE 2004*, volume 3290 of *LNCS*, pages 121–138. IBM Zurich Research Laboratory, Springer-Verlag, 2004.

[KH08] Tobias Küster and Axel Heßler. Towards transformations from BPMN to heterogeneous systems. In Massima Mecella and Jian Yang, editors, *BPM2008 Workshop Proceedings*, 2008.

[KHA06] Thomas Konnerth, Benjamin Hirsch, and Sahin Albayrak. JADL — an agent description language for smart agents. In Mateo Baldoni and Ulle Endriss, editors, *Declarative Agent Languages and Technologies IV*, volume 4327 of *LNCS*, pages 141–155. Springer Berlin / Heidelberg, 2006.

[KHH09] Tobias Küster, Axel Heßler, and Benjamin Hirsch. Modelling and transforming BPMN diagrams with the visual service design tool. In *Demo Track at BPM2009*, 2009.

[KLHH10] Tobias Küster, Marco Lützenberger, Axel Heßler, and Benjamin Hirsch. Integrating process modelling into multi-agent system engineering. In Michael Huhns, Ryszard Kowalczyk, Zakaria Maamar, Rainer Unland, and Bao Vo, editors, *Proceedings of the 5th Workshop of Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE) 2010*, 2010. to appear.

[KLL09] Ryan K.L. Ko, Stephen S.G. Lee, and Eng Wah Lee. Business process management (BPM) standards: a survey. *Business Process Management Journal*, 15(5):744 – 791, 2009.

[KtHB00] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 431–445, London, UK, 2000. Springer-Verlag.

[Küs07] Tobias Küster. Development of a visual service design tool providing a mapping from BPMN to JIAC. Diploma thesis, Technische Universität Berlin, April 2007.

[LAH+11] Marco Lützenberger, Sebastian Ahrndt, Benjamin Hirsch, Nils Masuch, Axel Heßler, and Sahin Albayrak. Strategic behaviour in a living environment. In *Proceedings of the Winter Simulation Conference, Phoenix, AZ, USA*, December 2011. To appear.

[LB03] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, June:47–56, June 2003.

[LK05] Rong Liu and Akhil Kumar. An analysis and taxonomy of unstructured workflows. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649, pages 268–284, 2005.

[LKHH09] Marco Lützenberger, Tobias Küster, Axel Heßler, and Benjamin Hirsch. Unifying JIAC agent development with AWE. In *Proceedings of the Seventh German Conference on Multiagent System Technologies, Hamburg, Germany*. Springer, 2009.

[LMH+11] Marco Lützenberger, Nils Masuch, Benjamin Hirsch, Sebastian Ahrndt, Axel Heßler, and Sahin Albayrak. The BDI driver in a service city. In Kagan Tumer, Pinar Yolum, Liz Sonenberg, and Peter Stone, editors, *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan*, pages 1257–1258. International Foundation for Autonomous Agents and Multiagent Systems, May 2011.

[LMSW05] Michael Luck, Peter McBurney, Onn Shehory, and Steve Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005.

[Lüt09] Marco Lützenberger. Development of a visual notation and editor for unifying the application engineering within the JIAC framework family. Diploma thesis, Technische Universität Berlin, March 2009.

[LW03] Michael Luck and Steve Willmott. Agents for commercial applications. Technical report, AgentLink, 2003.

[Man08] MantisBT Project. *Mantis Bug Tracker Administration Guide*, 2008.

[Mec04] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, 2004.

[Men09] Jan Mendling. *Metrics for Process Models - Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*, volume 6 of *Lecture Notes in Business Infromation Processing*. Springer, 2009.

[Met06] Steven John Metsker. *Design Patterns in Java*. Addison-Wesley Longman, 2nd edition, April 2006.

[MG07] Haralambos Mouratidis and Paolo Giorgini. Secure tropos: A security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309, 2007.

[MHC00] Richard Monson-Haefel and David A. Chappell. *Java Message Service*. O'Reilly, 2000.

[MLZ05] Jan Mendling, Kristian Bisgaard Lassen, and Uwe Zdun. Transformation strategies between blockoriented and graph-oriented process modelling languages, 2005.

[MM03] Joaquin Miller and Jishnu Mukerji. MDA guide version 1.0.1, 2003. Document Number: omg/2003-06-01.

[MMRS55] John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon. A proposal for the dartmouth summer research project on artificial intelligence, August 31 1955.

[MPP08] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 9–16, New York, NY, USA, 2008. ACM.

[Müc08] Amrei Mücke. *Service Centric Bank*. PhD thesis, Technische Universität Berlin, 2008.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language. W3C Recommendation, 2004. `http://www.w3.org/TR/owl-features/`.

[MWK+05] Cynthia Matuszek, Michael Witbrock, Robert C. Kahlert, John Cabral, Dave Schneider, Purvesh Shah, and Doug Lenat. Searching for common sense: Populating cyc from the web. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*. AAAI, July 2005.

[NGM08] Natalya F. Noy, Nicholas Griffith, and Mark A. Musen. Collecting community-based mappings in an ontology repository. In *7th International Semantic Web Conference (ISWC 2008). Conference Proceeding*. Springer, 2008.

[NN99] H. Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction, revised Edition*. Wiley, 1999.

[NN05] Madjid Nakhjiri and Mahsa Nakhjiri. *AAA and Network Security for Mobile Access: Radius, Diameter, EAP, PKI and IP Mobility*. Wiley, 2005.

[NPM04] Glen Newton, Jeff Pollock, and Deborah L. McGuinness. Semantic web rule language (swrl), 2004. `http://www.w3.org/Submission/2004/03/`.

[Oas07] Oasis Committee. Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Technical report, Oasis, 2007.

[Obj05] Object Management Group. Software Process Engineering Metamodel (SPEM) Specification. Version 1.1. Object Management Group, Inc., January 2005.

[Obj06] Object Management Group. Business Process Modeling Notation (BPMN) Specification. Final Adopted Specification dtc/06-02-01, OMG, 2006. `http://www.bpmn.org/Documents/OMGFinalAdoptedBPMN1-0Spec06-02-01.pdf`.

[ODBtH06] Chun Ouyang, Marlon Dumas, Stephan Breutel, and Arthur H. M. ter Hofstede. Translating standard process models to BPEL. In Eric Dubois and Klaus Pohl, editors, *CAiSE*, volume 4001 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 2006.

[OJ96] Greg M. P. O'Hare and Nick Jennings, editors. *Foundations of distributed artificial intelligence*. Wiley, 1996.

[OMG08] OMG. Business Process Modeling Notation (BPMN). Specification Version 1.1, Object Management Group, January 2008. formal/2008-01-17.

[OMG09] OMG. Business Process Modeling Notation (BPMN). Specification Version 1.2, Object Management Group, January 2009. formal/2009-01-03.

[OMG11] OMG. Business Process Model and Notation. Specification Version 2.0, Object Management Group, January 2011. formal/2011-01-03.

[OPB00] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, 2000.

[Pat99] Fabio Paterno. *Model-based Design and Evaluation of Interactive Applications*. Springer, 1999.

[PBFO04] H. Van Dyke Parunak, Sven Brueckner, Mitch Fleischer, and James Odell. A design taxonomy of multi-agent interactions. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *Agent-Oriented Software Engineering IV. 4th International Workshop, AOSE 2003.*, volume 2935 of *LNCS*, 2004.

[Per02] J. Steven Perry. *Java Management Extensions*. O'Reilly, 2002.

[PNL02] Adam Pease, Ian Niles, and John Li. The suggested upper merged ontology: A large ontology for the semantic web and its applications. Technical Report WS-02-11, AAAI, 2002.

[PT09] Marcel Patzlaff and Erdene-Ochir Tuguldur. Micro-JIAC 2.0 - The Agent Framework for Constrained Devices and Beyond. Technical Report TUB-DAI 07/09-01, DAI-Labor, Technische Universität Berlin, July 2009. `http://www.dai-labor.de/fileadmin/files/publications/microjiac\_20\_2009\_07\_02.pdf`.

[PW02] Lin Padgham and Michael Winikoff. Prometheus: A methodology for developing intelligent agents. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering III. Revised Papers and Invited Contributions of the Third International Workshop (AOSE 2002)*, volume 2585 of *0558 Lecture Notes in Computer Science*. Springer, 2002.

[PW04] Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. Wiley Series in Agent Technology. Wiley, 2004.

[Rao96] Anand. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Rudy van Hoe, editor, *Agents Breaking Away, 7^{th} European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96,*, volume 1038 of *Lecture Notes in Computer Science*, pages 42—55, Eindhoven, The Netherlands, January 1996. Springer Verlag.

[RCF+05] Andreas Rieger, Richard Cissée, Sebastian Feuerstack, Jens Wohltorf, and Sahin Albayrak. An agent-based architecture for ubiquitous multimodal user interfaces. In *Proceedings of the 2005 International Conference on Active Media Technology*, pages 119–124. IEEE, 2005.

[RCK05] Giovanni Rimassa, Monique Calisti, and Martin E. Kernland. Living Systems® Technology Suite. In Rainer Unland, Matthias Klusch, and Monique Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser Basel, 2005.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Second International Conference (KR'91)*, pages 473–484. Morgan Kaufmann, San Mateo, CA, 1991.

[Rie00] Dirk Riehle. *Framework Design - A Role Modeling Approach*. PhD thesis, Institute of Technology Zurich, 2000.

[RM06] Jan Recker and Jan Mendling. On the translation between BPMN and BPEL: Conceptual mismatch between process modeling languages. In T. Latour and M. Petit, editors, *In: T. Latour, M. Petit, eds.: CAiSE 2006 Workshop Proceedings - Eleventh International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2006), June 5 - 6, 2006, Luxembourg, pages 521-532.*, pages 521–532, 2006.

[RN03] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[RPU+07] Ivo Raedts, Marija Petkovic, Yaroslav S. Usenko, Jan Martijn E. M. van der Werf, Jan Friso Groote, and Lou J. Somers. Transformation of BPMN models for behaviour analysis. In Juan Carlos Augusto, Joseph Barjis, and Ulrich Ultes-Nitsche, editors, *MSVVEIS*, pages 126–137. INSTICC PRESS, 2007.

[SA02] Ralf Sesseler and Sahin Albayrak. JIAC IV - an open, scalable agent architecture for telecommunications applications. In *Proceedings of the First International NAISO Congress on Autonomous Intelligent Systems (ICAIS 2002)*. ICSC Interdisciplinary Research, 2002.

[Sch06] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, February:25–31, 2006.

[Sea69] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[Ses02] Ralf Sesseler. *Eine modulare Architektur für dienstbasierte Interaktionen zwischen Agenten*. PhD thesis, Technische Universität Berlin, 2002.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[SO00] Wasim Sadiq and Maria E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000.

[THHA08] Erdene-Ochir Tuguldur, Axel Heßler, Benjamin Hirsch, and Sahin Albayrak. Toolipse: An IDE for development of JIAC applications. In *Proceedings of PROMAS08: Programming Multi-Agent Systems*, 2008.

[Tho01] ThoughtWorks. CruiseControl. `http://cruisecontrol.sourceforge.net`, 2001.

[TK10] Jakob Tonn and Silvan Kaiser. ASGARD - a graphical monitoring tool for distributed agent infrastructures. In *Proceedings of 8th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2010)*, Salamanca, Spain, 2010.

[TKK+09] Alexander Thiele, Thomas Konnerth, Silvan Kaiser, Jan Keiser, and Benjamin Hirsch. Applying JIAC V to real world problems — the MAMS case. In *Proceedings of the German conference on Multi-Agent System Technologies*, pages 268 – 277. Springer, 2009.

[TKKH09] Alexander Thiele, Silvan Kaiser, Thomas Konnerth, and Benjamin Hirsch. MAMS service framework. In Ryszard Kowalczyk, Quoc Bao Vo, Zakaria Maamar, and Michael Huhns, editors, *Service-Oriented Computing: Agents , Semantics, and Engineering: AAMAS 2009 International Workshop, SOCASE 2009, Budapest, Hungary, May 11, 2009, Revised Selected Papers*. Springer, 2009.

[TNNM10] Tania Tudorache, Natalya F. Noy, Csongor Nyulas, and Mark A. Musen. Use cases for the interoperation between an ontology repository and an ontology editor. In *Proceedings of the Workshop on Semantic Repositories for the Web*, 2010.

[TP08] Erdene-Ochir Tuguldur and Marcel Patzlaff. Collecting Gold: MicroJIAC Agents in MULTI-AGENT PROGRAMMING

CONTEST. In M. Dastani, A. El Fallah Segrouchni, A. Ricci, and M. Winikoff, editors, *ProMAS 2007 Post-Proceedings*, volume 4908 of *LNAI*, pages 257–261. Springer Berlin / Heidelberg, 2008.

[tra08] *The Trac User and Administration Guide*, Oct 2008.

[UKC05] Rainer Unland, Matthias Klusch, and Monique Calisti, editors. *Software Agent-Based Applications, Platforms and Develoment Kits*. Whitestein Series in Software Agent Technologies. Birkhauser Verlag, 2005.

[vdAL08] Wil M. P. van der Aalst and Kristian Bisgaard Lassen. Translating unstructured workflow processes to readable BPEL: Theory and implementation. *Inf. Softw. Technol.*, 50(3):131–159, 2008.

[vdS02] William van der Sterren. *AI Game Programming Wisdom*, chapter 5.3 Squad Tactics:Team AI and Emergent Maneuvers, pages 233–246. Charles River Media, 2002.

[W3C04] W3C. XML schema, 2004.

[WBA10] Florian Weingarten, Marco Blumendorf, and Sahin Albayrak. Towards multimodal interaction in smart home environments: the home operating system. In *Proceedings of the 8th ACM Conference on Designing Interactive Systems (DIS'10)*, 2010.

[WCR05] Jens Wohltorf, Richard Cissée, and Andreas Rieger. Berlintainment: An agent-based context-aware entertainment planning system. *IEEE Communications Magazine*, 43(6):102–109, June 2005.

[WDB+01] S. Wilmott, J. Dale, B. Burg, P. Charlton, and P O'Brien. Agentcities: A Worldwide Open Agent Network. *AgentLink News*, Issue 8, November 2001.

[WHF09] Stefan Warwas, Christian Hahn, and Klaus Fischer. A visual development environment for JADE. In Decker; Sichman; Sierra; Castelfranchi, editor, *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 1349–1350, 2009.

[Whi04a] Stephen A. White. Introduction to BPMN. Technical report, IBM Corporation, May 2004. http://bpmn.org/Documents/Introduction%20to%20BPMN.pdf.

[Whi04b] Whitestein Technologies. *Agent Modeling Language (AML). Language Specification. Version 0.9*, 2004.

[Whi05] Stephen A. White. Using BPMN to model a BPEL process. Technical report, IBM Corporation, March 2005. http://www.bpmn.org/Documents/Mapping%20BPMN%20to%20BPEL%20Example.pdf.

[Whi09] Whitestein Technologies. Cost Assessment of Risk-sensitive Banking Processes with the Living Systems Process Suite, 2009.

[Win05] Michael Winikoff. *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter Jack Intelligent Agents: An Industrial Strength Platform, pages 175–193. Springer, 2005.

[WJ05] Gerhard Weiß and Ralf Jakob. *Agentenorientierte Softwareentwicklung — Methoden und Tools*. Xpert.press. Springer Berlin/Heidelberg, 2005.

[WJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.

[Woo00] Mark F. Wood. Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems. Master's thesis, Air University, March 2000.

[Wor] Workflow Management Coalition. XPDL: XML Process Definition Language.

[WP04] Michael Winikoff and Lin Padgham. *Developing Intelligent Agent Systems: A Practical Guide*. Wiley and Sons, 2004.

[Yu95] Eric Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1995.

[ZJW03] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003.

[ZJW05] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. *Multiagent systems as computational organisations: the Gaia methodology*, chapter VI, pages 136–171. Idea Group Publishing, 2005.

[ZWA07] Paul Zernicke, Carsten Wirth, and Sahin Albayrak. Towards collaborative user-centric healthcare services. In *International Conference on Information Society (i-Society 2007)*, 2007.