Ingrid Oliveira de Nunes

A Domain Engineering Process for Developing Multi-agent Systems Product Lines

MASTER DISSERTATION

DEPARTMENT OF INFORMATICS

Postgraduate Program in Informatics

Rio de Janeiro

March 2009

Pontifícia Universidade Católica
do Rio de Janeiro

Ingrid Oliveira de Nunes

A Domain Engineering Process for Developing Multi-agent Systems Product Lines

Master Dissertation

Dissertation presented to the Postgraduate Program in Informatics of the Department of Informatics, PUC–Rio, as partial fulfillment of the requirements for the Master Degree in Informatics.

Supervisor : Prof. Carlos José Pereira de Lucena
Co-supervisor: Prof. Uirá Kulesza

Rio de Janeiro
March, 2009

Pontifícia Universidade Católica
do Rio de Janeiro

Ingrid Oliveira de Nunes

A Domain Engineering Process for Developing Multi-agent Systems Product Lines

Dissertation presented to the Postgraduate Program in Informatics of Department of Informatics, PUC–Rio, as partial fulfillment of the requirements for the Master Degree in Informatics.

Prof. Carlos José Pereira de Lucena
Supervisor
Departmento de Informática — PUC–Rio

Prof. Uirá Kulesza
Co-supervisor
Departamento de Informática e Matemática Aplicada – UFRN

Prof. Ricardo Choren Noya
Seção de Engenharia de Computação – IME

Prof. Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Prof. José Eugenio Leal
Coordinator of the Centro Técnico Científico — PUC–Rio

Rio de Janeiro— March 31, 2009

Ingrid Oliveira de Nunes

She has completed her undergraduate course at the Federal University of Rio Grande do Sul (Porto Alegre, Brazil), receiving the degree of Bachelor in Computer Science in 2006. She worked as a software developer at the e-Core Desenvolvimento de Software company from 2005 to 2007. She is currently a Ph.D. student and researcher at the Pontifical Catholic University of Rio de Janeiro, and integrates the Software Engineering Laboratory (LES). Her focus is Software Engineering, and her main research interests are multi-agent systems and software product lines.

Ficha Catalográfica

Nunes, Ingrid Oliveira de

A Domain Engineering Process for Developing Multi-agent Systems Product Lines / Ingrid Oliveira de Nunes; orientador: Carlos José Pereira de Lucena; co–orientador: Uirá Kulesza. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2009.

138 f: ; 29,7 cm

1. Dissertação (mestrado) - Pontifical Catholic University of Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Multi-agent Systems Product Lines. 3. Domain Engineering. 4. Software Process. 5. Software Product Lines. 6. Multi-agent Systems. 7. Software Engineering. I. Lucena, Carlos José Pereira de Lucena. II. Kulesza, Uirá. III. Pontifical Catholic University of Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

To my parents, Daltro and Suzana.
To my brothers, Gustavo and Matthias.
I love you all.

Acknowledgments

I would like to thank all of the people that have helped me in the preparation of this dissertation.

First and foremost, I would like to extend my sincere thanks to my supervisors Professor Carlos Lucena and Professor Uirá Kulesza. I would like to express my gratitude to Professor Lucena for his invaluable support and guidance. His guiding advices lighted up my way to go. Thank you for believing in me and giving me several opportunities. I would also like to thank Professor Uirá, who has provided tremendous help over this period with editing, research assistance, feedback and encouragement. Both of them are insightful professors and are people I am thankful to have been able to work with.

I would also like to thank the other members of my examining committee, Professor Ricardo Choren and Professor Simone Barbosa, for taking the time of reading my dissertation and giving me very constructive feedbacks.

My sincere appreciation is then extended to Professor Michael Luck, who visited our University last year. I am thankful for his valuable suggestions and useful discussions. Special thanks to Leandro Daflon for his SPEM lessons and Professor Viviane Torres da Silva, who gave me valuable feedback even though she recently came from Spain.

I would like to thank all my lab members for their help, support and friendship. A special thank you for Camila Nunes, who has been working with me since I started my master and has been a great friend, and Elder Cirilo, who has discussed great ideas with me and has also been a wonderful friend. Vera Menezes, thank you for helping in so many things during my master, her friendship and our endless conversations.

My thanks and gratitude goes to my family and friends, who, even far away, always believed me and encouraged me. Every time that I go to Porto Alegre, I see that truly feelings do not change.

I am especially indebted to my parents, Daltro Nunes and Suzana Nunes, for their love and support throughout my studies and work. Without their encouragement, assistance, patience, generosity and support, this dissertation would not have been able to be completed.

Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). Graduate scholarship as well as the research assistantship have provided personal financial support for day to day living while pursuing the master degree. These supports are duly acknowledged.

Abstract

Nunes, Ingrid Oliveira de; Lucena, Carlos José Pereira de Lucena; Kulesza, Uirá. **A Domain Engineering Process for Developing Multi-agent Systems Product Lines**. Rio de Janeiro, 2009. 138p. Master Dissertation — Department of Informática, Pontifical Catholic University of Rio de Janeiro.

Multi-agent System Product Lines (MAS-PLs) have emerged to integrate two promising trends of software engineering: Software Product Lines (SPLs) and Agent-oriented Software Engineering. The former came to promote reduced time-to-market, lower development costs and higher quality on the development of applications that share common and variable features. These applications are derived in a systematic way, based on the exploitation of application commonalities and large-scale reuse. The later is a new software engineering paradigm that addresses the development of Multi-agent Systems (MASs), which are composed by autonomous and pro-active entities, named agents. This paradigm aims at helping on the development of complex and distributed systems, characterized by a large number of parts that have many interactions. The main goal of MAS-PLs is to incorporate the benefits of both software engineering disciplines and to help the industrial exploitation of agent technology. Only recent research work aimed at integrating SPL and MAS. In this context, this work presents a domain engineering process for developing MAS-PLs. We define the process activities with their tasks and work products, in order to provide a process that addresses modeling agent variability and agent features traceability. Our approach is the result of an investigation of how current SPL, MAS and MAS-PL approaches can model MAS-PLs. Based on this experience, we propose our process, which includes some techniques and notations of some of these approaches: PLUS method, PASSI methodology and MAS-ML modeling language. In particular, the scenario we are currently exploring is the incorporation of autonomous and pro-active behavior into existing web systems. The main idea is to introduce software agents into existing web applications in order to (semi)automate tasks, such as providing autonomous recommendation of products and information to users.

Keywords

Multi-agent Systems Product Lines. Domain Engineering. Software Process. Software Product Lines. Multi-agent Systems. Software Engineering.

Resumo

Nunes, Ingrid Oliveira de; Lucena, Carlos José Pereira de Lucena; Kulesza, Uirá. **Um Processo para Engenharia de Domínio para o Desenvolvimento de Linhas de Produto de Sistemas Multi-agentes**. Rio de Janeiro, 2009. 138p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Desde a introdução de agentes no início dos anos 70, sistemas de software evoluíram de sua natureza isolada e puramente reativa para serem abertos, autônomos, altamente interativos e sensíveis ao contexto. Sistemas Multi-agentes (SMAs), os quais se baseiam na abstração de agentes, surgiram como uma importante abordagem para sistemas complexos e distribuídos com estas características. Agentes são entidades pró-ativas, reativas, autônomas e que exibem características sociais. A propriedade de autonomia refere-se à habilidade de um agente de agir sem a intervenção de humanos e outros sistemas, dado que um agente autônomo tem controle tanto sobre seu estado interno como seu comportamento.

O advento de abordagens computacionais autonômicas, pervasivas e inspiradas na biologia, nas quais muitas propriedades de agentes tornam-se essenciais, conduziu a uma adoção crescente de abordagens multi-agentes para a construção de sistemas distribuídos. Conseqüentemente, avanços na área da Engenharia de Software Orientada a Agentes (ESOA) têm sido propostos através de novas técnicas, tais como metodologias, linguagens de modelagem, processos, e estratégias de implementação direcionadas a SMAs. Entretanto, mesmo que muitas metodologias da ESOA tenham sido introduzidas na literatura, metodologias de SMAs existentes carecem de suporte apropriado de um processo sistemático que guie o desenvolvimento de aplicações multi-agentes e que tirem proveito do potencial reuso oferecido pelas Linhas de Produto de Software (LPSs) e customização em massa em larga escala.

Metodologias de SMA combinadas com LPSs e customização em massa criam novos problemas os quais potencializam a necessidade de novos modelos e processos de desenvolvimento. Assim, neste trabalho, nós oferecemos as seguintes contribuições: (i) nós propomos um novo processo de engenharia de domínio para o desenvolvimento de Linhas de Produto de Sistemas Multi-agentes (LP-SMAs); (ii) nós estendemos abordagens de modelagem existentes para dar suporte à análise, projeto e realização de linhas de produto baseadas em SMAs através da introdução de modelos novos e modificados, bem como provemos técnicas para modelar e rastrear variabilidades de agentes; e (iii) nós aplicamos o processo sistemático em estudos de caso específicos para ilustrar e avaliar a abordagem. Nossa abordagem modela *features* de agentes de

forma independente, conseqüentemente facilitando a incorporação de agentes em sistemas existentes, os quais foram projetados com outros paradigmas (e.g. orientação-a-objetos). A relevância deste trabalho é que ele melhora o entendimento de como LP-SMAs são desenvolvidas e demonstra um processo sistemático que envolve análise, projeto e técnicas de implementação para dar suporte à sua construção.

LPSs estão se tornando uma técnica essencial de reuso que tem sido aplicada na construção de sistemas de software produzidos em massa. Uma LPS é uma família de produtos de software relacionados, construídos a partir de um conjunto comum de componentes de software, onde cada componente tipicamente implementa uma *feature* distinta. LPSs exploram e beneficiam-se de *features* comuns e variáveis de um conjunto de sistemas e conduzem ao desenvolvimento de arquiteturas flexíveis que suportam a derivação de sistemas customizados. Benefícios das LPSs incluem um tempo médio mais baixo para criar e fazer o *deploy* de um produto; um número médio de defeitos mais baixo por produto (i.e. maior qualidade nos produtos); menor esforço para fazer o *deploy* e manter um produto; menor *time-to-market* para novos produtos; maiores margens de lucro; e redução de risco no desenvolvimento de novos produtos. Enquanto LPSs têm demonstrado benefícios à engenharia de software, elas não lidam com abstrações relacionadas com agentes, tais como crenças, objetivos e planos, para permitir uma derivação sistemática de SMAs. Customização em massa permite que sistemas sejam mais flexíveis e coerentes com as necessidades dos clientes, o que significa que as *features* e opções dos produtos podem ser selecionadas e produtos finais podem ser efetivamente gerados de forma que a produção de software seja facilitada, os custos sejam mais baixos, e o tempo para o produto ser entregue ao usuário seja reduzido. Entretanto, a fim de suportar o desenvolvimento de LP-SMAs, *features* devem ser associadas com abstrações relacionadas com agentes de forma explícita.

Resumindo, nem as abordagens existentes de LPS oferecem modelos para projetar conceitos de agentes e mapear esses conceitos em *features*, nem as metodologias de SMA consideram *features* comuns e variáveis nos modelos de agentes, como também modularização e rastreamento de *features*. A maioria das abordagens atuais de LPS são voltadas para as tecnologias orientadas-a-objetos e orientadas-a-componentes, e apenas pesquisa recente visou à combinação de LPSs e SMAs. O resultado dessa integração é denominada Linha de Produtos de Sistema Multi-agentes (LP-SMA), e tem por objetivo obter vantagens de ambas tecnologias para assistir à exploração industrial da tecnologia de agentes. Entretanto, não há trabalho anterior que ofereça um processo de engenharia de domínio para o desenvolvimento de LP-SMAs.

Nós caracterizamos uma LP-SMA por uma LPS que contenha *features* de agentes, as quais são *features* que apresentam propriedades típicas de agentes, tais como comportamento autônomo ou pró-ativo, e que conseqüentemente pode ser apropriadamente modelado utilizando-se as abstrações de agentes. Para modelar essas *features* de agentes, é essencial rastrear o seu comportamento e a localização de sua variabilidade. Nosso processo de engenharia de domínio define fases, atividades e *work products* que envolvem tanto a modelagem como rastreabilidade de *features* de agentes. Na definição do nosso processo, nós introduzimos modelos novos e modificados, bem como tiramos proveito de algumas atividades e notações que consistem partes de métodos de abordagens existentes de LPSs e SMAs. As abordagens que foram utilizadas em nosso processo foram: (i) PLUS, que é um método baseado na UML, o qual fornece notações para a documentação de variabilidade; (ii) PASSI, que é uma metodologia orientada-a-agentes, a qual propõe alguns diagramas que foram usados para especificar *features* de agentes em nível de análise; e (iii) MAS-ML, a qual foi a linguagem de modelagem utilizada na fase de projeto do domínio para modelar agentes, papéis e organizações. Além disso, nós explicitamente separamos a modelagem de *features* de agentes e que não são de agentes, a fim de facilitar a integração da nossa abordagem com abordagens que não envolvam *features* de agentes (e.g. orientação-a-objetos), e para dar suporte à evolução de aplicações e LPSs existentes que foram desenvolvidas usando outras metodologias através da incorporação de *features* de agentes. Um cenário específico que nós estamos explorando é a evolução de aplicações *web* desenvolvidas com arquiteturas tradicionais, às quais nós adicionamos *features* de agentes, tais como recomendação autônoma de produtos e informações aos usuários.

Palavras–chave

Linhas de Produto de Sistemas Multi-agentes. Engenharia de Domínio. Processo de Software. Linhas de Produto de Software. Sistemas Multi-agentes. Engenharia de Software.

Table of Contents

List of Figures

List of Tables

Abbreviation and Acronym List

AOP Aspect-oriented Programming

AOSE Agent-oriented Software Engineering

API Application Program Interface

BDI belief-desire-intention

DAO Data Access Object

EC ExpertCommittee

FIPA Foundation for Intelligent Physical Agents

FODA Feature-Oriented Domain Analysis

FORM Feature-Oriented Reuse Method

GUI Graphical User Interface

KobrA Komponentenbasierte Anwendungsentwicklung

MaCMAS Methodology Fragment for Analysing Complex MultiAgent Systems

MAS-PL Multi-agent System Product Line

MAS Multi-agent System

MDD Model-driven Development

MPP marketing and product plan

MVC model-view-controller

OMG Object Management Group

OLIS OnLine Intelligent Services

OVM Orthogonal Variability Model

PASSI Process for Agent Societies Specification and Implementation

PLUS Product Line UML-based Software Engineering

SPEM Software Process Engineering Metamodel Specification

SPL Software Product Line

SPLE Software Product Line Engineering

TAO Taming Agents and Objects

UML Unified Modeling Language

WAF Web Application Framework

XML Extensible Markup Language

1
Introduction

Over the past decades, agents have become a powerful software abstraction to support the development of complex and distributed systems (Jennings 2001). They are a natural metaphor to understand systems that present some particular characteristics such as high interactivity and multiple loci of control. These systems can be decomposed in several autonomous and pro-active agents comprising a Multi-agent System (MAS). This autonomy property refers to agents able to act without the intervention of humans or other systems: they have control both over their own internal state, and over their behavior (Wooldridge 1999). In order to develop agent-based systems, adequate techniques that explore their benefits and their peculiar characteristics are required (Zambonelli 2000). In this context, Agent-oriented Software Engineering (AOSE) has emerged as a new software engineering paradigm to support the development of MASs and to help on the industrial exploitation of agent technology; and then several research work has been proposed in this direction, such as methodologies (Wooldridge 2000a, Cossentino 2005, Bresciani 2004), modeling languages (Silva 2007, Silva 2003a, Silva 2004a, Silva 2008, Bauer 2001, Bauer 2002), development platforms (Bellifemine 2007, Pokahr 2005, Bordini 2007, Howden 2001), tools and programming languages. As a consequence, MASs are gaining wide acceptance in both industry and academia; however, although many agent-oriented methodologies have been proposed, none is mature enough to be used in industrial and commercial environments (Shehory 2001).

The main aims of software engineering is to produce methods, techniques and tools to develop software systems with high levels of quality and productivity. Software reuse (Griss 1997) is one of the main strategies proposed to address these software engineering goals. Software reuse techniques have contributed for significant improvements to reduce both time and cost of software development. Over the last years many reuse techniques have been proposed and refined by the software engineering community. Examples of these techniques are: component-based development (Szyperski 2002), object-oriented application frameworks (Fayad 1999) and libraries, software architectures

(Shaw 1996) and patterns (Gamma 1995, Buschmann 1996). In addition, some research work has been published in order to bring the advantages of software reuse to the MAS development. Several of these works are exploiting patterns reuse (Cossentino 2002, Cossentino 2003, Gonzalez-Palacios 2004, Lind 2002); however, most of the agent-oriented methodologies do not take into account the adoption of extensive reuse practices, and address the development of single systems (Girardi 2002).

In the context of software reuse, the concepts of system families and Software Product Lines (SPLs) have gained a significant popularity throughout the software industry and research community, leading to the emergence of a new field called Software Product Line Engineering (SPLE). In this approach, reuse evolves from an *ad-hoc* to a systematic way. A SPL (Clements 2002) refers to a family of systems sharing a common, managed set of features that satisfy the needs of a selected market. The systems are developed from a common set of core assets in a prescribed way. SPLE aims at exploring existing common and variable features of the set of systems, in such way that different customized applications can be developed from a reusable set of artifacts with a reduced time-to-market. A feature (Czarnecki 2006) is a property or functionality of the SPL that is relevant to some stakeholder. The development of a SPL is typically divided into two key processes: (i) domain engineering – the commonality and the variability of the SPL are defined and realized; and (ii) application engineering – the applications of the SPL are built by reusing domain artifacts and exploiting the product line variability.

Over the last years, several approaches for developing system families and SPLs have been proposed (Weiss 1999, Czarnecki 2000, Atkinson 2002, Clements 2002, Gomaa 2004, Pohl 2005). The main goal of most of these approaches is to define, model and implement a common and flexible architecture, which addresses the common and variable features of the SPL. In order to develop such architecture, current approaches either provide high-level guidelines to implement SPLs (Weiss 1999, Clements 2002) or use different existing technologies, such as component-based (Atkinson 2002), object-oriented (Gomaa 2004), code generation (Czarnecki 2000) and Aspect-oriented Programming (AOP) (Alves 2006, Kulesza 2006a).

Only recent research (Dehlinger 2007, Pena 2006a) has explored the integration synergy of SPLs and MASs technologies, by incorporating their respective benefits. This integration results in product lines that present features that take advantage of agent technology, comprising Multi-agent System Product Lines (MAS-PLs). These features, named agent features, present an autonomous or pro-active behavior; therefore the agent abstraction

is appropriate for their development.

1.1 Problem Statement and Limitations of Existing Work

As stated in the previous Section, both MASs and SPLs can provide several benefits for the software development. Nevertheless, little effort has been made in order to combine these technologies. In (Pena 2006b), there are many challenges in the MAS-PL development to be overcome. After reviewing SPL, MAS and MAS-PL approaches, we have identified some issues that need to be addressed in the MAS-PL development.

SPL methodologies do not address agent features. In (Nunes 2008a), we have identified that most of the SPL methodologies provide useful notations to model the agent features. However, none of them completely covers their specification. Agent technology provides particular characteristics that need to be considered in order to take advantage of this paradigm.

MAS methodologies do not address the development of system families. Several MAS methodologies have been proposed (Sellers 2005); however they are devoted to developing single products. Therefore, these approaches do not cover some of the activities of SPL. These are mainly concentrated on commonality analysis, and its implications for the SPL approach.

MAS methodologies do not consider the integration with other existing approaches. MAS methodologies usually propose to distribute all the system functionalities and responsibilities among agents. Agents are an abstraction that provides some particular characteristics, such as autonomy and pro-activeness. Therefore, we claim that features of the SPL that do not take advantage of agent technology can be modeled and implemented using traditional design and programming techniques.

MAS-PL approaches fail to provide a complete solution for MAS-PL development. MAS-PL approaches do not address development scenarios of traditional SPL architectures using agent technology. Instead, they adopt an existing MAS methodology as a basis and extend it with SPL techniques for a particular purpose. The main problems that we have observed (Nunes 2008a) in these MAS-PL approaches to model and document MAS-PLs were: (i) they do not offer a complete solution to address the modeling of agent features in both domain analysis and design; (ii) they suggest the introduction of complex and heavyweight notations that are difficult to understand when adopted in combination with existing notations (e.g. UML); and (iii) do not explicitly capture the separate modeling of agent features.

1.2 Proposed Solution and Contributions Overview

This dissertation contemplates the definition of a domain engineering process for developing MAS-PLs. The process focuses on system families and includes domain scoping and variability modeling techniques. Our approach is the result of an investigation of how current SPL, MAS and MAS-PL approaches can model MAS-PL. Based on this experience, we propose our process, which is built on top of some MAS and SPL approaches. We have combined some techniques and notations of these approaches (Gomaa 2004, Cossentino 2005, Silva 2007) and added some extensions/adaptations. The scenario that we have explored is the incorporation of autonomous or proactive behavior into existing web systems. The main idea is to introduce software agents into existing web applications in order to allow the (semi)automation of tasks, such as autonomous recommendation of products and information to users (Holz 2008). Due to the existence of many web applications already developed and deployed on application servers, our MAS-PL approach aims at extending these web applications with the aim to bring the minimum impact to their provided features and services that are adequately structured according to classical architectural patterns, such as Layer and model-view-controller (MVC) (Buschmann 1996).

Therefore, the main contributions of this dissertation are:

(i) to define a whole domain engineering process for developing MAS-PLs, detailing its activities and their respective artifacts;

(ii) to provide notations to model and document agent variability;

(iii) to provide models to capture agent features traceability;

(iv) our approach models agent features independently, promoting a low impact in the incorporation of agents into existing systems designed with other technologies, such as object-oriented; and

(v) to present design and implementation guidelines to help in the development of MAS-PLs, in particular an architectural pattern that provides a general structure to add autonomous behavior to existing web applications using agent technology.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows.

Chapter 2 presents the state-of-art of the two development technologies combined in this dissertation: Multi-agent Systems and Software Product Lines. Besides presenting them, we point out some of their relevant approaches.

Chapter 3 presents the proposed domain engineering process. First, it gives an overview and details the key concepts of the process, and later describes each one of the phases and activities that compose it.

Chapter 4 describes two MAS-PL case studies for the web-domain, which were developed in order to help in the elaboration of our process and to evaluate it. Besides detailing the case studies, it presents artifacts generated during their development.

Chapter 5 presents lessons learned during the development of our process and case studies. We show guidelines to design and implement agent features in MAS-PLs. These guidelines aims at providing modularization to agent features in order to support variability. In addition, we discuss identified limitations of our approach.

Chapter 6 presents the related work of this dissertation. Two kinds of work are shown: MAS-PL approaches and integration approaches of web applications and software agents.

Chapter 7 presents concluding remarks and directions for future work. The main contributions of the dissertation are also detailed in this Chapter.

2
Background on Multi-agent Systems and Software Product Lines

Multi-agent System Product Lines (MAS-PLs) aim at integrating Software Product Lines and agent-oriented techniques by incorporating their respective benefits and helping the industrial exploitation of agent technology. This Chapter gives an overview of these two technologies, detailing some of their methodologies and processes. Furthermore, a comparative analysis among Software Product Line approaches is presented in Section 2.1.5. The purpose of this investigation is to identify how Software Product Lines and agent-oriented techniques can help in the development of MAS-PLs.

2.1
Software Product Lines

Software Product Lines (SPLs) are a new trend in the context of software reuse that provide a way to design and implement several closely related systems together by changing the form of reuse from an *ad-hoc* to a systematic way. The term family of programs was first introduced by Parnas in (Parnas 1976), defining it as a set of programs with so many common properties that it is an advantage to study these common properties before analyzing individual members. Nowadays, the concept given to a SPL is similar to this definition (Clements 2002): "a set of software intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". According to (Czarnecki 2000), a feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a product line. The features are organized into a coherent model referred to as a feature model (originally proposed in (Kang 1990)), which specifies the features of a product line as a tree, indicating mandatory, optional and alternative features. Mandatory features are part of the SPL core and are present in all products derived from it. Optional features are present just in some members of the SPL and alternative features are the ones that vary from one member to another.

Features are essential abstractions that both customers and developers understand. The main aim of SPLE is to analyze the common and variable features of applications from a specific domain, and to develop a reusable infrastructure that supports the software development. Variability management is the major product-line-unique discipline that must be newly established within non-product-line organizations. It is responsible for systematically managing the scope itself, and ensuring its traceability with genericity of product line artifacts.

There are several motivations for the adoption of a SPL approach, being the three main ones the reduction of developments costs, enhancement of quality and reduction of time-to-market. The costs are reduced by reusing artifacts to derive products from the SPL. This is the same reason for a reduced time-to-market. The enhancement of quality is achieved by reviewing and testing the SPL artifacts in many products. Nevertheless, the development of reusable artifacts requires an up-front investment and a higher time-to-market in the initial phases of the SPL development, but this effort is usually compensated after the third derived product (Pohl 2005). Additional motivations for SPL are: (i) reduction of maintenance effort – whenever an artefact from the SPL is changed, the changes can be propagated to all products in which the artefact is being used; (ii) organized evolution – the introduction of a new artefact into the SPL core gives an opportunity for the evolution of all kinds of products derived from the platform; (iii) complexity reduction – SPLs provide a structure that determines which components can be reused at what places by defining variability at distinct locations.

Migrating from a single-system engineering to a SPL approach has far-reaching consequences. It usually requires a company reorganization in order to establish new modular units compromised with the SPL management or even though to redefine processes, workflows and technologies that are being used. According to (Krueger 2001), SPL adoption strategies are classified in three different categories: proactive, reactive and extractive. The proactive approach motivates the development of product lines considering all the products in the foreseeable horizon. A complete set of artifacts to address the product line is developed from scratch. In the extractive approach, a SPL is developed starting from existing software systems. Common and variable features are extracted from these systems to derive an initial version of the SPL. Finally, the reactive approach advocates the incremental development of SPLs. Initially, SPL artifacts address only a few products. When there is a demand to incorporate new requirements or products, common and variable artifacts are incrementally extended in reaction to them. Commonly, a pro-

active approach demands a huge up-front investment to develop artifacts that address the whole family of systems – as it was previously stated, the pay-off is around three systems. All the effort is concentrated at the initial phases of the SPL development. In addition, companies might not take the risk to invest in a SPL development, considering that the requirements can change. Nevertheless, the adoption of the reactive and extractive approaches allow the amortization of this investment, distributing it in incremental cycles. These approaches are not mutually exclusive. A combined strategy can be, for instance, to develop a SPL based on legacy software and incrementally evolve the architecture to address new features, enabling the derivation of new products. This means that first an extractive approach is adopted, followed by a reactive approach.

To enable SPLE, a well-accepted convention is to divide the engineering process into two different processes: domain engineering and application engineering. Domain Engineering is the process of SPLE in which the commonality and the variability of the SPL are defined and realized. It is responsible for scoping the SPL, and ensuring that the core assets have the variability that is needed to support the desired scope of products. Domain engineering approaches (Kang 1990, Prieto-Diaz 1999, Almeida 2007) aim at collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems (Czarnecki 2000). Domain engineering encompasses three main process components: (i) Domain Analysis – the main concepts and activities in a domain are identified and modeled using adequate modeling techniques. The common and variable parts of a family of systems are identified; (ii) Domain Design – whose purpose is to develop a common system family architecture and production plan for the SPL; and (iii) Domain Implementation (or Realization) – involves implementing the architecture, components, and the production plan using appropriate technologies. The qualifier "domain" emphasizes the multisystem scope of these process components. Application Engineering is the process of SPLE in which applications of the SPL are built by reusing domain artifacts and exploiting the SPL variability. It takes the common assets of the SPL and uses them to create products. Domain engineering and application engineering can be called *engineering-for-reuse* and *engineering-with-reuse*, respectively.

Next sections describe some of the main SPL approaches, detailing their key concepts and method structures. Next, a comparative analysis of these approaches is presented (Section 2.1.5). This study helped to understand these approaches and identify their usefulness and deficiencies while documenting

and modeling MAS-PLs.

2.1.1 Feature-Oriented Reuse Method (FORM)

Feature-Oriented Reuse Method (FORM) (Kang 1998) is a systematic method that analyzes applications in a domain in order to identify their commonalties and differences in terms of features. The analysis results are then used to develop domain architectures and components. The model that captures the commonalties and differences is called the feature model and it is used to support both engineering of reusable domain artifacts and development of applications using the domain artifacts. FORM extends Feature-Oriented Domain Analysis (FODA) (Kang 1990) to the software design phase and prescribes how the feature model is used to develop domain architectures and components for reuse.

The FORM method consists of two major engineering processes: domain engineering and application engineering. There are three phases in FORM domain engineering: context analysis, domain (or feature) modeling, and architecture (and component) modeling. FORM proposes the modeling of a SPL architecture using three models: (i) *subsystem model* – presents the overall system structure; (ii) *process model* – details the dynamic behavior of the system; and (iii) *module model* – specifies each reusable component of the architecture. Application engineering proceeds by first analyzing user's requirements and selecting appropriate and valid domain features from the feature model, identifying the corresponding reference model, and completing the application development by reusing software components in a bottom-up fashion.

FORM states some engineering principles for designing reusable architectures and components, which are: (i) separation of concerns and information hiding; (ii) localization of function, data, and control; (iii) parametrization of artifacts using the features; (iv) layering; (v) separation of components from the component connection mechanism; and (vi) synthesis of design components based on feature selection. These engineering principles support the development of architectures that are adaptable and reconfigurable for different applications.

The FORM authors have later extended (Kang 2002) the FORM method to incorporate a marketing and product plan (MPP) to help propel asset development. MPPs identify the information to gather during the marketing and business analysis. It includes a market analysis, a marketing strategy, product features, and product feature delivery methods.

2.1.2 Framework for SPLE by Pohl et al.

Pohl et al. propose in (Pohl 2005) a framework for SPLE that incorporates the central concepts of traditional SPLE, namely the use of platforms – a collection of reusable artifacts – and the ability to provide mass customization. The framework encompasses the two process of the SPLE paradigm: domain engineering and application engineering. Their proposal has its roots in the ITEA projects ESAPS, CAFÉ, and FAMILIES and is based on the differentiation between the domain and application engineering processes. For each one of the sub-processes that compose the framework, the authors provide their inputs and outputs, activities to be performed and emphasize the differences from single-system engineering.

The domain engineering process is composed of five sub-processes: (i) Product Management – deals with the SPL economic aspects and defines the SPL scope; (ii) Domain Requirements Engineering – provides the activities to elicit and document common and variable requirements; (iii) Domain Design – provides the activities to define the reference architecture; (iv) Domain Realization – deals with the detailed design and software components implementation; and (v) Domain Testing – responsible for validating and verifying the reusable components and developing reusable test artifacts. The application engineering encompasses similar sub-processes (Application Requirements Engineering, Application Design, Application Realization and Application Testing). Each of the sub-processes uses domain artifacts and produces application artifacts.

Variability management is an important activity in SPLs. For managing variability, an adequate documentation of variability information is essential. Although most of SPL approaches use feature models to document variability, Pohl et al. propose the Orthogonal Variability Model (OVM), a model that defines the variability of a SPL, relating the variability defined to other software development models such as feature models, use case models, design models, component models, and test models. They claim that with this approach there is a significant reduction of model size and complexity, because only the variable aspects of a product line are documented in the OVM models. Additionally, the OVM approach can be used to document the variability of arbitrary development artifacts, ranging from textual requirements to design models and even test cases. The core concepts of the OVM language are variation point ("what does vary") and variant ("how does it vary"). Each variation point has to offer at least one variant (offers-association). Additionally, the constrains-associations between these elements describe dependencies

between variable elements.

2.1.3 Product Line UML-based Software Engineering (PLUS)

Product Line UML-based Software Engineering (PLUS) (Gomaa 2004) is a UML-based software design method for product lines, which provides a set of concepts and techniques to extend UML-based design methods and processes for single systems in a new dimension to address SPLs. It explores how each of the Unified Modeling Language (UML) modeling views – use case, static, state machine, and interaction modeling – can be extended to address software product families.

The method consists of two major processes or life cycles: Software product line engineering and Software application engineering. Each one of them encompasses five phases: Requirements Modeling, Analysis Modeling, Design Modeling, Implementation and Testing. In the SPLE process, a product line use case model, product line analysis model, software product line architecture, and reusable components are developed. All artifacts produced during each phase are stored in the SPL repository for being used during the software application engineering in order to derive individual applications.

Besides the method structure, PLUS provides an UML-based notation to address variability in the requirements, analysis, and design models. It basically uses stereotypes to distinguish among orthogonal characteristics of elements of the models, such as $\ll kernel \gg$, $\ll optional \gg$ and $\ll alternative \gg$ for representing the reuse category; and $\ll entity \gg$ and $\ll control \gg$ for representing the role of the class in the application. The method also proposes notations, such as the use of packages or tables to represent feature/elements of the models dependencies, and architectural patterns for SPLs.

2.1.4 Komponentenbasierte Anwendungsentwicklung (KobrA)

The Komponentenbasierte Anwendungsentwicklung (KobrA) (Atkinson 2000) method is a component based product line engineering approach with UML, which has been developed at the Fraunhofer Institute for Experimental Software Engineering (IESE). KobrA is an abbreviation of *Komponentbasierte Anwendungsentwicklung* and means a component-based application development method. The authors from KobrA approach aimed at developing a simple, systematic, prescriptive, flexible and scalable method.

KobrA's software life cycle consists of two basic SPLE activities: Framework Engineering and Application Engineering. The purpose of the Framework

Engineering activity is to create, and later maintain, a generic framework that embodies all product variant in a family, including information about their common and disjoint features. This activity applies the Komponent, i.e. KobrA component, modeling and implementation activities, accompanied by additional subactivities for handling variabilities and decision models, to support a family of similar applications. A framework contains a generic Komponent tree that captures common and variable characteristics of a SPL.

The subactivities that are part of the Framework Engineering are the following: (i) Context Realization – produces a set of realization models for the environment of the system; (ii) Component Specification – produces a specification of a component in relationship to a given realization; (iii) Component Realization – produces a realization of a component in relationship to a given specification; (iv) Component Reuse – matches the needs of a given realization to the capabilities of a preexisting component via a mutually agreed specification; and (v) Quality Assurance – ensures the quality of the KobrA models using inspections, testing and quality modeling.

The Application Engineering activity uses the framework developed during Framework Engineering to build particular applications. The variabilities in the framework are removed, and the decisions in the decision model are resolved.

2.1.5 Comparative Analysis

The SPL approaches presented in the previous Sections were compared according to different aspects related to the SPLE. The goals of this comparison were to obtain a clearer overview of the approaches and find out how existing approaches differ for the development of SPLs; however it was not a goal to rate these approaches in order to choose the best one. The comparison was made using an evaluation framework proposed in (Matinlassi 2004), whose purpose is to compare SPL design methods. The framework considers the methods from the points of view of method context, user, structure and validation. The categories and elements that compose the evaluation framework are depicted in Table 2.1.

Tables 2.2 and 2.3 present several approaches for developing SPLs and MAS-PLs described according to the evaluation framework. The approaches are: FORM (Kang 1998), KobrA (Atkinson 2000), Pohl et al.'s Framework (Pohl 2005), PLUS (Gomaa 2004), MaCMAS Extension (Pena 2007) and GAIA-PL (Dehlinger 2007). The first four approaches are SPL methodolgies/methods/processes, which were detailed in the previous Sections. The last

Table 2.1: Evaluation Framework – Source: (Matinlassi 2004).

Category	**Element**	**Question**
Context	Specific Goal	What is the *specific* goal of the method?
	Product Line Aspect(s)	What aspects of the product line does the method cover?
	Application Domain(s)	What is/are the application domain(s) the method is focused on?
	Method Inputs	What is the starting point for the method?
	Method Outputs	What are the results of the method?
User	Target Group	Who are the stakeholders addressed by the method?
	Motivation	What are the user's benefits when using the method?
	Needed Skills	What skills does the user need to accomplish the tasks required by the method?
	Guidance	How does the method guide the user while applying the method?
Contents	Method Structure	What are the design steps that are used to accomplish the method's specific goal?
	Architectural Viewpoints	What are the architectural viewpoints the method applies?
	Language	Does the method define a language or notation to represent the models, diagrams and other artifacts it produces?
	Variability	How does the method support variability expression?
	Tool Support	What are the tools supporting the method?
Validation	Method Maturity	Has the method been validated in practical industrial case studies?
	Architecture Quality	How does the method validate the quality of the output it produces?

two are MAS-PL approaches that are detailed in Section 6.1. This investigation did not have the purpose of selecting the best approach for SPL development but of helping to identify useful activities and notations for developing MAS-PLs.

Based upon the results of this study, the following conclusions are offered regarding the development of SPLs:

- Feature model is the typical notation to model variability in SPLs. Proposed in FODA, it is used in FORM, PLUS, MaCMAS Extension, FeatuRSEB (Griss 1998) and in several research works (Lee 2006, Satyananda 2007). Some approaches use alternative notations:
  - Pohl et al. employ OVMs for variability management, providing a general variability model comprising the domain's variation points and variants. With these models, the authors claim that a reduced complexity can be achieved, because only the variable aspects of a SPL are modeled. However, their approach proposes mapping the OVM into other models, such as use case and class diagrams, and this approach can not be scalable with complex diagrams. In addition, managing all the features of a SPL including the mandatory ones can help in evolution scenarios of SPLs, such as when a mandatory feature become optional. Moreover, feature models are widely known in the SPL community and have tool support (Antkiewicz 2004);

Table 2.2: Approaches Comparison (1).

Category	Element	FORM	KobrA	Pohl et al.'s Framework
Context	Specific Goal	How to apply domain analysis results (commonality and variability) to the engineering of reusable and adaptable domain components with specific guidelines.	Support a model-driven UML-based representation of components, and a product line approach to their development and evolution. Be as concrete and prescriptive as possible.	Define the key sub-processes of the domain engineering and application engineering processes as well as the artefacts produced and used in these processes.
	Product Line Aspect(s)	Extends FODA to the design and implementation phases and prescribes how the feature model is used to develop domain architectures and components for reuse.	Defines a full SPLE process with activities and artifacts, including implementation, releasing, inspection and testing.	Defines a full SPLE process with activities and artifacts, dividing the process into two key-processes (Domain engineering and Application engineering) and their respective sub-processes.
	Application Domain(s)	Telecommunication domain and information domain.	Information systems domain.	Home Automation case study.
	Method Inputs	The primary input is the information on systems that share a common set of capabilities and data.	Framework engineering: idea of a new framework with two or more applications; Application engineering: elicitation of user requirements within the scope of the framework.	Domain engineering: company goals defined by top management; Application engineering: domain requirements and product roadmap with the major features of the corresponding application.
	Method Outputs	Domain engineering: feature model, reference architecture and reusable components; Application engineering: application software, application architecture and reusable components (selected from feature model/reference architecture/reusable components respectively).	Framework engineering: generic Komponents and decision model; Application engineering: framework instantiated for one particular member of the SPL.	Domain engineering: Reusable software components; Application engineering: running application together with the detailed design artefacts.
User	Target Group	Academic audience.	Software engineers and designers currently working in the industry.	Students, professionals, lecturers, and researchers interested in SPLE.
	Motivation	Customers and engineers often speak of product characteristics in terms of "features the product has and/or deliver".	Simple, systematic, scalable and practical method.	OVM is smaller and less complex than extended UML diagrams or feature models.
	Needed Skills	FORM Notation	-	OVM Notation
	Guidance	Case studies, special guideline	Case studies, extensive manuals	Book with case studies
Contents	Method Structure	First, define the context of the system. After the main two phases are Domain engineering and Application engineering.	Framework engineering and Application engineering.	First, define the product roadmap. After the main two phases are Domain engineering and Application engineering.
	Architectural Viewpoints	Subsystem, process and module.	Specification, realization, containment and context realization.	Logical, development, process and code views.
	Language	-	Adapted UML and manual transformation to code	Own language to design the variability model and UML
	Variability	Define support for variability in requirements elicitation.	Concentrate on capturing variability with graphical language in architectural design.	Captures variability into a variability model.
	Tool Support	ASADAL.	Commercial UML tool; word processing tool and configuration management.	-
Validation	Method Maturity	Validated in practical industrial case studies.	Validated in practical industrial case studies.	Framework defined based on industrial experiences and the results of the European SPLE research projects ESAPS, CAFÉ, and FAMILIES.
	Architecture Quality	Non-architectural evaluation methods, such as model checking, inspections and testing.	Scenario based architecture evaluation (SAAM) for ensuring maintainability.	Testing in Domain and Application engineering.

Table 2.3: Approaches Comparison (2).

Category	**Element**	**PLUS**	**MaCMAS Extension**	**GAIA-PL**
Context	Specific Goal	Provide a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle software product lines.	Provide explicit support for MAS-PLs and manage the modeling of the evolution of the system in a systematic way.	Capture requirements in such a way that they can be easily and safely reused during system evolution for mission-critical, agent-based distributed software systems.
	Product Line Aspect(s)	Extends the USDP adding: Requirements Modeling, Analysis Modeling, Design Modeling and Application Engineering.	Domain requirements engineering, domain design and domain realization.	Requirements engineering phase.
	Application Domain(s)	Microwave Oven, Electronic Commerce and Factory Automation case studies.	NASA (Prospecting Asteroid Mission) and human organization case studies.	Constellation of context-aware microsatellites case study.
	Method Inputs	SPL engineering: product line requirements; Application engineering: application requirements and SPL repository.	Domain Engineering: requirements statement, goal hierarchy; Application engineering: requirements statement, goal hierarchy, feature model, core architecture.	Requirements.
	Method Outputs	SPL Engineering: SPL repository; Application engineering: executable application.	Domain Engineering: core architecture; Application Engineering: products (MAS seen as a collection od multiple products).	Role Schema and Role Variation Points.
User	Target Group	Professional and academic audience.	-	-
	Motivation	Extends object-oriented methods to model product families and provides explicit modeling of the similarities and variations in a product line.	First approach that deals with architectural changes of evolving systems based on MAS-PL.	Dynamically changing configurations of agents can be captured and reused for future similar systems.
	Needed Skills	-	MacMAS Methodology	GAIA Methodology
	Guidance	Book with case studies	Articles and Case Studies	Articles and Case Studies
Contents	Method Structure	Software product line engineering and Software application engineering.	Domain engineering and Application engineering.	-
	Architectural Viewpoints	Use Case Model, Static Model, Collaboration Model, Statechart Model and Feature Model views.	Static acquaintance and Behavior of acquaintance organization views.	-
	Language	UML (use of stereotypes).	MacMAS	Natural Language
	Variability	Feature Model (with UML).	Feature Model	Role Schema and Role Variation Points
	Tool Support	UML Design Tools and Product Line UML Based Software Engineering Environment (PLUSEE).	Plugin of the ArgoUML CASE tool	Text Editors
Validation	Method Maturity	-	NASA case study	-
	Architecture Quality	Testing in Software product line engineering and Application engineering.	Formal analysis would complement the simulation and testing.	-

  - MaCMAS extension uses a goal-oriented approach to perform the domain analysis, instead of a feature-oriented approach. Goals are represented in a feature model notation in order to provide variability information;
  - Even though PLUS also adopts feature models, it uses a notation based on UML;
  - KobrA, PuLSE (Bayer 1999) and FAST (Weiss 1999) adopt decision models to describe the choices that distinguish distinct members of the family. However, feature models provide an important higher abstraction level.

- Almost all approaches do not address explicitly the modeling of the SPL requirements;

  - PLUS approach defines a customization of the use case model to specify and document the SPL requirements;
  - GAIA-PL proposes a requirements specification template to capture and reuse dynamically changing configurations of agents for future similar systems;

- In the domain design, most of the investigated SPL approaches only provide support to document and detail the SPL architectures in a very high-level manner;

  - PLUS is an object-oriented approach that adopts traditional UML models marked with additional stereotypes to classify the system classes;
  - KobrA is an component-oriented approach that proposes several models (structural, behavioral, functional, non-functional, quality and decision models) to specify its Komponents.

In the context of MAS, the investigated approaches do not provide explicit support to specify and model the SPL architecture and its respective components. However, most of the SPL approaches provide useful notations to model the agent features. Nevertheless, none of them completely covers their specification. Agent technology provides particular characteristics that need to be considered in order to take advantage of this paradigm.

2.2 Multi-agent Systems

Over the past decades, software agents have become a powerful abstraction to support the development of complex and distributed systems (Jennings 2001). They are a natural metaphor to understand systems that present some particular characteristics such as high interactivity and multiple loci of control. These systems can be decomposed in several autonomous and pro-active agents comprising a Multi-agent System (MAS). A software agent is an abstraction that enjoys mainly the following properties (Wooldridge 1995):

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

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

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

Pro-activeness agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

Agent-oriented Software Engineering (AOSE) (Wooldridge 2000b) has emerged as a new software engineering paradigm to help on the development of MASs. In this context, several research works were proposed, such as methodologies and processes (Wooldridge 2000a, Cossentino 2005, Bresciani 2004) and modeling languages (Silva 2007). Some of these approaches are described in the next Sections. Further comparison among MAS can be seen in (Sellers 2005).

A problem of the current MAS methodologies is that most of them do not take into account the adoption of extensive reuse practices that can bring an increased productivity and quality to the software development (Girardi 2002). Software reuse techniques, such as component-based development, object-oriented application frameworks and libraries, software architectures, patterns, have been widely used in the software engineering context to provide reduced time-to-marked, quality improvement and lower development costs. Moreover, none of these MAS methodologies address the development of SPLs and, consequently, do not provide notations to express agent variabilities.

2.2.1 Gaia

Using the analogy of human-based organizations, the Gaia (Wooldridge 2000a, Zambonelli 2003) methodology provides an approach that both a developer and a non-technical domain expert can understand, facilitating their interaction. It exploits organizational abstractions and, taking into account the specific issues associated with the exploitation of such abstractions, defines a coherent design process for MAS development. Gaia prescribes following an ordered sequence of steps, the ordered identification of a set of organizational models, and an indication of the interrelationships, guiding developers towards the development of a MAS.

The organizational abstractions exploited by Gaia are: (i) Environment – considering that a MAS is always situated in some environment, the authors believe this is a primary abstraction during the analysis and design phases; (ii) Roles and Interactions – the role of an agent defines what it is expected to do in the organization, both in concert with other agents and with respect to the organization; (iii) Organizational Rules – explicit identification of constraints is very important for the correct understanding of the characteristics that the organization-to-be must express and for the subsequent definition of the system structure by the designer; and (iv) Organizational Structures – the structure of a MAS is more appropriately derived from the explicit choice of an appropriate organizational structure.

The Gaia process starts with the analysis phase, whose aim is to collect and organize the specification, which is the basis for the design of the computational organization (which implies defining an environmental model, preliminary roles and interaction models, and a set of organizational rules). Then, the process continues with the architectural phase, aimed at defining the system organizational structure in terms of its topology and control regime (possibly exploiting design patterns), which, in turn, helps to identify complete roles and interaction models. Eventually, the detailed design phase can begin. Its aim is to produce a detailed, but technology-neutral, specification of a MAS (in terms of an agent model and a services model) that can be easily implemented using an appropriate agent-programming framework. After the successful completion of the Gaia design process, developers are provided with a well-defined set of agent classes to implement and instantiate, according to the defined agent and services model. Gaia considers the output of the design phase as a specification that can be picked up by using a traditional method or that could be implemented using an appropriate agent-programming framework.

Gaia does not directly deal with implementation issues. Its authors state

that it may be the case that specific technology platforms may introduce constraints over design decisions and may require being taken into account during analysis and design. In addition, Gaia does not deal with the activities of requirements capture and modeling and, specifically, of early requirements engineering.

2.2.2 Process for Agent Societies Specification and Implementation (PASSI)

Process for Agent Societies Specification and Implementation (PASSI) (Cossentino 2005) is a step-by-step requirement-to-code methodology for designing and developing multi-agent societies, integrating design models and concepts from both object-oriented software engineering and artificial intelligence approaches using (more properly extending) the UML notation.

The PASSI process encompasses five process components: (i) System Requirements – a model of the system requirements in terms of agency and purpose; (ii) Agent Society – a model of the social interactions and dependencies among the agents involved in the solution; (iii) Agent Implementation – a classical model of the solution architecture in terms of classes and methods; the most important difference with the common object-oriented approach is that we have two different levels of abstraction, the social (multi-agent) level and the single-agent level; (iv) Code – a model of the solution at the code level; and (v) Deployment – a model of the distribution of the parts of the system across hardware processing units and their migration between processing units. The models and phases of the PASSI methodology are illustrated in Figure 2.1.

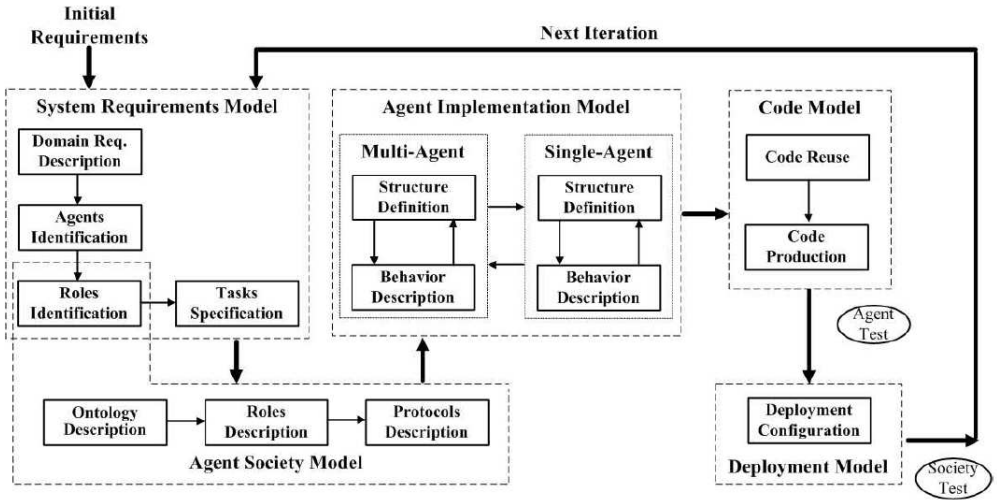


Figure 2.1: PASSI models and phases – Source: (Cossentino 2005).

PASSI differs with some agent-orient methodologies in describing requirements in terms of use case diagrams, instead of making use of goals in require-

ments engineering. In addition, PASSI promotes pattern reuse and code production during the code reuse phase. Code production is strongly supported by the automatic generation of a large amount of code through the PASSI ToolKit (PTK) used to design the system and a library of reusable patterns of code and pieces of design managed by the AgentFactory application.

PASSI was conceived following one specific guideline: the use of standards whenever possible. This justifies the use of UML as modeling language, the use of the Foundation for Intelligent Physical Agents (FIPA) architecture for the implementation of our agents, and the use of Extensible Markup Language (XML) in order to represent the knowledge exchanged by the agents in their messages. An agile version of PASSI, the Agile PASSI, is proposed in (Chella 2006) in order to have a design process that specifically addresses the needs of developing robotic systems.

2.2.3 Tropos

The Tropos methodology (Bresciani 2004) is intended to support all analysis and design activities in the software development process, from application domain analysis down to the system implementation. In particular, Tropos rests on the idea of building a model of the system-to-be and its environment, which is incrementally refined and extended, providing a common interface to various software development activities, as well as a basis for documentation and evolution of the software.

Tropos is based on two key ideas. First, the notion of agent and all related mentalistic notions (for instance goals and plans) are used in all phases of software development, from early analysis down to the actual implementation. Second, Tropos covers also the very early phases of requirements analysis, thus allowing for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human agents.

Tropos adopts the concepts offered by i^* (Yu 1996), a modeling framework proposing concepts such as actor (actors can be agents, positions, or roles), as well as social dependencies among actors, including goal, softgoal, task, and resource dependencies. As a consequence, models in Tropos are acquired as instances of a conceptual metamodel resting on the following concepts/relationships: (i) Actor – models an entity that has strategic goals and intentionality within the system or the organizational setting; (ii) Goal – represents actors' strategic interests; (iii) Plan – represents, at an abstract level, a way of doing something; (iv) Resource – represents a physical or an informa-

tional entity; (v) Dependency (between two actors) – indicates that one actor depends, for some reason, on the other in order to attain some goal, execute some plan, or deliver a resource; (vi) Capability – represents the ability of an actor of defining, choosing and executing a plan for the fulfillment of a goal, given certain world conditions and in presence of a specific event; and (vii) Belief – represents actor knowledge of the world.

The proposed methodology spans four phases that can be used sequentially or iteratively: (i) Early requirements – concerned with the understanding of a problem by studying an organizational setting; (ii) Late requirements – where the system-to-be is described within its operational environment, along with relevant functions and qualities; (iii) Architectural design – the system's global architecture is defined in terms of subsystems, interconnected through data, control, and other dependencies; and (iv) Detailed design – where the behavior of each architectural component is further refined.

The Tropos methodology in its current form is not suitable for sophisticated software agents requiring advanced reasoning mechanisms for plans, goals, and negotiations.

2.2.4 MAS-ML

MAS-ML (Silva 2004b, Silva 2004a, Silva 2007, Silva 2008) is an agent-oriented modeling language, which extends the UML meta-model describing new meta-classes and stereotypes, and extending some diagrams and proposing new ones. The MAS-ML meta-models showing the new metaclasses are depicted in Figures 2.2 and 2.3.

MAS-ML is based on the Taming Agents and Objects (TAO) (Silva 2003b) conceptual framework (meta-model). TAO defines the static and dynamic aspects of MASs. The static aspect of TAO captures the system's elements and their properties and relationships. The elements defined in TAO are agents, objects, organizations, environments, agent roles and object roles. The relationships that link these elements are inhabit, ownership, play, control, dependency, associations, aggregation and specialization. The dynamic aspects of TAO are directly related to the relationships between the elements of MASs and they define the domain-independent behaviors associated with the interaction between MAS elements.

Using the MAS-ML meta-model and diagrams, it is possible to represent the elements associated with a MAS and to describe the static relationships and interactions between these elements. The structural diagrams in MAS-ML are the extended UML class diagram and two new diagrams: organization

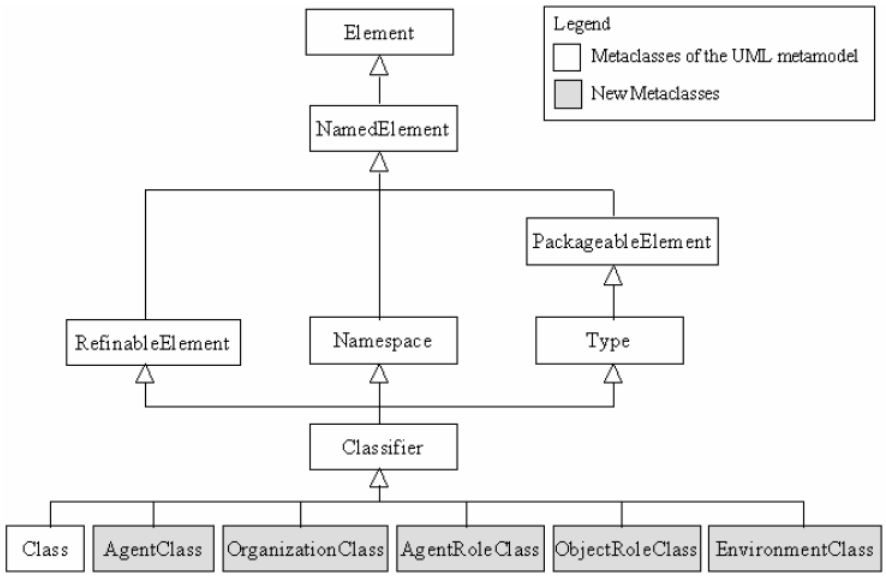


Figure 2.2: MAS-ML metamodel: new metaclasses (part I) – Source: (Silva 2004a).

and role. MAS-ML extends the UML class diagram to represent the structural relationships between agents, agents and classes, organizations, organizations and classes, environments, and environments and classes. The organization diagram models the system organizations and the relationships between them and other system elements. Finally, the role diagram is responsible for modeling the relationships between the roles defined in the organizations. Additionally, MAS-ML extends the sequence diagram to represent the interaction between agents, organizations and environments.

2.3 Final Remarks

This Chapter presented an overview of Software Product Lines and Multi-agent Systems. The former allows the development of families of applications that share common and variable features through a systematic method. The latter are systems whose architecture relies on softwares agents, which are a powerful abstraction to develop complex and distributed systems. Several works have been proposed in both contexts aiming at providing software engineering techniques to help on their development, such as process, methodologies and modeling languages. On the one hand, most of the SPL approaches provide useful notations to model the agent features; however, none of them completely covers their specification. On the other hand, MAS methodologies address the development of single systems, and do not exploit the benefits provided by software reuse, e.g. lower development cost and reduced time-to-market.

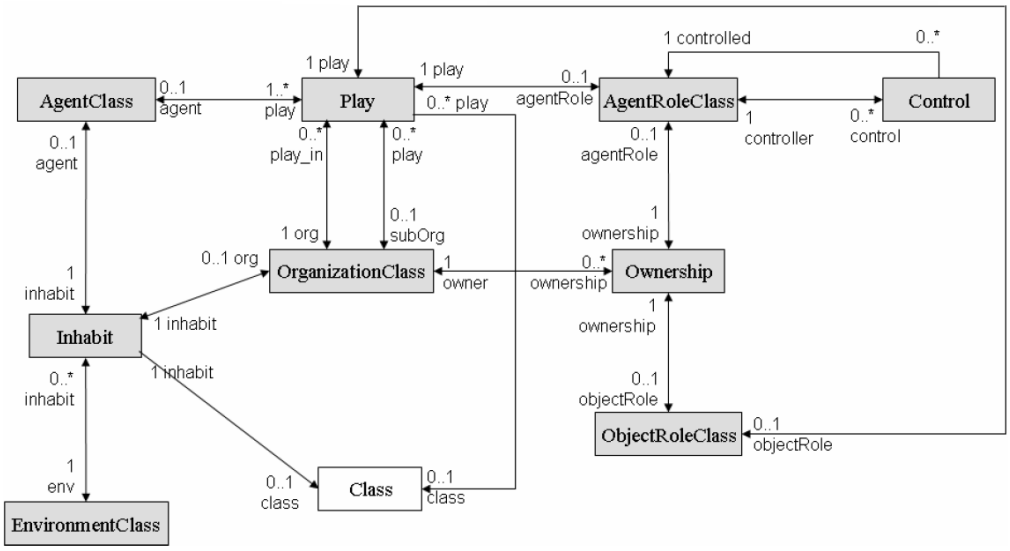


Figure 2.3: MAS-ML metamodel: new metaclasses (part II) – Source: (Silva 2004a).

Even though SPL and MAS approaches do not address the development of MAS-PLs, they provide useful models and notations for that. Therefore, they could contribute to the elaboration of our domain engineering process.

3
The Domain Engineering Process

This Chapter presents our domain engineering process. Initially, it gives an overview of our approach for developing MAS-PLs (Section 3.1), detailing its basic concepts. Current SPL methodologies (Clements 2002, Atkinson 2002, Gomaa 2004, Pohl 2005) cover a great variety of SPL development activities, related to domain and application engineering, and to the process management as well. Nevertheless, they are either too abstract, lacking design details, or are based on technologies that are, for example, object-oriented and component-oriented, not addressing the development of SPLs that use agent technology. On the other hand, agent-oriented methodologies support the development of MASs, but they do not cover typical activities of SPL development, such as feature modeling. As a result, we propose a process that is based on the integration of existing SPL and MAS methodologies, instead of proposing a whole new approach. The MAS and SPL approaches that compose our process are briefly described in Section 3.2, as we also point out why we chose them. We have also defined the agent features granularity that we are dealing with in MAS-PLs development, which is described in Section 3.3. Finally, each one of the process phases and activities are detailed in Section 3.4, explaining their purpose, tasks to be performed and work products (inputs and outputs).

3.1
Process Overview

A first part of our work was to investigate how SPL and MAS approaches deal with MAS-PLs. SPL approaches do not provide models to design agent concepts and map them to features; and MAS methodologies do not consider variability on agent models and do not take into account feature modularization and traceability. However, these approaches provide useful notations and activities that can be integrated to model MAS-PLs. Consequently, our objective is not to create a brand new approach, but to extract the major benefits of some of the current MAS and SPL approaches to compose ours. So, our domain engineering process was conceived by the *incorporation of notations and activities of existing works* in the context of MASs and SPLs, which are:

(i) PLUS (Gomaa 2004) method; (ii) PASSI (Cossentino 2005) methodology; and (iii) MAS-ML (Silva 2007) modeling language. In addition, we propose additional adaptations and extensions for them; as well as new models and activities. Furthermore, some SPL approaches only provide vague and high-level guidelines, consequently users have little idea what they should do. Thus, we aimed at defining a *systematic* process in the sense of providing clear and detailed guidelines about how it should used. Finally, our process is *feature-oriented*, given that system families and the SPLs are analyzed in terms of features and later all the development is driven by them, which means that they should be developed as modularized as possible in order to provide a better feature management and SPL maintenance. Figure 3.1 illustrates our approach by showing how MAS-PLs are modeled in different abstraction levels and some of the artifacts generated in each one of the phases.

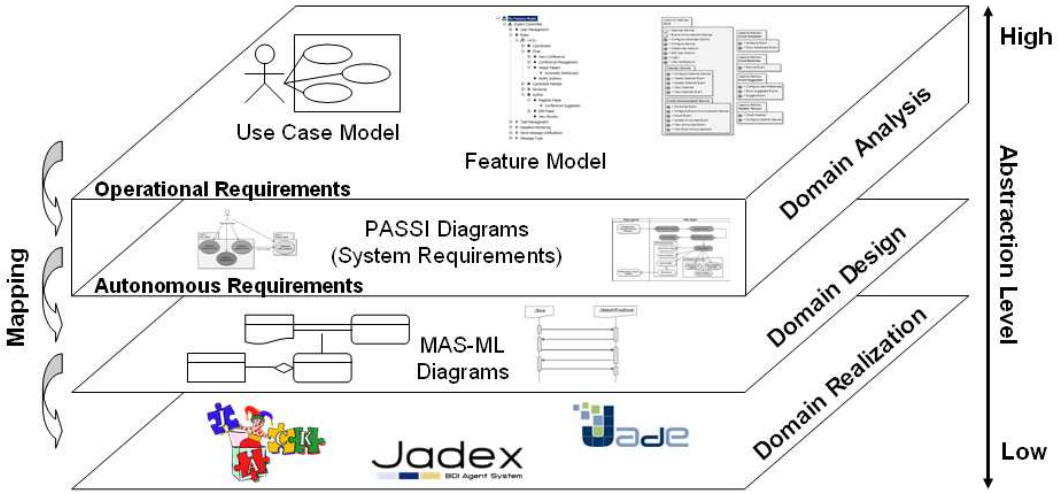


Figure 3.1: Domain Engineering Process Overview.

There are some notations and guidelines that were adopted along all our process, which are: (i) use of ≪*kernel*≫, ≪*optional*≫ and ≪*alternative*≫ stereotypes to indicate variability in elements of several models, such as use cases, classes, components and agents; (ii) separate modeling of features, meaning that diagrams are split according to them (an exception is when modeling crosscutting features); (iii) specific models to provide features traceability along all the process; and (iv) use of colors to structure models in terms of features. A different color is attributed for each feature and this color is used in all model elements related to the feature. This is a redundant information used to provide a better visualization of features traceability, even though it is already provided by dependencies models.

3.1.1 Process Structure

The proposed domain engineering process is structured (in terms of process metamodel) according to the Software Process Engineering Metamodel Specification (SPEM) (OMG 2008) proposed by the Object Management Group (OMG). SPEM provides a common syntax and modeling structure to construct software process models. In addition, it provides a way of documenting processes so that they may be studied, understood, and evolved in a logical, standardized fashion. SPEM 2.0 separates reusable core method content from its application in processes. Roughly, a method content defines the core elements of every method such as roles, tasks, and work product definitions. The tasks define work being performed by roles and are associated to input and output work products (like a diagram or a text document). The tasks can also be categorized into disciplines based upon similarity of concerns and cooperation of work effort.

These core elements of the method content are then used to define the process. The process aggregates activities, which represent general units of work. They can have sub-structures, for instance it can be composed by other activities or a group of tasks. Typically, the activities are grouped into phases, which are significant periods in a process. Our process structure is basically based on three main levels: phases, activities and tasks.

3.1.2 Process Disciplines

The process tasks are classified into six different disciplines. The first five disciplines are part of typical software processes; however they contain some tasks that are specific for MAS-PL development. The last discipline, Configuration Knowledge, encompasses tasks specific for the SPL development. Next, we describe each one of the disciplines and list their tasks.

Requirements. This discipline encompasses the tasks whose purpose is to elicit stakeholder requests and transform them into a set of requirements work products that scope the SPL to be built and provide detailed requirements for what the products of the SPL must do. In addition, the requirements are also specified in terms of features and the commonality and variability of the product line members are analyzed.

Tasks: Document requirements; Elicit requirements; Identify features; Model feature constraints; Validate feature model and Validate requirements.

Business Modeling. The aim of business modeling is to first establish a better understanding and communication channel between business engineering and software engineering. Understanding the business means that software engineers must understand the structure and the dynamics of the target domain, the current problems in the market segment to be addressed by the SPL and possible improvements. These improvements can be the automation of tasks performed by users;

Tasks: Describe Use Cases; Identify agent features; Identify autonomous/proactive behavior use cases; Identify use cases and Refine use cases.

Analysis and Design. The goal of analysis and design is to show how the SPL features will be realized. The aim is to build an architecture that: (i) performs the tasks and functions specified in the use-case descriptions; (ii) support the variability allowing the derivation of specific products; and (iii) is easy to change when the SPL evolves;

Tasks: Choose architectural patterns; Delegate responsibility to agents; Describe agent tasks; Identify agent roles; Identify communication paths; Identify components; Identify ontology concepts; Model organizations; Identify subsystems; Model agents' plans; Model agent roles interaction; Model roles; Model agents' structure; Model components' dynamic behavior; Model components' structure; Model concept relationships; Model reference architecture and Model roles' protocols.

Implementation. The purposes of implementation are to select the different technologies to be used (e.g. frameworks and platforms) and to implement design elements (classes, agents, ...) in terms of components (source files, binaries, executables, and others), comprising the core assets of the SPL.

Tasks: Analyze and select technologies; Identify candidate technologies; Implement Assets; Identify implementation patterns and Define implementation strategy.

Configuration Knowledge. This discipline explains how to trace the features along SPL models. It allows one to detect which use cases, design and implementation elements are related to a specific feature. Features traceability provides a better feature management, an easier SPL evolution and helps on the automation of the derivation process.

Tasks: Model feature/agents dependency; Model feature/concepts dependency; Model feature/components dependency; Model feature/use

case dependency; Map design elements to implementation elements; Refine feature/agents dependency.

3.2 Integrated MAS and SPL Approaches

As stated in Section 3.1, we have incorporated fragments of existing works into our process. Next, we present on overview of each one of them, emphasizing why they were chosen to incorporate our process.

The PLUS method provides a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle SPLs. We adopted PLUS based on our study reported in (Nunes 2008a). Basically the reasons were: (i) PLUS explicitly models the commonality and variability in a SPL, mainly by the use of stereotypes; (ii) it uses feature modeling to analyze variability at an analysis level, which is used by most SPL approaches; and (iii) some approaches focus on managing SPLs (Clements 2002), lacking design details, or just give high level guidelines (Weiss 1999). As a consequence, most of the notations proposed by this approach, such as specific stereotypes for SPL, were used in our process.

Nevertheless, agent technology provides particular characteristics that need to be considered in order to take advantage of this paradigm. In order to model agent features in MAS-PLs, we have adopted the phases of the System Requirements model of PASSI methodology. We made some adaptations in these phases and they became activities incorporated into the Domain Analysis phase of our process. PASSI (Cossentino 2005) is an agent-oriented methodology that specifies models with their respective phases for developing MASs. PASSI integrates concepts from object-oriented software engineering and artificial intelligence approaches, and follows the guideline of using standards whenever possible; and this justifies the use of UML as modeling language. The key reason for choosing PASSI is that the use of an UML-based notation brings facilities to the incorporation of notations proposed in PLUS and keeps a standard to modeling agent and non-agent features.

PASSI also uses conventional class, sequence and activity UML diagrams to design agents, and this approach has been successfully used in the development of embedded robotics applications. However, our focus is to allow the design of agents that follow the belief-desire-intention (BDI) model (Rao 1995). This model proposes that agents be described in terms of three mental attitudes – beliefs, desires and intentions – which determine the agent's behavior. BDI model advantages include: it is relatively mature, and has been successfully used in large scale systems; it is supported by several agent platforms,

e.g. Jadex (Pokahr 2005), Jason (Bordini 2007) and JACK (Howden 2001); and it is based on solid philosophical foundations. As discussed in (Silva 2008), some important agent-oriented concepts, such as environment, cannot be modeled with UML and the use of stereotypes is not enough because objects and agent elements have different properties and different relationships. Thus, our process uses the MAS-ML (Silva 2008) modeling language, with some extensions, to model agents. MAS-ML extends the UML meta-model in order to express specific agent properties and relationships. Using its meta-model and diagrams, it is possible to represent the elements associated with a MAS and to describe the static relationships and interactions between these elements. Others MAS modeling languages do not allow to model some agent concepts (Silva 2008). For instance, AUML (Bauer 2001) does not define organizations and environments and as a consequence the relationships between agents and these elements cannot be modeled.

In summary, we have (i) adopted PLUS notations along all the process; (ii) incorporated three PASSI phases as activities in the Domain Analysis phase; and (iii) used MAS-ML to model agent concepts in the Domain Design phase. In addition, we have proposed (iv) some adaptations to PASSI phases and MAS-ML in order to allow agent variability and agent features traceability; and (v) defined new activities and models to address MAS-PL particularities, as well as specified the sequence and relation among this activities.

3.3 Agent Features Granularity

Features granularity refers to the degree of detail and precision that a design element that implements a feature presents. In the literature, there are many examples of SPLs with coarse-grained features. This means that these features can be implemented wrapped in a specific unit, such as a class or an agent. Besides the usual variabilities present in SPLs, we have considered three different kinds of agent variability in the context of MAS-PLs: (i) agents; (ii) agent roles; and (iii) capabilities. Therefore, using our definition, these are the elements that can be mandatory, optional or alternative when specifying the variability within a MAS-PL architecture. We have excluded the possibility an optional belief, for instance.

A capability (Padgham 2000) is essentially a set of plans, a fragment of the knowledge base that is manipulated by those plans and a specification of the interface to the capability. This concept is implemented by JACK (Howden 2001) and Jadex (Pokahr 2005) agent platforms. Capabilities have been introduced into some MASs as a software engineering mechanism to

support modularity and reusability while still allowing meta-level reasoning. The reason for choosing capabilities instead of beliefs, goals and plans to vary in a MAS-PL is that we believe that variations in these fine-grained elements can be encapsulated into a capability.

Modularity is very important in this context because an essential engineering principle of SPL is the separation of concerns. Separation of concerns is the process of breaking the product line architecture into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program and is used as a synonym of feature. Techniques, such as modularity and encapsulation with the help of information hiding, are used in order to obtain separation of concerns. Separation approaches are of particular interest when features should be realized as components that can be composed in many ways, which is the case in mature and highly flexible architectures.

Even though many SPLs can and have been implemented with the coarse granularity of existing approaches, fine-grained extensions are essential when extracting features from legacy applications (Kästner 2008). An example is considering two versions of a MAS, on which a belief of an agent varied between the two versions. Nevertheless, the modeling techniques that we adopted in our process do not deal with fine-grained features.

3.4 Process Detailed Description

Our proposal is a Domain Engineering process that defines the phases and their respective activities to develop MAS-PLs. Our process was split into typical domain engineering phases (described in Chapter 2): Domain Analysis, Domain Design and Domain Realization. The general purposes of these phases are also the typical ones; however they aggregate some activities, tasks and work products that are specific to model software agents with their variabilities, and agent features traceability.

Figures 3.2 summarizes the phases and activities that compose our process, and the output work products in each one of them. Next sections detail each one of the phases and their respective activities of our domain engineering process. Each activity is illustrated by two figures: (i) activity diagram – shows the order in which the tasks of the activity are performed; and (ii) detailed activity diagram – shows the roles that perform each task, and its inputs and outputs as well. Models and notations adopted to model MAS-PLs are not illustrated in this Chapter, but they can be seen in Chapter 4, on which two MAS-PL case studies are presented.

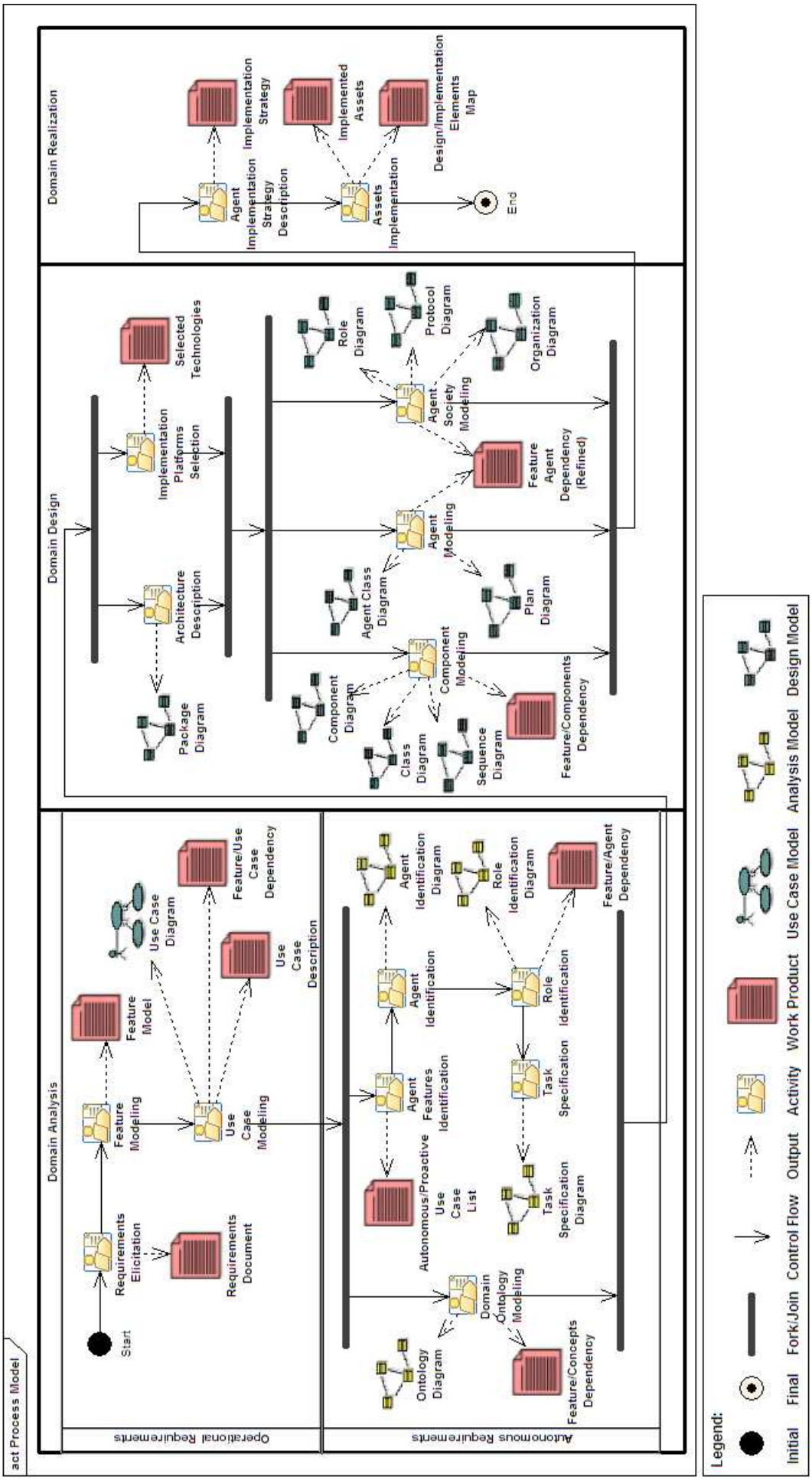


Figure 3.2: The Domain Engineering Process.

3.4.1 Domain Analysis Phase

The Domain Analysis phase defines activities for eliciting and documenting the common and variable requirements of a SPL. It is concerned with the definition of the domain and scope of the SPL, and specifies the common and variable features of the SPL to be developed. This phase is divided in two sub-phases: Operational Requirements and Autonomous Requirements.

Operational Requirements

In the Operational Requirements sub-phase, the family of systems is analyzed and its common and variable features are identified defining the scope of the SPL. Latter, requirements are described in terms of use case diagrams and descriptions. Next, we detail each one of the activities that comprise this phase.

Requirements Elicitation. The first activity of our domain engineering process is to elicit requirements, whose purpose is to understand and provide a very high-level description of the SPL to be developed. It comprises three tasks: (i) Elicit requirements – the Domain Analyst identifies SPL requirements; (ii) Document requirements – each one of the identified requirements are documented and better specified by a Requirements Specifier; and (iii) Validate requirements – the Requirements Specifier validates requirements with Domain Specialists and Stakeholders; if they are not correct, previous tasks must be performed again until requirements are validated.

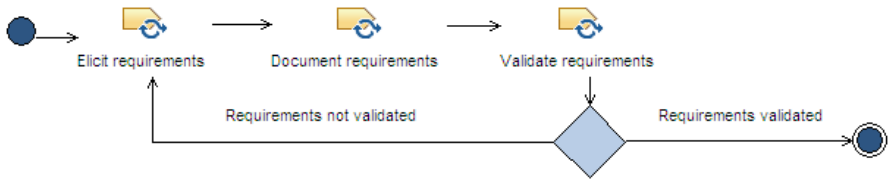


Figure 3.3: Requirements Elicitation activity diagram.

During the execution of all tasks, there must be an effective interaction with both Domain Specialists and Stakeholders. Several conversations, meetings, workshops must be done in order to capture SPL requirements. The output of this activity is the requirements document, which we do not precisely specify. The document can be, for instance, a list detailing each one of the requirements or a table on which rows are requirements, columns are products of the SPL and checked cells mean that a certain requirement is selected for a certain product.

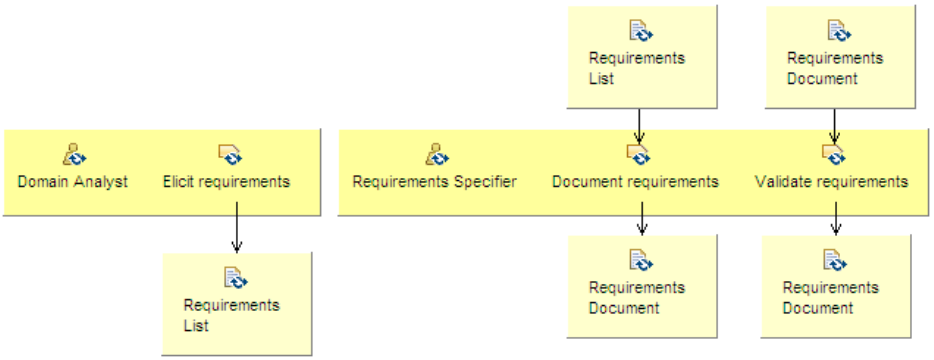


Figure 3.4: Requirements Elicitation detailed activity diagram.

Feature Modeling. Feature modeling was originally introduced by the FODA method and is the activity of modeling the common and variable properties of concepts and their interdependencies in SPLs. Features are essential abstractions that both customers and developers understand.

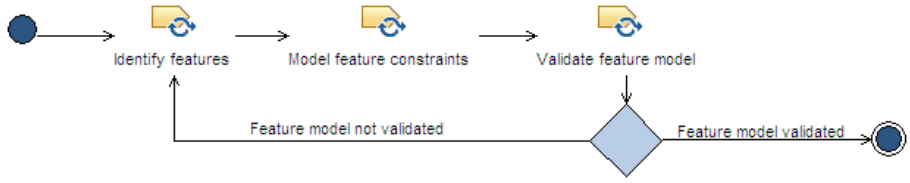


Figure 3.5: Feature Modeling activity diagram.

This activity is composed of three tasks performed sequentially by a Domain Analyst and receives as input requirements artifacts generated in the previous activity. The first task is Identify Features, which refers to the identification of features that are part of the system family. The core of the SPL is characterized by the features that are present in all the products (mandatory features). In addition, there are some features present in only some of the products of the SPL, so they are classified as optional features. And the features that vary from one product to another are the alternative ones. The identified features are then organized into a tree representation called features diagrams, with a specific notation for each variability category (mandatory, alternative and optional). Although FODA proposes a notation for the feature diagram, we adopt the FMP tool (Antkiewicz 2004) and its notation, because of the facility of modeling the diagram.

A feature model refers to a features diagram accompanied by additional information such as dependencies among features, and it represents the variability within a system family in an abstract and explicit way. So, the second task is to model features constraints that express valid combinations of fea-

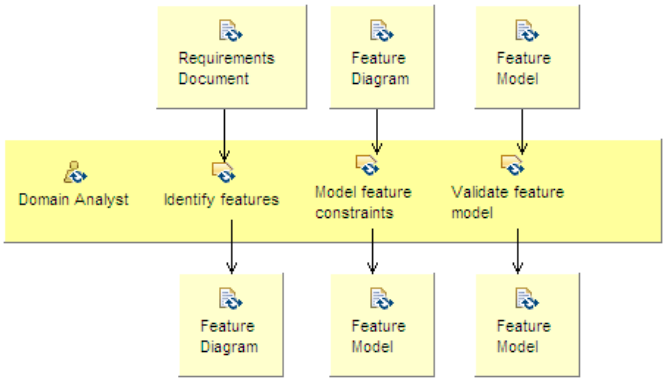


Figure 3.6: Feature Modeling detailed activity diagram.

tures to derive a product. A feature constraint can determine, for instance, if a feature implies the exclusion or inclusion of another.

Both tasks have additional roles that perform them: Domain Specialist and Stakeholder. The interaction between them and the Domain Analyst assures the correct modeling of the feature model. So, the last task of this activity is to validate the feature model (feature diagram plus constraints) with the Domain Specialist and Stakeholder. If they do not approve the feature model, the tasks of this activity are performed again.

Use Case Modeling. In this activity, SPL functional features are described in terms of use cases. A Business Process Analyst is responsible for identifying uses cases and Business Designers should later describe these use cases. So, the first task in this activity is to identify SPL use cases and to create a first version of a use case diagram.

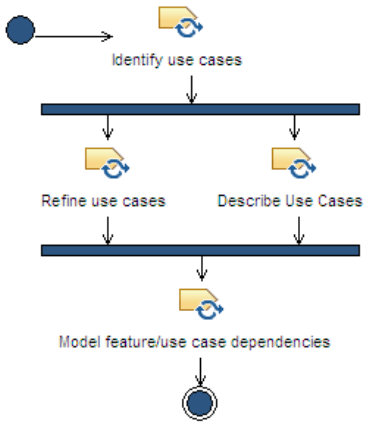


Figure 3.7: Use Case Modeling activity diagram.

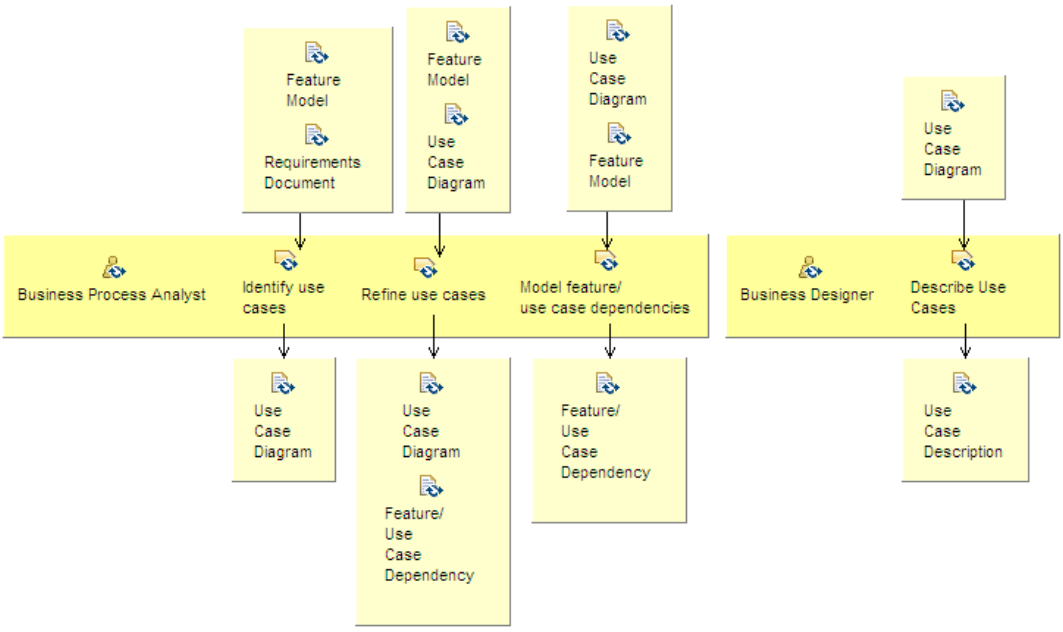


Figure 3.8: Use Case Modeling detailed activity diagram.

Later, the use case diagram should be refined by: (i) refactoring use cases to provide feature modularization; and (ii) adding stereotypes to give variability information. According to the separation of concerns principle mentioned in Section 3.3, each use case should correspond to only one feature. So, the use case diagram must be refactored to fulfill this principle. For instance, if a use case has an optional or alternative part, it must be decomposed into two or more use cases connected by relationships such as extend and include. Moreover, there is a particular kind of feature, named crosscutting feature, which has impact in several use cases/features of the SPL. For these situations, the variable part related to the crosscutting feature is modularized into a specific use case and a crosscut relationship is created between the this use case and the one that is crosscut.

In addition, use case diagrams are adapted to express variability in SPLs, and for that we used the notation proposed by PLUS approach. In the this approach, stereotypes are used to indicate if a use case is part of the SPL kernel and is present in all products ($\ll kernel \gg$), is present in only some products ($\ll optional \gg$) or varies among SPL products ($\ll alternative \gg$). Besides stereotypes, we also use colors in use cases to indicate which feature they are related to. This color indication is used in almost all artifacts to provide a better understanding of features traceability. In parallel to the Refine use cases task, Business Designers should describe use cases helping to identify variable parts on them. Use case descriptions are widely used in the literature, thus we adopted them in our process.

The last task of this activity is to model Feature/Use Case Dependency. A particularity of SPLE is that domain models should contain traceability links from features and variation points in the feature models to their realizations in the other analysis and design models. So, we provide another use case view to map use cases to features: use cases are grouped into features with the UML package notation plus stereotypes. These packages are stereotyped with: (i) ≪*common feature*≫ – represents all mandatory features and groups all kernel use cases; (ii) ≪*optional feature*≫ – represents optional features and groups use cases related to a specific optional feature; (iii) ≪*alternative feature*≫ – represents alternative features and groups use cases related to a specific alternative feature. Furthermore, some of these packages can have another stereotype, ≪*crosscutting feature*≫, which indicates that the features crosscut others. Additionally, a crosscut relationship is created between packages that represent crosscutting features and the ones crosscut by them to express this feature interaction.

Autonomous Requirements

The purpose of the Autonomous Requirements sub-phase is to better understand the domain, by modeling autonomy and pro-active concerns with respect to the current problem domain. This kind of concerns is distinguished because they do not need a user that supervises their execution. Furthermore, they are not precisely described in use cases, and consequently they need a more detailed specification. Agents are an abstraction of the problem space that are a natural metaphor to model pro-active or autonomous behaviors of the system. Therefore, these pro-active and autonomy concerns are identified and specified in models in terms of agents and roles.

So, in order to specify agent features, our process incorporates some activities that generate these models, which correspond to some phases of the Domain Requirements model of PASSI methodology. The Domain Requirements model generates a model of the system requirements in terms of agency and purpose. Table 3.1 depicts the PASSI phases used in our process and extensions we made to them. The activities that encompass the Autonomous Requirements phase are the following.

Domain Ontology Modeling. A domain ontology models a specific domain, or part of the world. It is a formal representation of a set of concepts within a domain and the relationships between those concepts. So, in this activity a Designer is responsible for modeling the domain ontology of the MAS-PL being developed. Based on use case descriptions generated in the Use Case Modeling

Table 3.1: PASSI extensions.

PASSI Phase	Extensions
Agent Identification	Only use cases selected as pro-active or autonomous are distributed among agents An use case can be delegated to more than one agent Use of an arrow outside an agent package to indicate communication between two instances of an agent Use of stereotypes (kernel, alternative or optional) Use of ≪crosscut≫ relationship Use of colors to trace features
Role Identification	Diagrams split according to features Use of UML 2.0 frames for crosscutting features Use of colors to trace features Feature/Agents Dependency model
Task Specification	One diagram per agent and feature Use of UML 2.0 structured activities for crosscutting features Use of colors to trace features

activity, the Designer should identify the ontology concepts, generating a first version of the ontology diagram. The concepts should be modeled taking into account features, by using techniques such as generalization to modularize features. This diagram is represented by UML class diagrams, on which classes represent concepts and their attributes represent slots. Later, this diagram is refined by the identification of relationships among ontology concepts. Finally, a model represented by a table that maps concepts to features is created in order to provide feature traceability. This model enables the selection of the appropriate concepts during the application engineering.

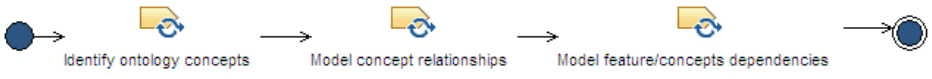


Figure 3.9: Domain Ontology Modeling activity diagram.

Agent Features Identification. In this activity, features that present pro-active or autonomous behavior are identified. These features are classified as agent features, and the agent abstraction is indicated to model this kind of feature. So, we adopt a new stereotype (≪*agent feature*≫) to indicate that a feature is an agent feature. This stereotype is added to feature packages of the Feature/Use Case Dependency model generated in the Use Case Modeling

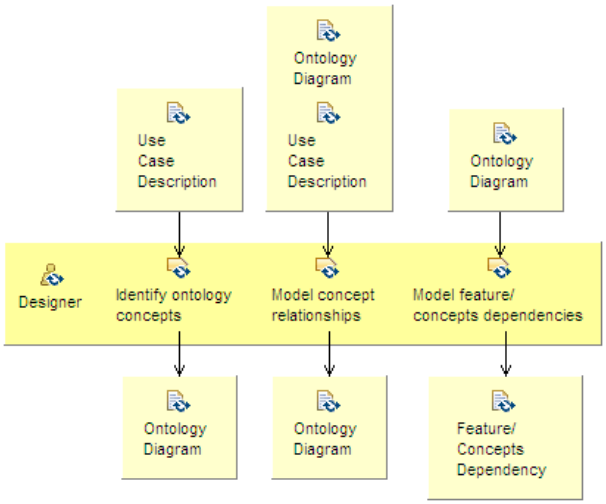


Figure 3.10: Domain Ontology Modeling detailed activity diagram.

activity. This is accomplished in the first task of this activity by a Business Process Analyst.

In addition, among use cases related to agent features, some of them may not present autonomy or pro-activeness. For instance, an agent feature can have two use cases related to it: one that describes collecting news from the web, filter it according to user preferences, generates a report to the user and store it in a database; and another that retrieves the generated report when a user request it. The first use case presents pro-activeness, and the second does not. So, the last does not need to be modeled using the agent abstraction. As a consequence, the second task of this activity is to generate a list of the autonomous and pro-active behavior use cases.

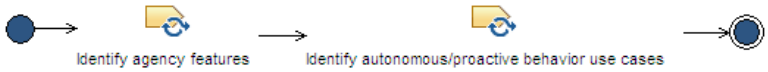


Figure 3.11: Agent Features Identification activity diagram.

Agent Identification. In this activity, responsibilities are attributed to agents, which are represented as stereotyped UML packages. The input of this phase is the autonomous and pro-active behavior use case list generated in the Agent Features Identification activity. This activity is originally from PASSI, but we made some adaptations to it. According to PASSI methodology, all use cases are grouped to be performed by agents; however, we propose that only the selected use cases are considered. So, these use cases are grouped

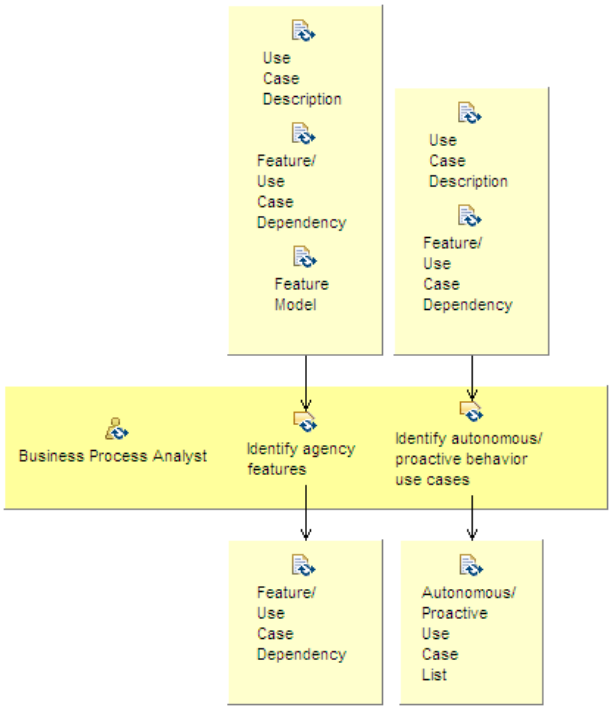


Figure 3.12: Agent Features Identification detailed activity diagram.

into ≪*agent*≫ stereotyped packages so as to form a new diagram. Each one of these packages defines the functionalities that a specific agent should provide.

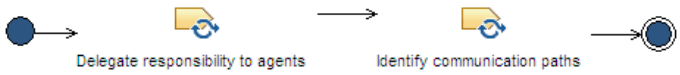


Figure 3.13: Agent Identification activity diagram.

This activity comprises two tasks. A Designer should first identify agents to perform the selected use cases. Later, he models the communication among agents. A communication is represented by a relationship between two use cases, stereotyped with ≪communicate≫. The direction of the relationship arrow represents which agent started the communication.

Use case stereotypes used in the Use Case diagram are still present. As a result, if only optional use cases are given to an agent, this agent will also be optional, for example. Relationships in the Use Case diagram are also preserved. Moreover, we adopted two modifications in the Agent Identification diagram, which we have identified as necessary: (i) a use case can be delegated to more than one agent; and (ii) use of an arrow outside an agent package to indicate communication between two instances of an agent. PASSI does not mention both situations, and does not provide any example that illustrates

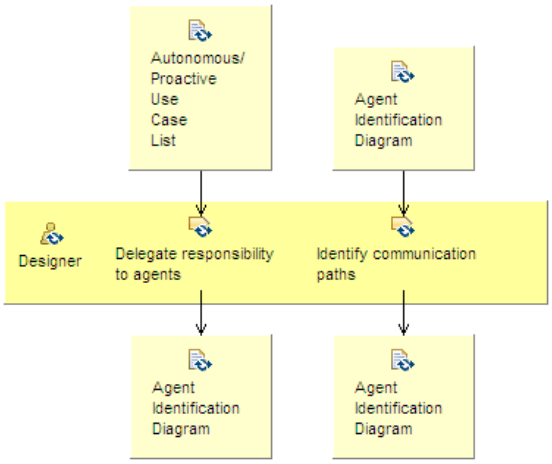


Figure 3.14: Agent Identification detailed activity diagram.

them. These are adaptations that can also be used in the development of MASs.

Agents that will compose the SPL products are not restricted to the agents identified in this phase; additional agents can be introduced in the Domain Design phase (Section 3.4.2).

Role Identification. In MASs, agents can play different roles in different scenarios. In the Role Identification activity, all the possible paths (a "communicate" relationship between two agents) of the Agent Identification diagram are explored. A path describes a scenario of interacting agents working to achieve a required behavior of the system. In different scenarios, agents can play different roles. Agent interactions are expressed through sequence diagrams, named Role Identification diagrams.

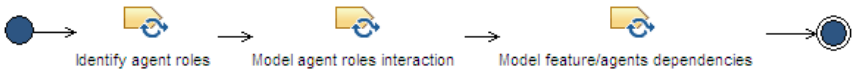


Figure 3.15: Role Identification activity diagram.

This activity aggregates three tasks performed by a Designer. The first two address modeling agent roles and the last addresses feature traceability. The Designer first identifies roles that will be played by agents identified in the previous activity while they are communicating. Later, he must model how agents interact. According to PASSI, each communication path has one diagram; however, when modeling MAS-PLs, if a diagram corresponds to more then one feature, it must be decomposed according to features. An exception for that is when there are crosscutting features. When the Role Identification

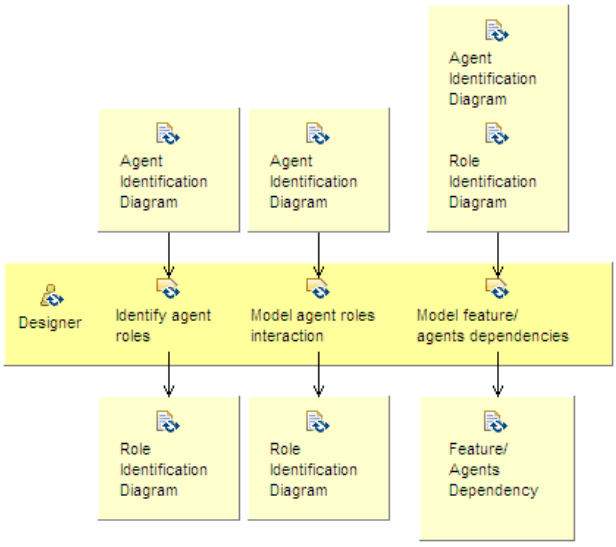


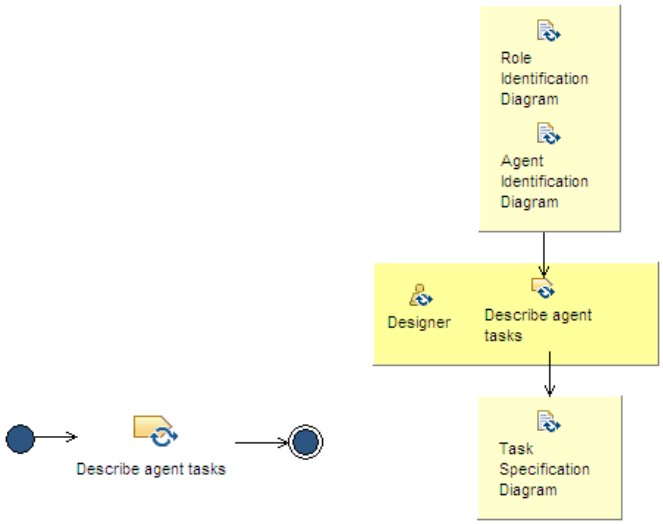
Figure 3.16: Role Identification detailed activity diagram.

diagram has a variable part that corresponds to a crosscutting feature, this part should be delimitated by an UML 2.0 frame. If this variable part corresponds to an optional crosscutting feature, the frame must be of the kind `opt` and its name is `Variation Point:` *`feature name`*. And if this variable part corresponds to an alternative crosscutting feature, the frame must be of the kind `alt` and its name is `Variation Point:` *`feature name`*`(`*`alternative option`*`)`.

After completing the Agent Identification activity and the previous tasks of this activity, agent features of the SPL are now described in terms of agents and their respective roles. So the last task of the Role Identification activity consists of generating a model, the Feature/Agents Dependency model, describing the relationship between features and these agent concepts in the SPL. This model is organized into a tree, in which the root is the SPL, which has children corresponding to agent features. Each feature has agents as its children indicating that these elements must be present in the product being derived if the feature is selected. Agents have roles as children, meaning that the agent play that roles in a certain feature. Some agents and roles can appear as a child of more than one feature, meaning that these elements are present in a certain product if at least one of these features is selected for this product.

Task Specification. In the Task Specification activity, activity diagrams are used to specify the capabilities of each agent. According to PASSI, for every agent in the model, we draw an activity diagram that is made up of two swimlanes. The one from the right-hand side contains a collection of activities symbolizing the agent's tasks, whereas the one from the left-hand side contains

some activities representing the other interacting agents.



3.17(a): Activity diagram. 3.17(b): Detailed activity diagram.

Figure 3.17: Task Specification activity

In these diagrams, we have made three adaptations, some of them were already adopted in other diagrams: (i) instead of drawing only one diagram per agent, we split the diagram according to features; (ii) use of UML 2.0 structured activities to show different paths when there is a crosscutting feature. The structured activity is stereotyped with ≪*variation point*≫ and its name is the feature name. In the case that the feature is alternative, the structured activity contains nested structured activities stereotyped with ≪*alternative*≫ and named with the alternative feature; and (iii) a colored indication showing with which feature the task is related to. The main objective of these adaptations is to provide a better feature modularization and traceability. Splitting the diagram in the way we propose allow the selection of the necessary diagrams during the application engineering according to selected features. The only task that composes this activity refers to modeling Task Specification diagram, which is performed by a Designer.

3.4.2 Domain Design Phase

The main purpose of the Domain Design phase is to define an architecture that addresses both common and variable features of a SPL. Based on SPL analysis models generated on the previous phase, designers should model the SPL architecture, determining how these models, including the variability, are implemented in this architecture. The modularization of features must be taken

into account during the design of the architecture core assets to allow the (un)plugging of optional and alternative features. In addition, there must be a model to map features to design elements providing a traceability of the features. Next we detail the activities that compose the Domain Design phase.

Architecture Description. During SPL design, its architecture is divided into subsystems and their main components. A subsystem is a major component of a system organized by architectural principles. It is a collection of interrelated classes, associations, operations, events, and constraints. A subsystem is a high-level subset of the entire model permitting architecture determination. Decomposing a SPL architecture into subsystems help to reduce the complexity and to allow several design teams to work independently. In SPLE, besides taking into account typical architectural principles such as logical analysis partitioning and design capability ownership, feature modularization may also be considered.

Besides, patterns (Fowler 2002, Buschmann 1996) capture existing, well-proven experience in software development and help to promote good design practice. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. Patterns can be used to construct software architectures with specific properties.

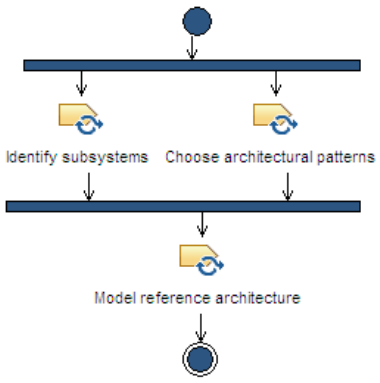


Figure 3.18: Architecture Description activity diagram.

Therefore, the first activity of the Domain Design phase is to define the SPL architecture by decomposing it in subsystems taking into account feature modularization. First, two tasks are performed in parallel: identification of SPL subsystems and choosing architectural patterns. Next, based on the output of both tasks (list of subsystems and selected architectural patterns), a reference architecture is modeled and represented in a UML package diagram. These tasks are performed by a Software Architect.

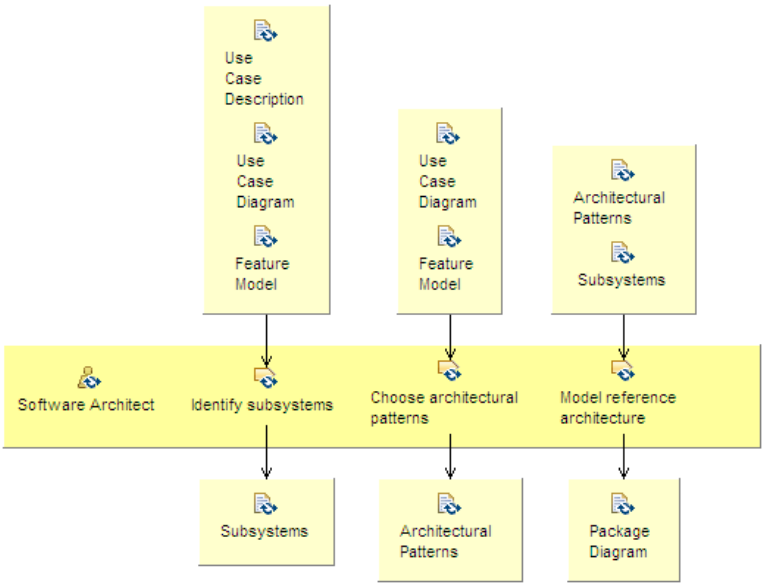


Figure 3.19: Architecture Description detailed activity diagram.

Implementation Platforms Selection. In the literature, there are several frameworks and platforms that help on the development of software systems. Examples are Web Application Frameworks (WAFs) that support web application development, such as Struts (Apache 2008) and JavaServer Faces (Sun 2008); and Spring framework (SpringSource 2008), which provides several modules (e.g. inversion of control and transaction management) that help in the development of complex Java applications. In the context of MASs, several agent platforms have been proposed, such as JADE (Bellifemine 2007), Jadex (Pokahr 2005) and Jason (Bordini 2007).

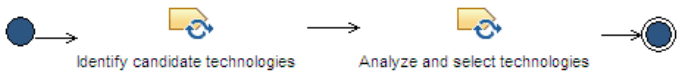


Figure 3.20: Implementation Platforms Selection activity diagram.

Choosing appropriate technologies for designing and implementing a SPL is a very important step on its development. These technologies can facilitate implementing a SPL, for instance Spring framework can help on choosing an implementation for a certain interface of the SPL. In addition, this is an important decision to be made, because it impacts on SPL design. Thus, this activity comprises two tasks performed by a Integrator – the first one is to identify technologies that can be used in the SPL and the second is to choose the technologies that will indeed be used.

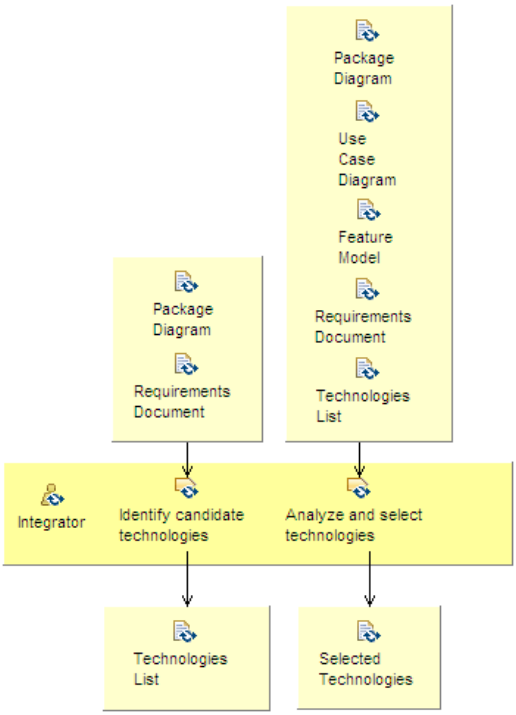


Figure 3.21: Implementation Platforms Selection detailed activity diagram.

Component Modeling. The Component Modeling activity refers to a detailed design of non-agent features. Based on the SPL architecture previously defined, each non-agent feature is modeled first in a component level, and then these components are specified in terms of classes and their attributes and methods.

The tasks of this activity is performed by a Designer. The first task consists of identifying components that will realize non-agent features, and in sequel the structure and dynamic behavior of these components are modeled. Based on several artifacts generated on the previous activities, classes diagrams are designed in order to determine how its elements are structured. This diagram has the purpose of capturing the structural aspects; however it has additional notations to indicate the common and variable elements (≪*kernel*≫, ≪*optional*≫ and ≪*alternative*≫ stereotypes). Moreover, when designing SPL features, techniques, e.g. generalization/specialization and design patterns, should be used to model variable parts of the SPL in order to modularize features.

The dynamic modeling addresses interaction between objects describing how these elements interact with each other. For each use case, the elements that participate in the use case are determined, and the ways in which the elements interact are shown in order to satisfy the requirements described in the use case. Considering that use cases were refactored to be related to only

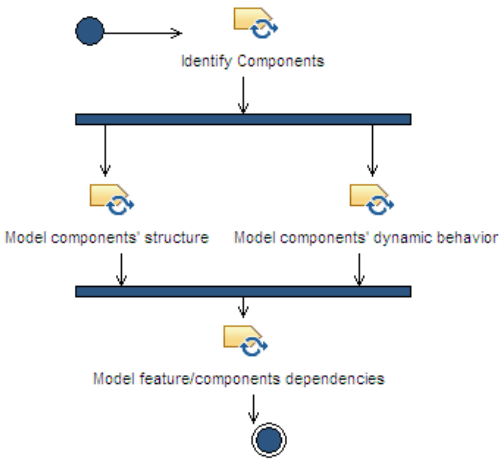


Figure 3.22: Component Modeling activity diagram.

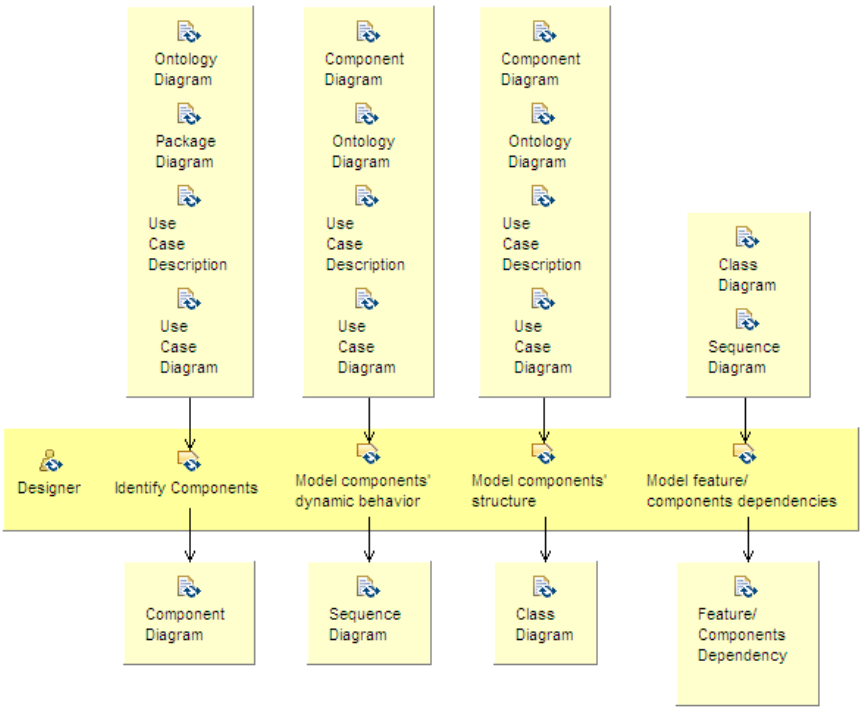


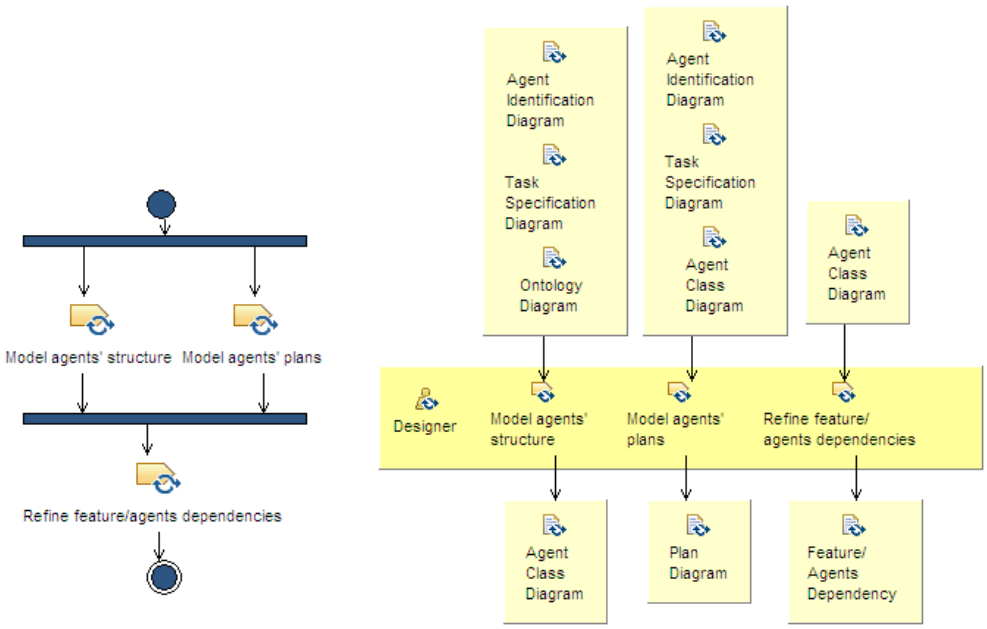
Figure 3.23: Component Modeling detailed activity diagram.

one feature, each sequence diagram is also related to only one feature. For crosscutting features, we include the variability in the sequence diagram to not breaking the flow. However, the behavior related to the crosscutting feature is delimitated by UML 2.0 frames. The last task of this activity has the purpose of providing a feature traceability. A Feature/Component Dependency model should contain information about which components and classes are necessary in a product when a certain feature is selected.

Agent Modeling. Agent features are modeled in two activities of our process, which are performed in parallel and one may contribute with the other. In the Agent Modeling activity, agents with their beliefs, goals and plans are modeled; and in the Agent Society Modeling activity, roles and organizations are modeled. In both activities, instead of using UML, we propose the use of MAS-ML (Silva 2008), a MAS modeling language. It is an UML extension based on the TAO conceptual framework (meta-model) (Silva 2003b). Using the MAS-ML meta-model and diagrams, it is possible to represent the elements associated with a MASs and to describe the static relationships and interactions between these elements. The structural diagrams in MAS-ML are the extended UML class diagram and two new diagrams: organization and role. MAS-ML extends the UML class diagram to represent the structural relationships between agents, agents and classes, organizations, organizations and classes, environments, and environments and classes. The organization diagram models the system organizations and the relationships between them and other system elements. Finally, the role diagram is responsible for modeling the relationships between the roles defined in the organizations.

To address variability in MAS-ML diagrams, we adopted four different adaptations: (i) use of the $\ll$*kernel*$\gg$, $\ll$*optional*$\gg$ and $\ll$*alternative*$\gg$ stereotypes to indicate that diagram elements are part of the core architecture, present in just some products or vary from one product to another, respectively; (ii) use of colors to indicate that an element is related to a specific feature; (iii) model each feature in a different diagram, whenever possible. It is not possible to be done when dealing with crosscutting features; however the use of colors helps to distinguish the elements related to these features; and (iv) introduction of the capability (Padgham 2000) concept to allow do modularization of variable parts in agents and roles. We represented a capability in MAS-ML by the agent role notation with the $\ll$*capability*$\gg$ stereotype. An aggregation relationship can be used between capabilities and agents, and capabilities and roles.

To model agents' dynamic behavior, we use UML sequence diagrams



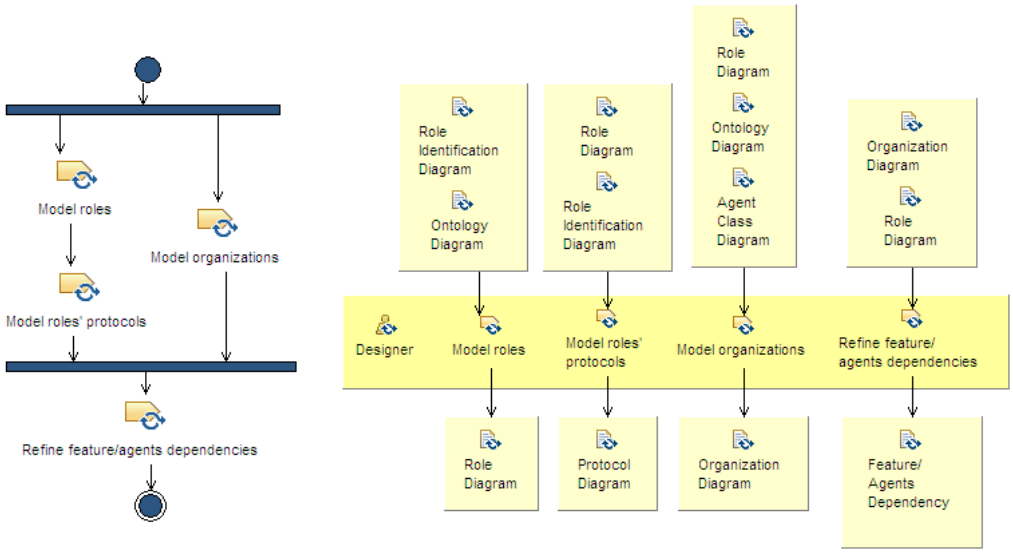
3.24(a): Activity diagram. 3.24(b): Detailed activity diagram.

Figure 3.24: Agent Modeling activity

extended by MAS-ML, which present a set of interactions between objects playing roles in collaborations. The extended version of this diagram represents the interaction between agents, organizations and environments. The only differences in the dynamic modeling activity for single systems and MAS-PLs are: (i) different features are modeled in different sequence diagrams; and (ii) UML 2.0 frames are used to indicate a behavior related to a crosscutting feature, as it was done in the Role Identification activity (Section 3.4.1).

In this activity, there is no task to identify agents because it was already done in Agent Identification activity. However, while modeling these agents or roles, new agents can be identified. Similarly to Component Modeling activity, agents have their structure and dynamic behavior specified, which is performed by a Designer in Model agents' structure and Model agents' dynamic behavior tasks, using MAS-ML and our proposed adaptations. These tasks receive as input the diagrams generated in the analysis of agent features. Finally, the Feature/Agent Dependency model is refined by introducing new agents and capabilities that were identified in this activity.

Agent Society Modeling. Besides modeling agents and their respective beliefs, goals and plans, it is essential in MASs to model agents society with its roles and organizations, and the specification of which agents are going to play roles in organizations. Thus, together with Agent Modeling activity, this activity completes modeling agent features.



3.25(a): Activity diagram. 3.25(b): Detailed activity diagram.

Figure 3.25: Agent Society Modeling activity

Using MAS-ML with the previously presented extensions, a Designer is in charge of modeling agents roles, which are modeled in role diagrams specifying roles structure with beliefs, goals, duties and rights. In addition, variabilities in roles must be modularized using capabilities. Moreover, sequence diagrams are used in order to model roles dynamic behavior, which define protocols of messages exchanged among agents. As it was adopted in other sequence diagrams, UML 2.0 frames indicate variable parts in protocols, which correspond to crosscutting features.

A task performed in parallel to modeling roles is to model agent organizations. The task generate organization diagrams, which show organizations and roles, and indicate agents that play these roles in these organizations. Note that role capabilities can not be played by agents, but only be aggregated another role. In addition, also agent capabilities can not play roles, but only be aggregated another agent. Later, the Feature/Agent Dependency model is again refined to incorporate new roles, organizations, environment and other agent concepts introduced in this activity.

3.4.3 Domain Realization Phase

The purpose of the Domain Realization phase is to implement the reusable software assets, according to the design diagrams generated in the previous phase. In addition, Domain Realization incorporates configuration mechanisms that enable the product instantiation process, which is based on

the documentation and reusable software assets produced during the Domain Engineering process. Two activities compose this phase: Agent Implementation Strategy Description and Assets Implementation.

Agent Implementation Strategy Description. The implementation of software agents is usually accomplished by means of agent platforms, such as JADE (Bellifemine 2007) and Jadex (Pokahr 2005). Agent platforms provide different concepts for implementing agents. For instance, JADE agents are implemented by extending the JADE `Agent` class and behaviors are added to these agents. It is a task-oriented platform. On the other hand, Jadex provides the BDI concepts, given that it follows the BDI architecture. Therefore, Jadex agents are implemented with goals, beliefs and plans. As a consequence, implementing agents based on design models may require a transformation from design concepts to the implementation concepts provided by the target agent platform.

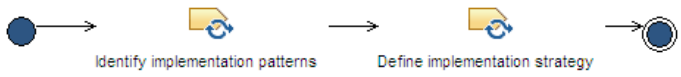


Figure 3.26: Agent Implementation Strategy Description activity diagram.

Thus, the goal of this activity is to define a strategy for implementing agents. An example is to map agent concepts (agents, beliefs and plans) onto object-oriented concepts (classes, attributes and methods) if an object-oriented agent framework is used. Before defining an implementation strategy, patterns of agent implementation are identified in the first task of this activity. Several research work have exploited patterns reuse in MASs (Cossentino 2003, Gonzalez-Palacios 2004), and these patterns show how agents are usually implemented in typical agent platforms. So, based on these patterns and on the agent platform selected on the Implementation Platforms Selection activity, an agent implementation strategy is adopted.

Assets Implementation. In this activity, elements designed in the previous phase are coded in some programming language. So, the first task of this activity is to implement SPL assets, which is performed by an Implementer.

Different implementation techniques can be used to modularize features in the code (Alves 2007), e.g. polymorphism, design patterns, frameworks, conditional compilation and aspect-oriented programming. Moreover, we have explored modularization techniques related to MASs. The Web-MAS architectural pattern is proposed in (Nunes 2008b) in order to integrate software agents

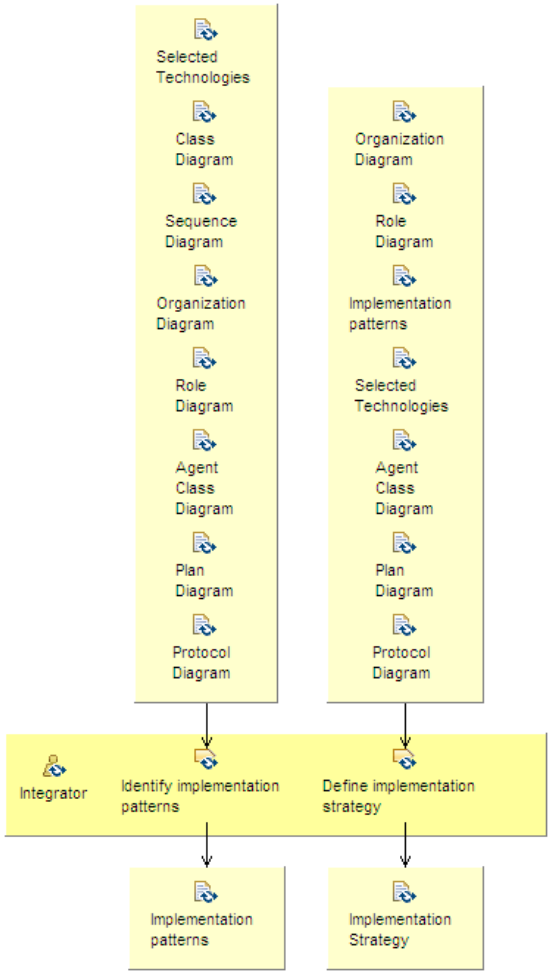


Figure 3.27: Agent Implementation Strategy Description detailed activity diagram.

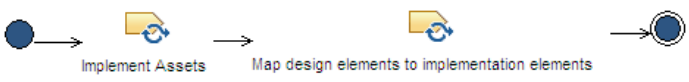


Figure 3.28: Assets Implementation activity diagram.

and web applications in a loosely coupled way, which is detailed in Chapter 5. In (Nunes 2008c), we presented a quantitative study of development and evolution of the EC MAS-PL (Chapter 4), consisting of a systematic comparison between two different versions of this MAS-PL: (i) one version implemented with object-oriented techniques and conditional compilation; and (ii) the other one using aspect-oriented techniques. And, in (Nunes 2009a), we report an empirical study that assesses the modularity of the OLIS MAS-PL (Chapter 4) through a systematic analysis of its releases. The study consists of a comparison among three distinct versions of this MAS-PL, each one implemented with a different technique: (i) Jadex platform and configuration files; (ii) JADE platform and configuration files; and (iii) JADE platform enriched with AOP mechanisms.

Finally, as stated in the previous activity, first-class concepts used in agent-based design may not be represented as implementation elements, and this may force transforming agent design concepts into platform specific ones. For example, roles are concepts that are not present in JADE framework, so they can be implemented using classes structured according to the Role Pattern (Baumer 1997), for instance. Therefore, implementation elements should be mapped into design elements for feature traceability purpose, what is done the second task of this activity.

3.5 Final Remarks

In this Chapter, we presented the domain engineering process we propose. It gives an overview of our approach, detailing its key concepts. Our process is founded on the integration of three well-succeeded approaches in the context of SPL and MAS: PLUS method; PASSI methodology; and MAS-ML modeling language. Besides integrating this technologies, new adaptations and extensions are proposed.

The process is structured according to the SPEM and is basically based on three main levels: phases, activities and tasks. Each one of the activities was detailed, by indicating tasks to be performed, roles that should perform them, and their inputs and outputs as well.

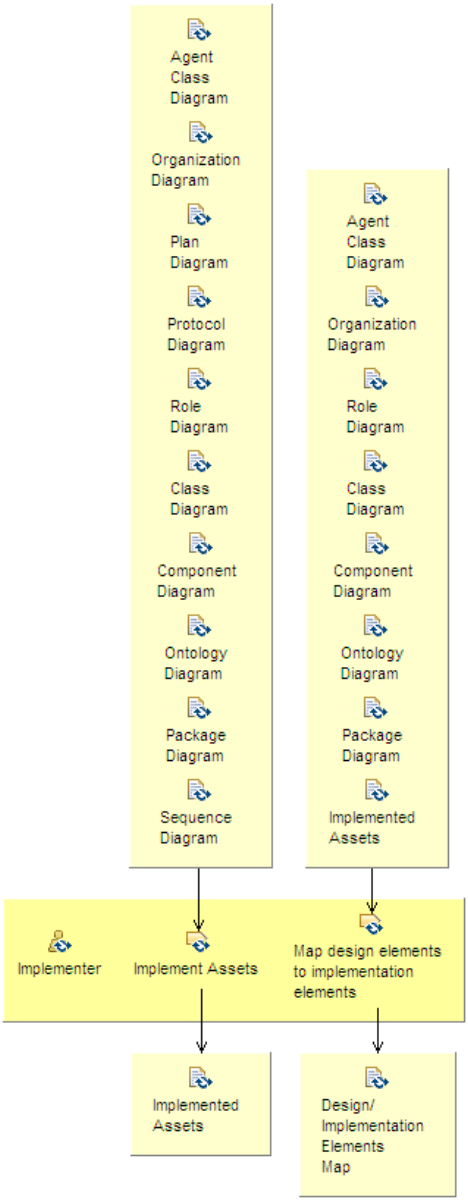


Figure 3.29: Assets Implementation detailed activity diagram.

4
Case Studies

This Chapter describes our experience with the development of MAS-PLs. We detail two developed case studies, both for the web domain: the ExpertCommittee (Section 4.1), a MAS-PL of conference management systems; and the OLIS (Section 4.2), a MAS-PL of systems that provide several personal services for users, such as Events Announcement. These product lines were developed using two different approaches: extractive for the ExpertCommittee (the MAS-PL was build based on a set of different system versions); and reactive for the OLIS (the MAS-PL was incrementally build). Event though no case study adopts a pro-active development approach, our process can also be applied in this situation.

4.1
ExpertCommittee Case Study

In this Section, we present our first case study, the ExpertCommittee (EC). This case study consists of developing a conference management system, which we have evolved over time, consequently it has several versions. This kind of system, first proposed in (Ciancarini 1998), has been widely used to the elaboration and application of agent-oriented methodologies. After developing the first version of the system (its core), some agent features were incorporated to it. The evolution of the EC was accomplished in an *ad-hoc* way, therefore we did not consider features modularity, leading to a tangled and spread code into all system components. Consequently, this was also reflected in the system design. Further, using an extractive approach (Section 2.1) of the SPL development, we build the EC MAS-PL. Based on the different EC versions, we developed an architecture in such way that we could semi-automatically derive all the different previously released versions of the system or any valid combination of its features.

So, first we give an overview of the EC evolution (Section 4.1.1), detailing which functionalities composed each one of the system versions. Next, we show how we transformed the several versions of the EC system into a MAS-PL using our approach (Section 4.1.2). Our process allowed to analyze the commonalities

and variabilities among the system versions, and to document and model (agent) features. As a result, we had to make some refactoring in the existing code in order to modularize the features. Besides helping on the construction of the MAS-PL, the use of our approach generates a documentation that: (i) provides a better understanding of the EC and its features; (ii) facilitates a posterior evolution of this MAS-PL to incorporate new (agent) features.

4.1.1 ExpertCommittee Overview

The first version of the EC consists of a typical web-based conference management system, which supports the paper submission and reviewing processes from conferences and workshops. In sequel, we evolved this version of the system, incorporating new optional and alternative features. Most of them are agent features, which automate some tasks previously performed by users and are mainly related to the specification and implementation of user agents. These changes implied the addition of new agents and roles, as well as changes in their behavior.

The EC first version provides functionalities to support the complete process of conference management, such as: (i) create conferences; (ii) define conference basic data, program committee, areas of interest and deadlines; (iii) choose areas of interest; (iv) submit paper; (v) assign papers to be reviewed; (vi) accept/reject to review a paper; (vii) review paper; (viii) accept / reject paper; (ix) notify authors about the paper review; and (x) submit camera ready. Each of these functionalities can be executed by an appropriate user of the system, such as, conference chair, coordinator, program committee members and authors. In the following versions, software agents were introduced into the EC architecture, providing autonomous behavior. We consider autonomous behavior actions that the system automatically performs and previously needed human intervention. The main aim of these new features was to help the tasks assigned to all the system users by giving them a user agent that addresses mainly the following functionalities: (i) deadline and pending tasks monitoring; and (ii) automation (or semi-automation) of the user activities.

Table 4.1 summarizes the different versions of the EC system. Each new version was implemented based on the previous one. The first version was built without any autonomous behavior, in other words, without software agents. The second version of the EC system added a feature to the system that is related to autonomous behavior, it monitors the system and automatically sends some important message notifications. Thus, in this second version, we

have used the agent abstraction and AOSE techniques to allow the introduction of this new agent feature in the system. Software agents were used to model and implement this autonomous behavior, including an agent representing each user of the system playing different roles. The third version provided new functionalities (non-agent features) to the system related to the Reviewer role; however the user agent had to be modified by the inclusion of this new role for supporting messages notifications. The following versions also included agent features and, finally, the last version changed a feature previously included (message notifications), resulting in an alternative feature.

Table 4.1: The EC versions.

Version	Description
Version 1	Typical web-based application that represents our MAS-PL core. It has the mandatory features that support the conference management process.
Version 2	Addition of message notifications to the system users through email.
Version 3	Addition of the Reviewer role and the functionalities related to it: accept/reject review and review paper.
Version 4	Addition of automatic suggestion of conferences to the authors.
Version 5	Addition of automatic paper distribution to committee members.
Version 6	Addition of notifications when a deadline is nearly expiring.
Version 7	Addition of task management.
Version 8	Change of the type of message notifications (SMS).

4.1.2 Transforming the EC System Versions into a MAS-PL

In this Section, we show how the EC system versions were refactored to comprise a MAS-PL. We detail the steps taken to accomplish that, which are according with the activities specified by our process. Some activities were omitted, e.g. components modeling, because we focused on detailing agent features. Several artifacts are presented, which were generated while modeling the EC MAS-PL.

Domain Analysis

In the Domain Analysis phase, we defined the scope of the EC MAS-PL based on all the different versions of EC system. The mandatory, optional and alternative features of the product line were organized into the EC feature model (Figure 4.1), and these features were specified in terms of use cases,

and additional diagrams (agent identification, role identification and task specification diagrams) for the agent features.

Operational Requirements. In this sub-phase, we analyzed all versions of EC in order to determine their commonalities and variabilities. Later, functionalities provided by features of the product line are specified in a use case diagram and use case descriptions. Finally, there is a mapping between the features and the use cases of the MAS-PL.

Requirements Elicitation. The first activity to be performed is Requirements Elicitation. In this activity, we analyzed all the EC system versions in order to identify requirements of all system versions. Our requirements document is a table on which columns are the EC versions and rows are the requirements, and we marked the cells indicating which requirements are part of each one of the EC versions.

Feature Modeling. Based on the generated requirements document, we have verified what is common to all of EC versions and what varied from one to another. Later, these commonalities and variabilities (i.e. features) were organized into an hierarchical form aggregated with additional information, consisting in a feature model. Figure 4.1 depicts the EC feature diagram that resulted from this analysis. It contains several mandatory features, which were present in all system versions; and optional and alternative features, which introduced autonomous behavior in some of the system versions.

The *User Management* is a mandatory feature that provides the management of user accounts in the system and is used by all the users of the EC. However, most of the EC functionalities are accomplished by a user playing a specific role. Therefore, we have another feature called *Role*, which is composed of the five different possible roles – Coordinator, Chair, Committee Member, Reviewer and Author – and each one has associated functionalities/features as its children. The Reviewer role is optional, but once it is selected, its children are mandatory.

The other optional features are agent features, which adds autonomous behavior to the EC. Some of them (*Automatic Distribution* and *Conference Suggestion*) are variabilities associated with another feature (*Assign Paper* and *Register Paper*, respectively). The feature *Send Message Notifications* is composed of four other features: three of them add new functionalities to the system, indicating which kind of messages is sent

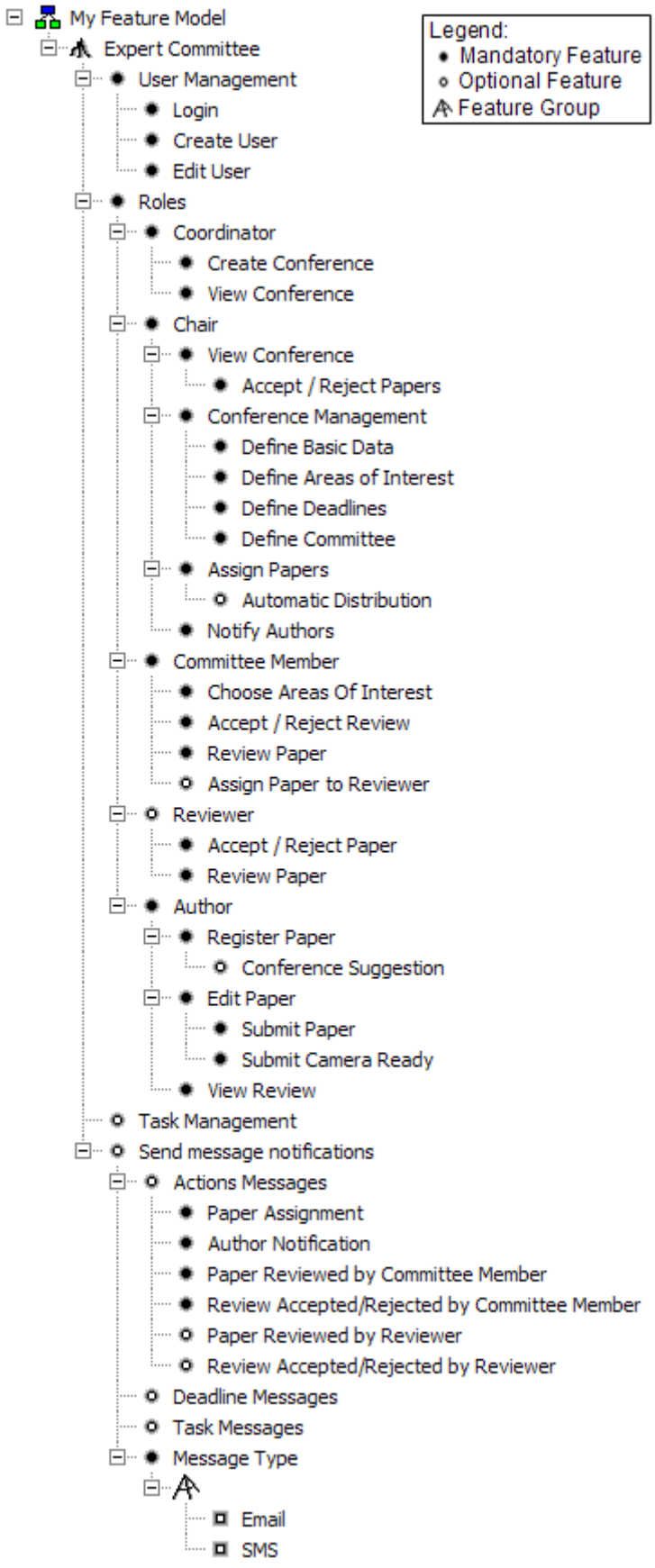


Figure 4.1: EC Feature Diagram.

by it; and the other one (*Message Type*) configures what kind of message (e-mail or SMS) is used.

Besides the feature diagram, the feature model is composed by two constraints[1]:

- `(//paperReviewedByReviewer) -> (//reviewer);`
- `(//reviewAcceptedRejectedbyReviewer) -> (//reviewer);`

Both of them indicate that if a certain feature is selected (*Paper Reviewed by Reviewer* and *Review Accepted/Rejected by Reviewer*) the feature *Reviewer* must be selected. This indicates a dependency relation from one feature to another.

Use Case Modeling. The next activity to be performed is to model the (mandatory, optional and alternative) features provided by the product line in terms of use cases. In Figure 4.2, we show the EC use case diagram. It can be seen in the diagram that there are several types of actor in the system. When a user is registered and logged in the system, he/she can play different roles (represented as actors that are a specialization of the *User* actor), and each one of these roles can interact with the system participating of specific use cases. Additionally, there is the actor *Time* whose purpose is to trigger some use cases according to a time event, such as when a deadline date is reached.

The EC use case diagram illustrates several mandatory (kernel) and optional use cases, and they are colored according to the feature that they are related to. The mandatory use cases are present in all system versions, such as *Create Conference* and *Assign Papers*. The optional use cases, e.g. *Assign Review to Reviewer* and *Suggest Conferences*, are present only when the feature related to the use case is selected. A particularity of use case modeling for SPLs is that a use case should be decomposed into two or more use cases to separate a part of the use case that corresponds to another feature. For instance, the *Manage Tasks* and *Send Task Massage* use cases could be described in only one use case if we were designing a single system. However, the behavior described in the *Send Task Massage* use case is optional and a possible product derived from the product line can have the *Manage Tasks*, but not the *Send Task Massage* use case. Therefore, the particular behavior of sending messages was modularized into one specific use case and connected to the *Manage Tasks* use case by a `extend` relationship.

[1]The constraints are written using the FMP Tool (Antkiewicz 2004) notation.

According to the feature model previously presented, the *Reviewer* role is optional. Even though only one new use case is inserted when this role is selected (*Assign Review to Reviewer*), the presence of this feature impacts other use cases: *Monitor Action Execution*, *Manage Tasks* and *Send Deadline Messages*. So, this variation is specified in specific use cases and connected to the appropriate use cases by a `crosscut` relationship.

Despite the fact that there is an alternative feature in the feature model, there is no alternative use case. This happens because this alternative feature (*Message Type*) does not impact in the functionalities described in the use cases.

After modeling the EC features and use cases, the latter should be mapped onto the former by grouping the use cases into stereotyped packages (Figure 4.3). This allows one know which use cases are related to which feature. All the mandatory use cases are grouped into a ≪*common feature*≫ stereotyped package, called *ExpertCommittee*. The optional feature *Role* is represented by a package whose content is four use cases. Some of these use cases impact other features of the system, as a consequence there is a `crosscut` relationship between the package *Role* and the packages that represent the feature that was crosscut. As the *Role* feature is a crosscutting feature, it is stereotyped with ≪crosscutting feature≫.

Autonomous Requirements. Next we detail most of the activities of the Autonomous Requirements sub-phase. We focus on the description of the modeling of two agent features: *Automatic Distribution* and *Task Massages*.

Agent Features Identification. In the EC MAS-PL, it was identified six different agent features: *Automatic Distribution*, *Conference Suggestion*, *Task Management*, *Task Massages*, *Deadline Messages* and *Action Messages*. The use cases that present pro-active or autonomous behavior, e.g. *Assign Papers Automatically* and *Suggest Conferences*, are grouped into packages named with the feature they are related to, stereotyped with ≪*agent feature*≫. Due to the pro-active and/or autonomous behavior of these features, the agent abstraction is appropriate to model them. Indeed, these features were implemented using the agent technology in the EC system versions.

Agent Identification. The first task is to identify the agents of the system (analysis level), by delegating the use cases to the agents of the system.

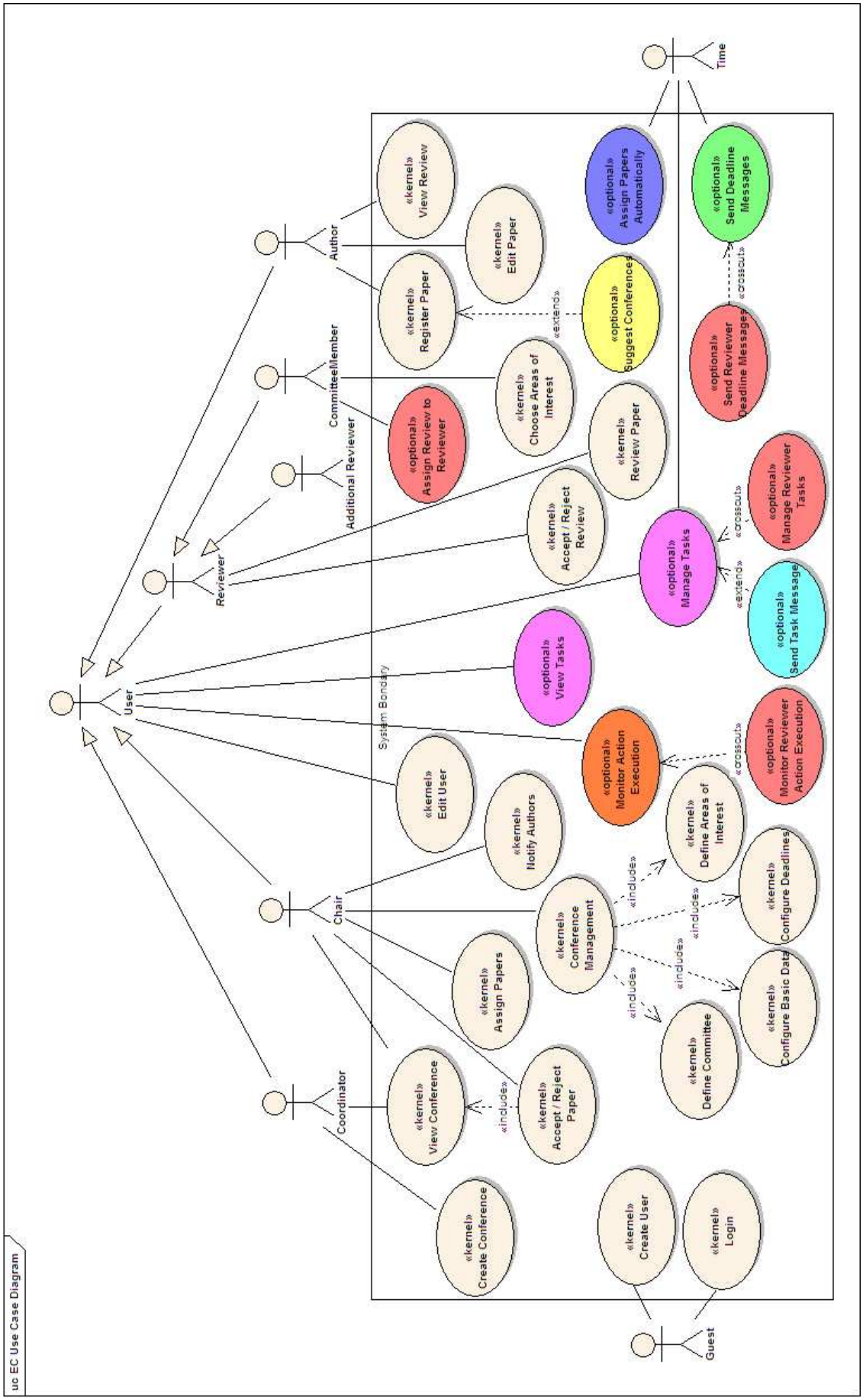


Figure 4.2: EC Use Case Diagram.

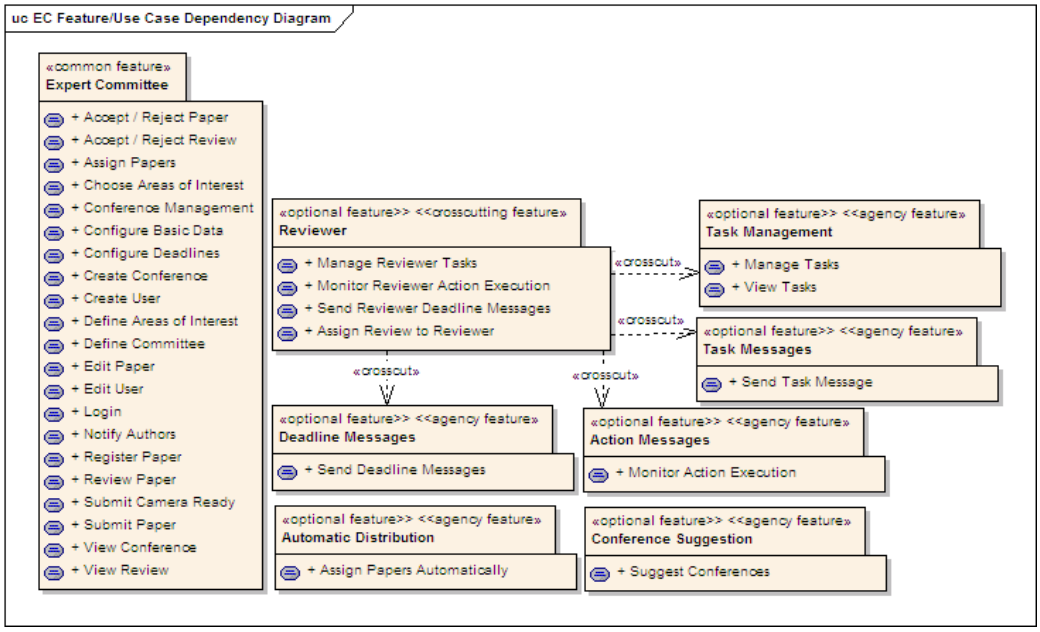


Figure 4.3: EC Feature/Use Case Dependency Diagram.

In the agent identification diagram (Figure 4.4) there are four different agents: (i) `UserAgent` – represents the users and acts on their behalf; (ii) `DeadlineAgent` – responsible for monitoring the deadlines and triggering appropriate actions when a deadline is expiring or has expired; (iii) `TaskAgent` – responsible for handling tasks; (iv) `NotifierAgent` – responsible for sending messages for the users. As the Figure 4.4 illustrates, some use cases are performed by several agents, e.g. *Send Deadline Messages*, meaning that these agents will collaborate to accomplish the behavior described in the use case. This collaboration appears as `communicate` relationship among use cases.

Role Identification. After identifying the agents of the system, the agent roles should be identified. So, we explored the "communicate" paths of the Agent Identification Diagram. Figures 4.5 and 4.6 present two Role Identification Diagrams – Assign Papers Automatically and Send Deadline Messages respectively. The agent roles that the `UserAgent` can play are equivalent to the roles that a user can play in a conference: Coordinator, Chair, Committee Member, Reviewer and Author. The diagrams illustrate the messages exchanged among the agents playing the specified roles.

Figure 4.5 shows how the system automatically distribute papers to be reviewed by committee members. When the `DeadlineAgent` playing the role `DeadlineMonitor` detects that the `SUBMIT_PAPER` deadline expired, it notifies the `Chair` of the conference about that. Then the

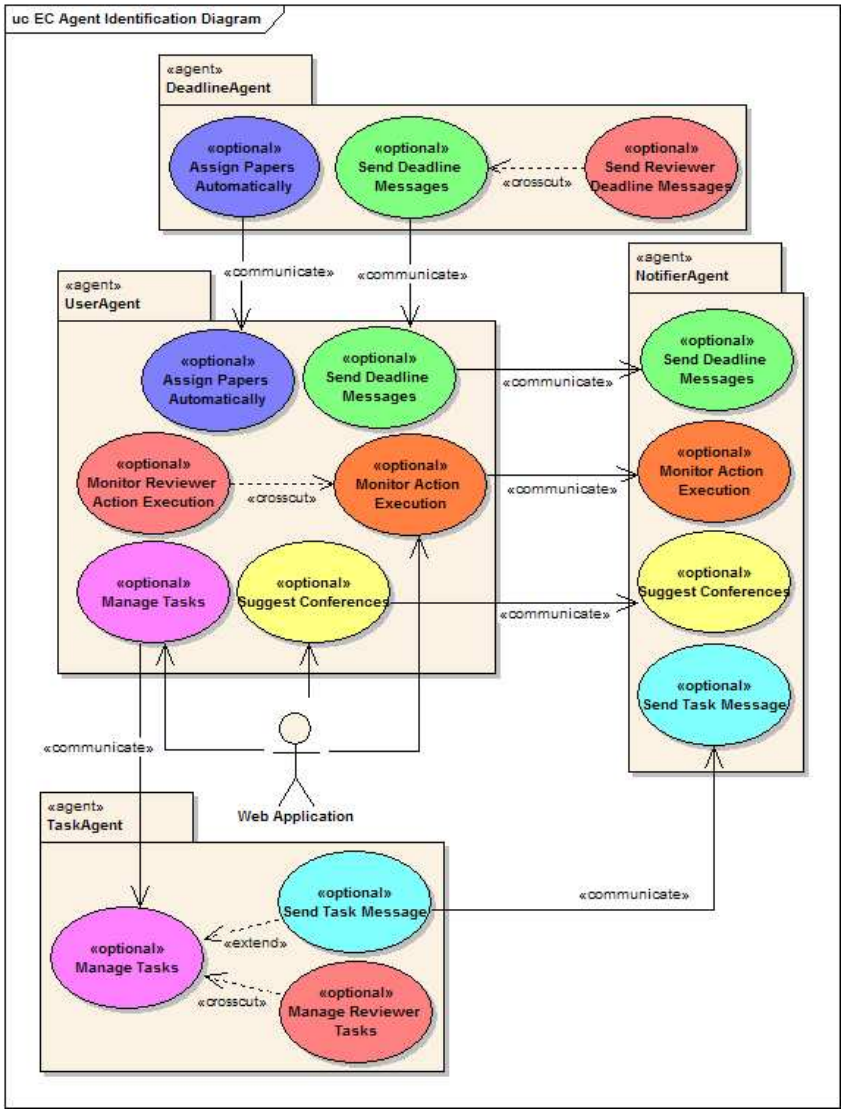


Figure 4.4: EC Agent Identification Diagram.

`Chair` distributes the papers and notify the `CommitteeMember`s. Later, a `CommitteeMember` can reject to review the paper, so it notifies the `Chair`, who delegates the paper review to another `CommitteeMember`. In the diagram, there are more then one instance of the `CommitteeMember`, indicating that the role is being played by different agents.

The diagram of Figure 4.6 presents the collaboration of the roles/agents to send message notifications about deadlines that are going to expire. There, the `DeadlineAgent` is playing a different role, the `DeadlineReminder`. This diagram has an UML 2.0 frame, named *Variation Point: Reviewer*, to indicate a specific behavior of the *Send Reviewer Deadline Messages* use case, which crosscuts the *Send Deadline Messages* use case.

After identifying the agents and roles that are part of the EC MAS-PL

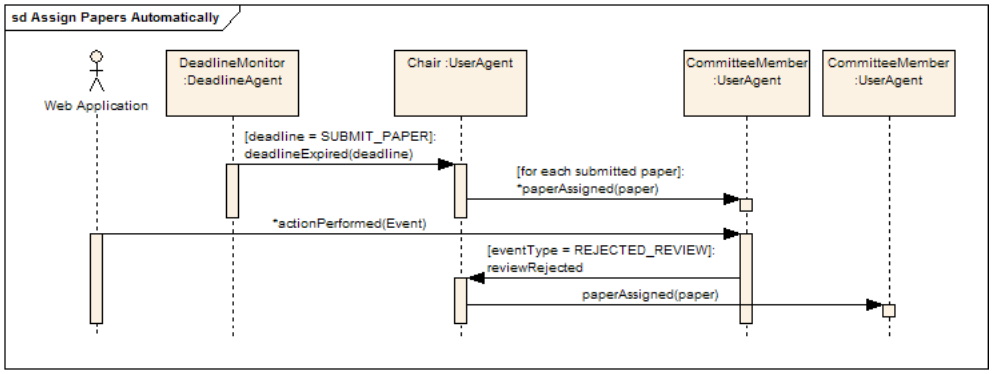


Figure 4.5: Role Identification Diagram - Assign Papers Automatically.

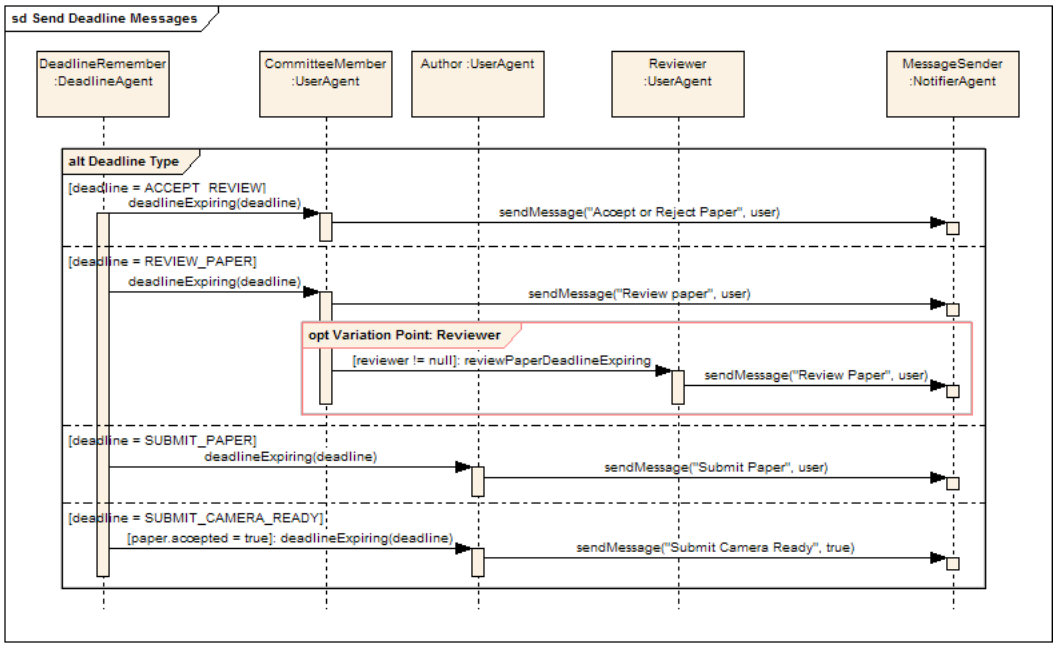
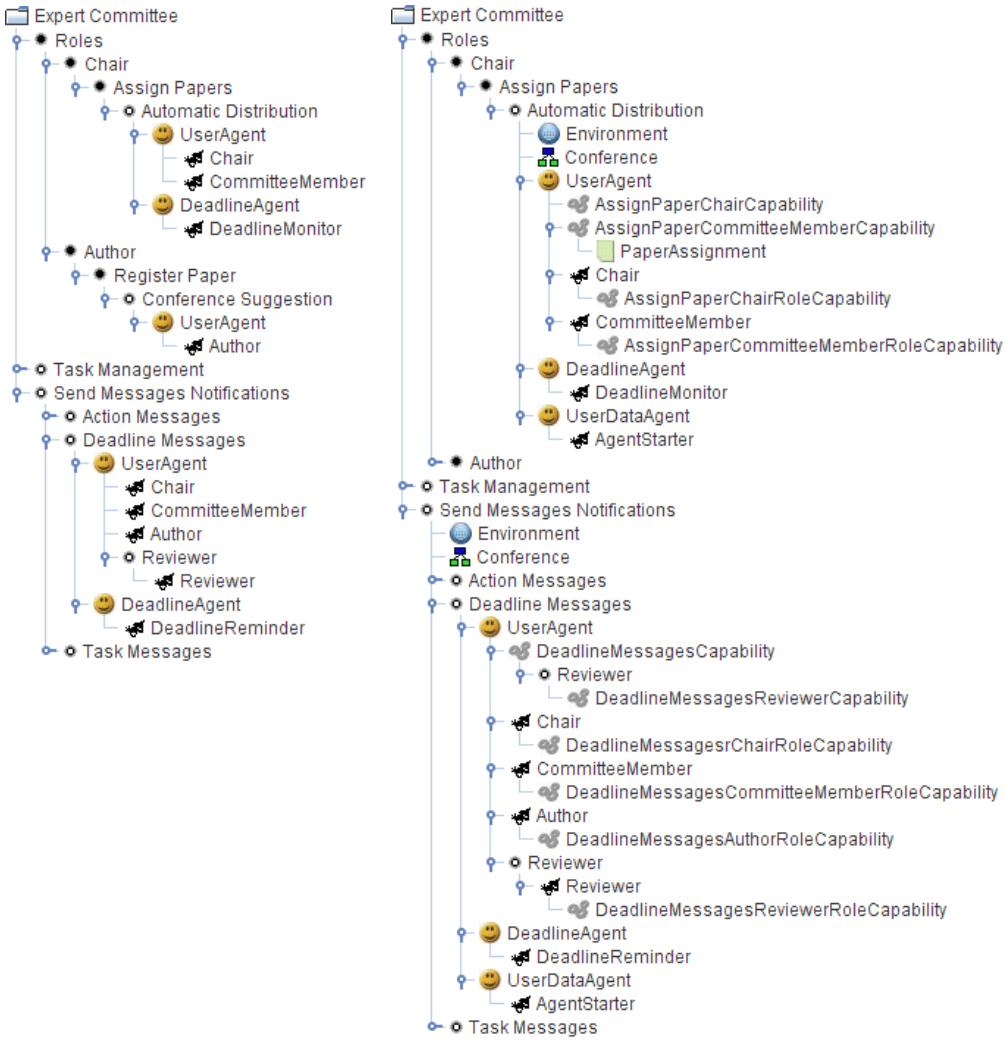


Figure 4.6: Role Identification Diagram - Send Deadline Messages.

at analysis level, these elements were mapped to features. In the EC, the `UserAgent` has five roles: `Coordinator`, `Chair`, `CommitteeMember`, `Author` and `Reviewer`, being the last one optional. These roles were defined based on the roles that people play in a conference, consequently, these roles act in several scenarios, which are related to different features. Therefore, when modeling the dependency between features and agents, it can be seen (Figure 4.7(a)) that the same roles, e.g. `Chair`, appear related to more than one feature. The roles will be present in a product if one of the features related to them are selected.

In addition, Figure 4.7(a) shows that in the *Deadline Messages* feature there is a role (`Reviewer`) that does not depend only on this feature, but also on the *Reviewer* feature. As a result, the `Reviewer` role will be present in a product if both features are selected.



4.7(a): Analysis Level 4.7(b): Design Level

Figure 4.7: EC Feature/Agent Dependency Modeling (Partial).

Task Specification. The next activity is to specify the tasks that the agents should perform independently of the role that they are playing. There is one diagram per feature and agent. Figures 4.8 and 4.9 shows the activities to be performed by the `UserAgent` to assign the papers automatically and to send deadline messages. The activities are colored according to the feature that they are related to. In Figure 4.9, there are some activities that are optional because they are present only if the *Reviewer* feature is selected. So, these activities are grouped into a ≪*variation point*≫ stereotyped structured activity.

Domain Design

The next phase of the domain engineering process is to design an architecture that supports the variability analyzed in the previous phase.

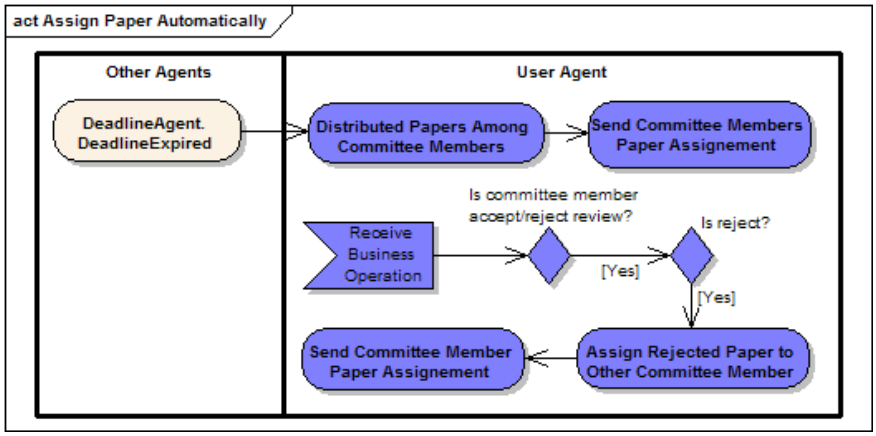


Figure 4.8: Task Specification Diagram - Assign Paper Automatically.

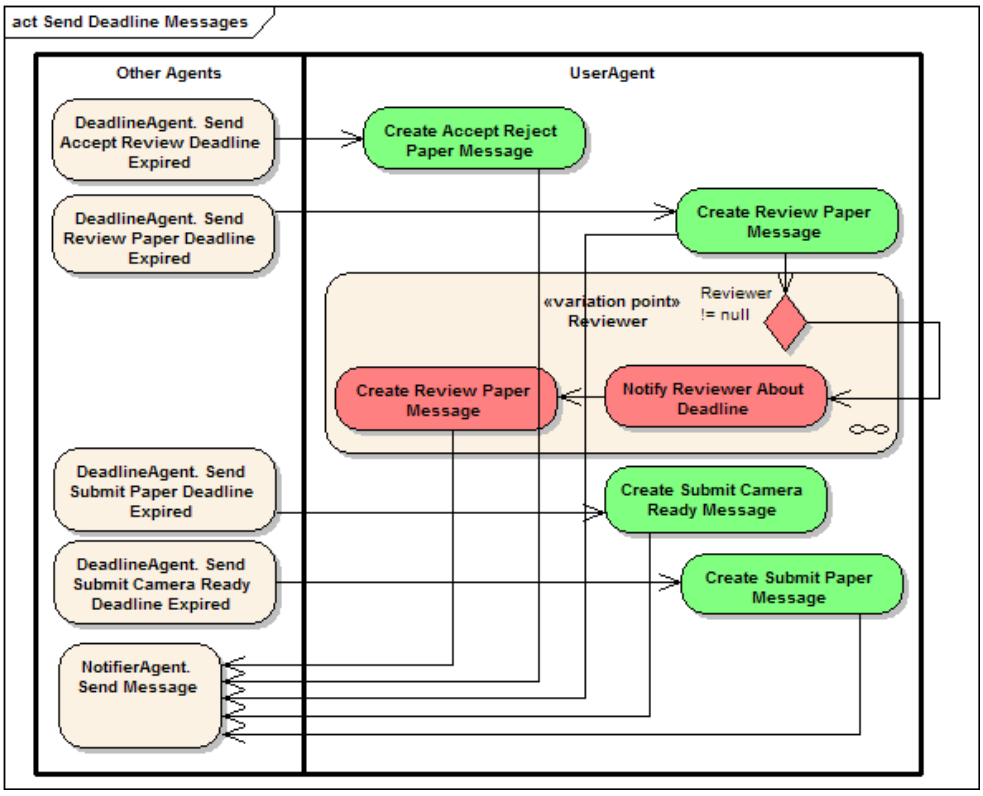


Figure 4.9: Task Specification Diagram - Send Deadline Messages.

Architecture Description and Implementation Platforms Selection. The products derived from the EC MAS-PL are web applications that have agents incorporated to their architecture to provide autonomous and pro-active behavior. The architecture is structured according to the Layer architectural pattern (Buschmann 1996) and comprises the following layers:

(i) `GUI` – this layer is responsible for processing the web requests submitted by the system users;

(ii) `Business` – is responsible for structuring and organizing the business services provided by the EC; and

(iii) `Data` – aggregates the classes of database access of the system, which was implemented using the Data Access Object (DAO) (Alur 2001) design pattern.

The typical web applications architecture was extended by the incorporation of two components, which allow the addition of agents. These components are: (i) business layer monitor – monitors the `Business` layer to capture the business operation processes that were performed and propagate them to the agent layer; (ii) agent layer – MAS that provides the autonomous and pro-active behavior.

Plenty of technologies were selected for the EC, some of them came from the previous EC implementations. The programming language used was Java and several frameworks were used to implement its architecture: (i) the Struts[2] framework is a flexible control layer based on standard technologies; (ii) the transaction management of the business services was implemented using the mechanisms provided by the Spring[3] framework; (iii) the Hibernate[4] framework was used to make persistent the objects in a MySQL[5] database; and (iv) the JADE (Bellifemine 2007) framework was used as the base platform to implement agents.

Agent Modeling and Agent Society Modeling. Besides the agents specified in the Domain Analysis phase, one more agent was introduced in the architecture: the `UserDataAgent`. This agent provided support for the other agents, for instance, creating new `UserAgent`s. Details about each agent that are part of the EC architecture are listed below:

[2]http://struts.apache.org/.
[3]http://www.springframework.org/.
[4]http://www.hibernate.org/
[5]http://www.mysql.org/.

`UserDataAgent:` this agent receives notifications when new users are created in the database. When it happens, it creates a new user agent that will be the representation of the user in the system. The initial execution of the `UserDataAgent` demands the creation of an user agent for each user already stored in the database;

`UserAgent:` each user stored in the system has an agent that represents him/her in the system. This is the autonomous behavior, agents performing actions on behalf of the users. The `UserAgent` was designed in such a way that it can dynamically incorporate new roles. Each agent role perform specific actions according to the role that the user plays in the conference, such as chair, coordinator and author;

`DeadlineAgent:` this agent is responsible for monitoring the conference deadlines. This monitoring serves two purposes: (i) to notify the `UserAgent`s when a deadline is nearly expiring; and (ii) to notify the `UserAgent`s when a deadline has already expired;

`TaskAgent:` this agent is responsible for managing the user tasks. It receives requests for creating, removing and setting the execution date of tasks. The requests are made by the `UserAgent`s;

`NotifierAgent:` this agent receives requests from other agents to send messages to the system users. In the current implementation, it sends these messages through email and SMS.

The EC MAS-PL has several MAS-ML diagrams as output of the Agent Modeling and Agent Society modeling activities. Features that do not use the agent abstraction were modeled in the Components Modeling activity in typical class diagrams, not using the elements extend in MAS-ML. Each agent feature has three different diagrams: (i) class diagram (Figure 4.10); (ii) role diagram (Figure 4.11); and (iii) organization diagram (Figure 4.12). The Figures illustrates a partial view of the *Automatic Distribution* feature diagrams.

There are some elements in the diagrams that are shared by different features, e.g. `Environment` and the `UserAgent`. So, these elements have no specific color and are composed only by common goals, beliefs and so on. The elements colored with blue are exclusively related with the *Automatic Distribution* feature. In order to modularize the common and variable parts of the elements, the variable parts were modularized into capabilities. For instance, the `UserAgent` has a capability aggregated to it, which provides the beliefs, goals and plans to enable the `UserAgent` to assign papers automatically.

Because the EC core is a web application with no agent features, all the agents are optional.

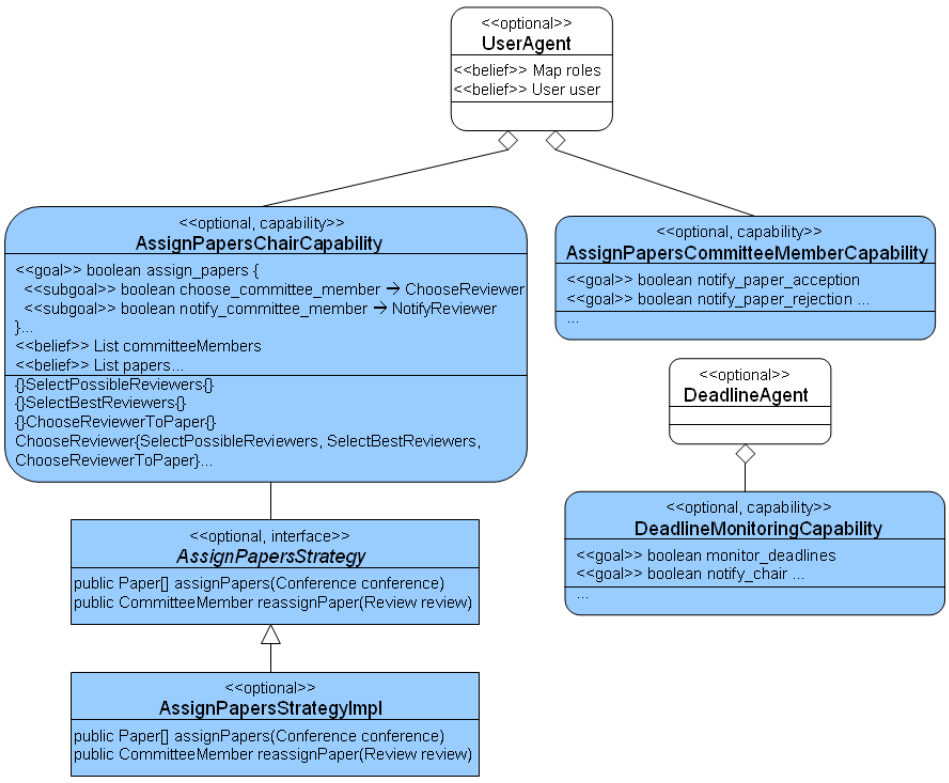


Figure 4.10: EC Class Diagram.

To complete the design of the EC MAS-PL agents, their dynamic behavior was be modeled. The dynamic behavior of agents are the execution of plans performed by them. So we modeled these plans, for instance `ChooseReviewer` and `NotifyReviewer`, using the sequence diagrams extended by MAS-ML. These plans were defined for one specific feature (*Automatic Distribution* in this case), so they are related exclusively to it. However, features that were crosscut by the *Reviewer* feature, e.g. *Send Deadline Messages*, have a variable behavior, which is optional. So, an UML 2.0 frame is used to specify this variability, just like in the Role Identification diagram (Figure 4.6).

When designing the EC MAS-PL, new design agent elements were introduced in the architecture, besides the already identified elements in the Domain Analysis phase. So, these elements were mapped to features, generating a refined Feature/Agent Dependency model depicted in Figure 4.7(b). These elements include the `Environment`, the `Conference` (main organization), `PaperAssignment` (object role), new agents and capabilities.

The design of the EC introduced several capabilities in the MAS-PL architecture. This happened because the same roles and agents are involved in several features, consequently their variabilities were modularized

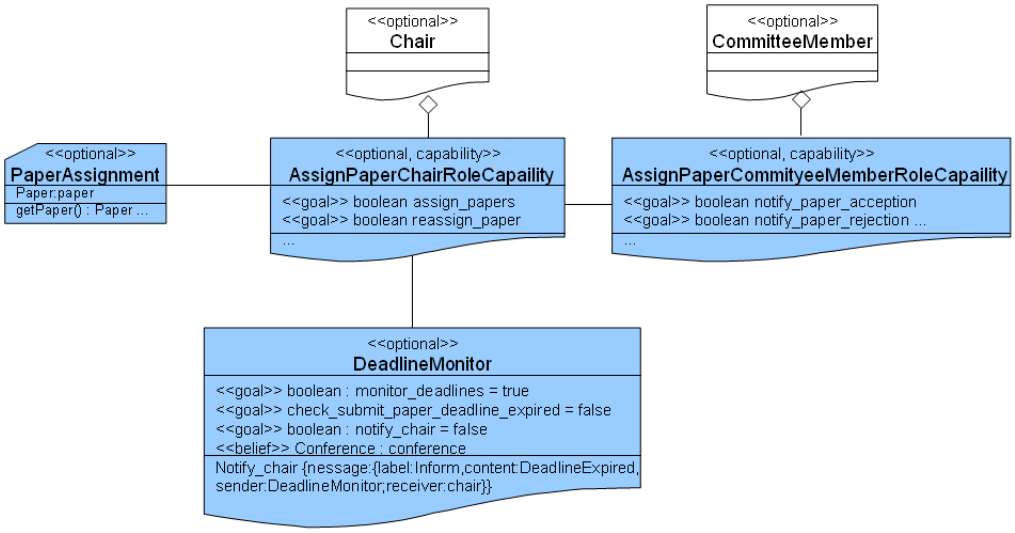


Figure 4.11: EC Role Diagram.

into capabilities. Figure 4.7(b) shows that the `Chair` role has several capabilities associated to it, such as `AssignPapersChairRoleCapability` and `DeadlineMessagesChairRoleCapability`. So, during the product derivation process, the `Chair` role will have the appropriate capabilities according to the selected features.

Domain Realization

This Section details the execution of the activities that compose the Domain Realization for the EC. Most of the existing code of the different versions of the system was reused to develop the code assets of the EC. Some implementation techniques were used in order to support the variability, which was not considered when the versions were being implemented. Next, we describe details of the strategy adopted for implementing agents. In addition, we present details of the EC MAS-PL assets implementation and how we mapped design elements to implemented assets.

Agent Implementation Strategy Description. Given that the JADE framework does not provide several agent concepts, such as roles and environment, there is the need to define implementation strategies to implement these concepts using the ones provided by the framework. So we solved this problem by following these principles:

– *Implementing the environment as an agent.* The `EnvironmentAgent` monitors the EC system by observing the execution of specific business services. These monitored events of the EC system represent the environment in which the `UserAgent`s are situated. Each `UserAgent` is specified

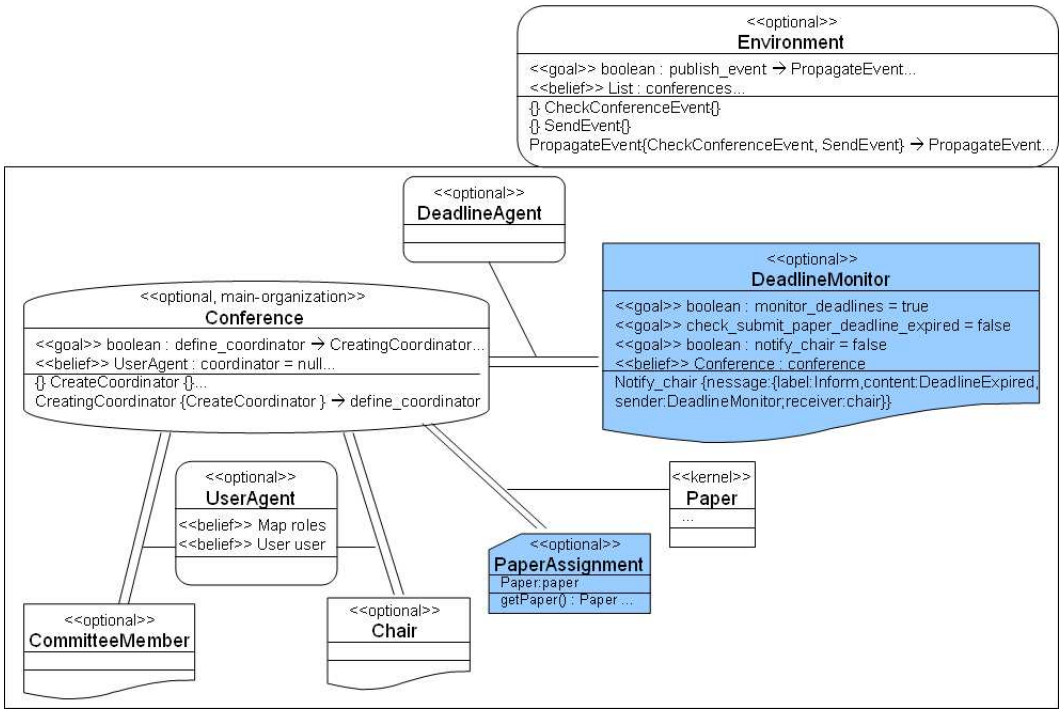


Figure 4.12: EC Organization Diagram.

to perceive changes in the environment and make actions according to them. The environment agent was implemented using the Observer design pattern (Gamma 1995). When it is initialized, it registers itself as an observer of the services that compose the `Business` layer. These services are observable objects that allow the observation of their actions. That means that services not only execute requested methods, but they also notify their respective observers for each call of the system business methods. The only observer in our implementation is the `EnvironmentAgent`, whose aim is to notify the other agents of the MAS-PL about the system changes;

– *Use of the Role Pattern to implement agent roles.* The Role Object pattern (Baumer 1997) models context-specific views of an object as separate role objects, which are dynamically attached to or are removed from the core object. This pattern was mainly used to provide a base implementation of the `UserAgent`s whose behavior can be incremented by attaching new roles (such as chair, author, committee, reviewer) to be played by these agents. The concept of role in this pattern and in MAS-ML is different. In MAS-ML, a role is like an interface with beliefs and goals that specifies if an agent, which can play a certain role, can play. In this pattern, the role provides the behavior, i.e. methods or plans, for the agent play the role;

– *Mapping agent to object-oriented concepts.* We mapped agent concepts,

e.g. beliefs and plans, to object-oriented concepts: agents' beliefs were mapped to attributes and plans were implemented as JADE behaviors. JADE framework does not provide a reasoning engine, therefore we had to manually implement the plan selection. As a result, the agents' goals were not necessary to be explicit at the code, but they are implicit in the algorithms that select plans.

– *Incorporating capabilities into agent/roles.* As JADE does not provide the capability concept, we could not modularize some agent variabilities into capabilities. Therefore, we incorporated the capabilities content into the agents/roles and used *conditional compilation* as implementation technique. An alternative option to this implementation technique could be the use of polymorphism and design patterns; however we adopted conditional compilation because it required less refactorings in the legacy code.

Assets Implementation. The EC MAS-PL was implemented by applying a series of refactorings in the system versions. The analysis and design phases were accomplished based on the existing "products"; however some techniques were used in order to modularize variable parts and to make the (un)plugging of optional features possible. Besides conditional compilation, this techniques included the use of object-oriented design patterns and the use of Spring configuration files, which allows the injection of dependencies inside the variable points of the EC MAS-PL architecture. It improves the capacity to produce and compose different configurations (products) of the SPL, and it also enables the automatic product derivation by means of model-based tools, such as: software factories (Greenfield 2004), generative programming (Czarnecki 2000), GenArch (Cirilo 2008a, Cirilo 2008b), pure::variants[6]. In an automatic product derivation process, the application engineer can generate a configuration (product) of the SPL by only selecting and choosing the features that are going to compose your product.

The customization of the MAS-PL components using the Spring framework was accomplished by specifying a configuration file that aggregates different options of configuration of the MAS-PL, such as: (i) different functional features of the conference management base system (edit_user, paper_distribution) and respective properties; (ii) the different agents and the respective plans; and (iii) the different agent roles and respective plans. These important elements of the EC were modeled using the bean abstraction of the Spring framework,

[6]http://www.pure-systems.com/

which offers a model to build applications as a collection of simple components (called beans) that can be connected or customized using dependency injection and aspect-oriented technologies. Spring container uses a XML configuration file to specify the dependency injection on application components. This file contains one or more bean definitions which typically specify: (i) the class that implements the bean, (ii) the bean properties and (iii) the respective bean dependencies. Listing 4.1 illustrates a fragment of our MAS-PL configuration file.

Listing 4.1: XML Configuration File.

```
<bean id="ExpertCommitteeConfig"
    class="br.puc.maspl.config.ExpertCommittee">
  <property name="optionalFeatures">
    <list>
      <value>edit_user</value>
    </list>
  </property>
  <property name="roles">
    <map>
      <entry key="CHAIR"><ref bean="Chair" /></entry>
       ...
      <entry key="AUTHOR"><ref bean="Author" /></entry>
    </map>
  </property>
  <property name="agents">
    <map>
      <entry key="deadlineAgent"><ref bean="DeadlineAgent" /></entry>
      <entry key="taskAgent"><ref bean="TaskAgent" /></entry>
      <entry key="notifierAgent"><ref bean="NotifierAgent" /></entry>
    </map>
  </property>
  <property name="services">
    <list>
      <ref bean="AuthorService" />
       ...
      <ref bean="TaskService" />
    </list>
  </property>
</bean> <bean id="Chair" class="br.puc.maspl.config.Role">
  <property name="optionalFeatures">
    <list>
      <value>automatic_paper_distribution</value>
    </list>
  </property>
</bean>
```

The EC MAS-PL was implemented using JADE framework, so agents, roles and plans were developed by extending the classes provided by the framework. Figure 4.13 shows the EC implementation elements, such as classes, interfaces, mapped to the design elements (partial view). It presents some of the techniques detailed in the previous Section: (i) the `Environment` was

implemented as an agent with a JADE behavior that publishes events to other agents; (ii) the `UserAgent` was implemented with the `UserAgent` interface and the `UserAgentCore` and `UserAgentRole` classes, which are according to the Role Pattern; and (iii) some design elements were implemented by code fragments that are delimitated by tags that allow the conditional compilation.

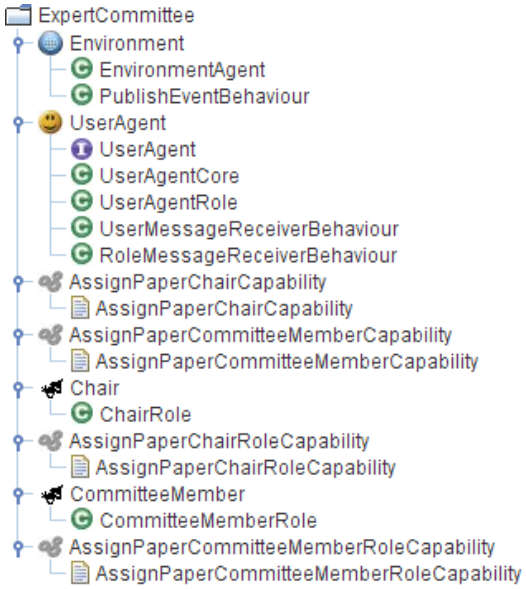


Figure 4.13: EC Design/Implementation Elements Mapping.

4.2 OnLine Intelligent Services (OLIS) Case Study

The second case study developed is the OnLine Intelligent Services (OLIS). It is a MAS-PL that allows the derivation of web applications that provide personal services for the users, such as calendar. The product line was developed using a reactive approach: first, we developed a SPL with an alternative feature, and incrementally we added new optional agent features to this SPL. These new versions of the SPL are characterized as a MAS-PL, because they address new agent features, which allow providing different customized products that have software agents on their architecture. One main difference between the EC (Section 4.1) and the OLIS MAS-PL development is that the EC had different *system* versions and later these versions were used as a base for the MAS-PL construction; on the other hand, OLIS was initially built as a product line, and the *product line* evolved, through the incorporation of new features.

In this Section, we start by introducing the OLIS MAS-PL detailing the features that compose it (Section 4.2.1). Second, we present the modeling of this MAS-PL using our development process (Section 4.2.2). The activities of

the process were performed each time that a new feature was added to the product line, resulting at the end in the artifacts presented here.

4.2.1 The OLIS MAS-PL Overview

The OLIS case study is a MAS-PL of web applications that provide several personal services to users, developed using a reactive approach. The first version of the product line is composed mainly by two services: the Events Announcement and the Calendar Services. However, the OLIS was designed in such a way that it can be evolved to incorporate new services without interfering the existing ones. Additionally, OLIS provides an alternative feature: the event type, thus derived products can deal with generic, academic or travel events.

The Events Announcement service allows the user to announce events to other system users through an events board. The events have some common basic attributes, such as subject, description, location, city, start and end dates, frequency that it happens, and some specific attributes according to the event type. The Calendar service lets the user to schedule events in his/her calendar. Besides the information of the events published in the events board, calendar events have a list of users that participate in it. Announced events can be imported into the users' calendar.

After developing the first version of OLIS SPL, we have identified that new autonomous behavior features could be introduced to automate some tasks in the system. So, we evolved it, incrementally adding new features, which take advantage of the agent technology. The new features incorporated to the OLIS first version are:

Events Reminder. The user configures how many minutes he/she wants to be reminded before the events, and the system sends notifications to the user about events that are about to begin;

Events Scheduler. When a user adds a new calendar event that involves more participants, the system checks the other participants' schedule to verify if the event conflicts with other events. If so, the system suggests a new date for the calendar event that is appropriate according to the participants schedule;

Events Suggestion. When a new event is announced, the system automatically recommends the event after checking if it is interesting to the users based on their preferences. The system also checks if the weather is going

to be appropriate according to the place type where the event is going to take place; and

Weather Service. This is a new user service. It provides information about the current weather conditions and the forecast of a location. This service is also used by the system to recommend announced travel events.

The evolution of the OLIS was accomplished by the introduction of software agents and agent roles on product line architecture. In the next Section, we describe all the activities performed for the development of the OLIS MAS-PL, and some important generated artifacts.

4.2.2 Developing OLIS with our Process

This Section presents and details some performed activities and main artifacts generated during the development of the OLIS MAS-PL. Some of the artifacts, such as the feature model and the use case diagram, were evolved each time the product line was incremented with a new feature.

Domain Analysis

In this Section, we describe the domain analysis some of the activities activities performed in the OLIS MAS-PL development.

Operational Requirements. Next, we detail the Feature Modeling and Use Case Modeling activities performed in the Operational Requirements subphase and present artifacts generated as well.

Feature Modeling. Figure 4.14 illustrates the feature diagram containing the features of the OLIS MAS-PL. It presents the available services provided by product line – *Calendar*, *Event Announcement* and *Weather* features; besides the *User Management*. Additionally, there are some optional features that represent customizations of these services. Furthermore, there is the *Event Type* feature, which is alternative, indicating which kind of event the derived product handles.

Complementing the information provided by the feature diagram, there are some constraints to ensure that a feature selection is valid ((Antkiewicz 2004) notation):

- `if (//generic) then not (//eventSuggestion) else true();`

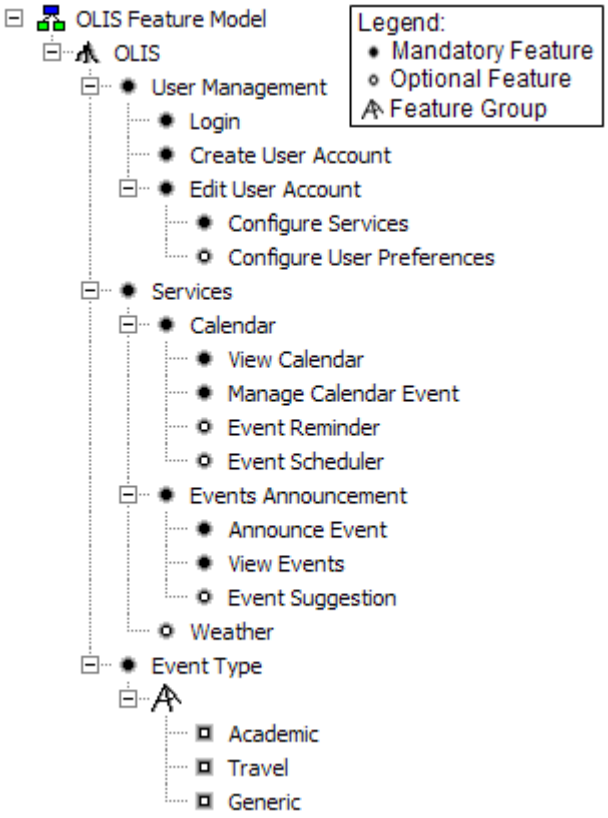


Figure 4.14: EC Feature Diagram.

- `(//eventSuggestion) -> (//configureUserPreferences);`
- `if not (//eventSuggestion) then not (//configureUserPreferences) else true();`
- `(//eventSuggestion && //travel) -> (//weather);`

These constrains express the following: (i) the *Event Suggestion* feature can only exist when the *Event Type* is either *Academic* or *Travel*; (ii) and (iii) the feature *Configure User Preferences* can be present if and only if the *Event Suggestion* is present; and (iv) the *Weather* feature must be present if the *Event Suggestion* feature and the *Travel* event type are selected.

Use Case Modeling. The OLIS use case diagram is depicted in Figure 4.15. Most of the use cases are mandatory and stereotyped with ≪kernel≫, e.g. *View Calendar* and *Announce Event*. There are optional use cases that represent the customization of the services. Some of them extend a functionally, such as the *Suggest Event*, modeled by a use case connected to the *Create Calendar Event* use case by a `extend` relationship. Others, for instance the *Event Reminder*, are directly connected to the user.

The *Event Type* feature impacts (crosscut) in several use cases. As the Figure 4.16 shows, there are four different use cases for the *Academic* and *Travel* features. These use cases crosscut some mandatory and optional use cases.

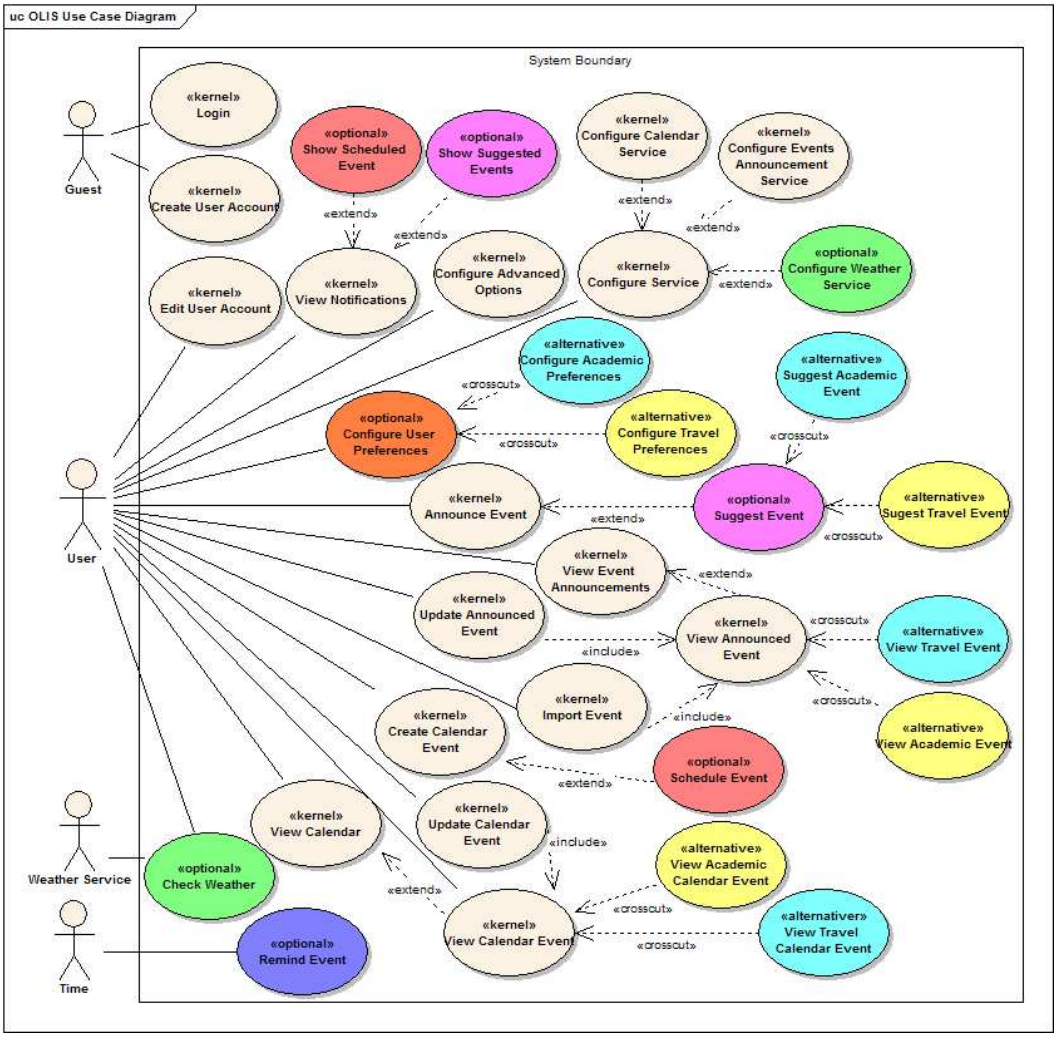


Figure 4.15: OLIS Use Case Diagram.

To complete the Use Case Modeling activity, the use cases should be mapped to the features. Figure 4.16 shows the OLIS Feature/Use Case Dependency model, indicating this relation. All the optional features of the system present a pro-active or autonomous behavior, so they were classified as agent features. Besides, the alternative feature Event Type crosscuts other features of the system, therefore there is the crosscut relationship between the package of this feature (stereotyped with ≪crosscutting feature≫) and the packages crosscut by it.

Autonomous Requirements. Four features were identified as agent features in the OLIS MAS-PL while performing the Agent Features Identification

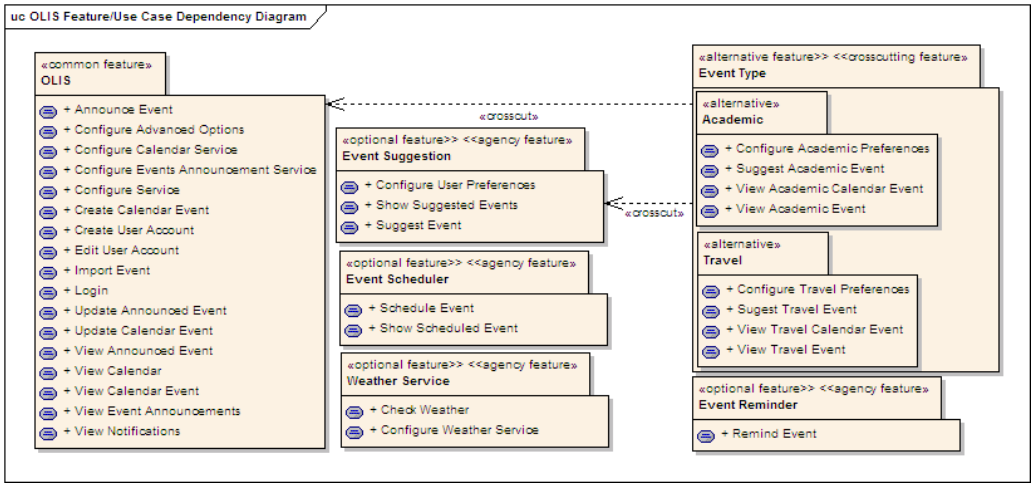


Figure 4.16: OLIS Feature/Use Case Dependency model.

activity. These features were stereotyped with ≪*agent feature*≫ in the OLIS Feature/Use Case Dependency model (Figure 4.16). As these features present an autonomous or pro-active behavior, they were be better detailed and specified in order to understand their behavior. The following activities were performed to accomplish this.

Agent Identification. Figure 4.17 depicts the agents identified in the Agent Identification activity. There are only two agents: the `UserAgent` and the `WeatherAgent`. Besides providing the weather, all other functionalities are accomplished by the `UserAgent`, which acts on behalf of the user. There are arrows from some use cases to themselves – the *Schedule Event* and the *Suggest Event* – indicating that there is a communication between two different instances of the `UserAgent`. This diagram also shows an alternative variability in the `UserAgent`: Academic and Travel events in the *Suggest Event* use case. As all use cases are optional, both agents are optional.

Role Identification. The `UserAgent` is responsible for performing almost all agent features of the MAS-PL; however this agent executes actions while playing different roles. Figure 4.18 presents the role identification diagram for the *Event Suggestion* feature, through which we have identified the agent roles. It shows an interaction between two different instances of the `UserAgent` playing the roles `EventAnnouncer` and `EventClient`. The *Event Suggestion* feature is crosscut by the *Event Type* feature, so a variation point can be seen in the diagram as a consequence of this feature interaction. The result is a communication between the `EventClient` and the `WeatherProvider` role played by the `WeatherAgent`.

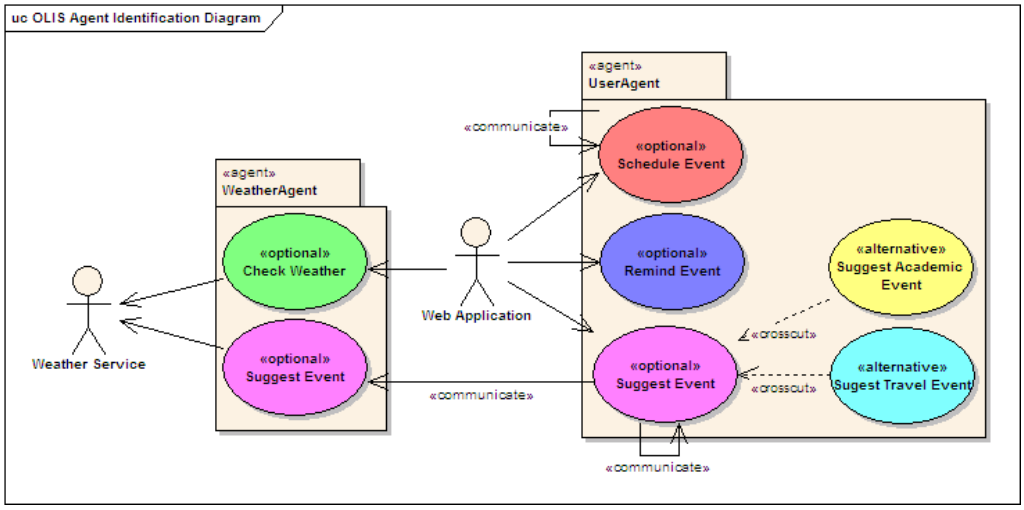


Figure 4.17: OLIS Agent Identification Diagram.

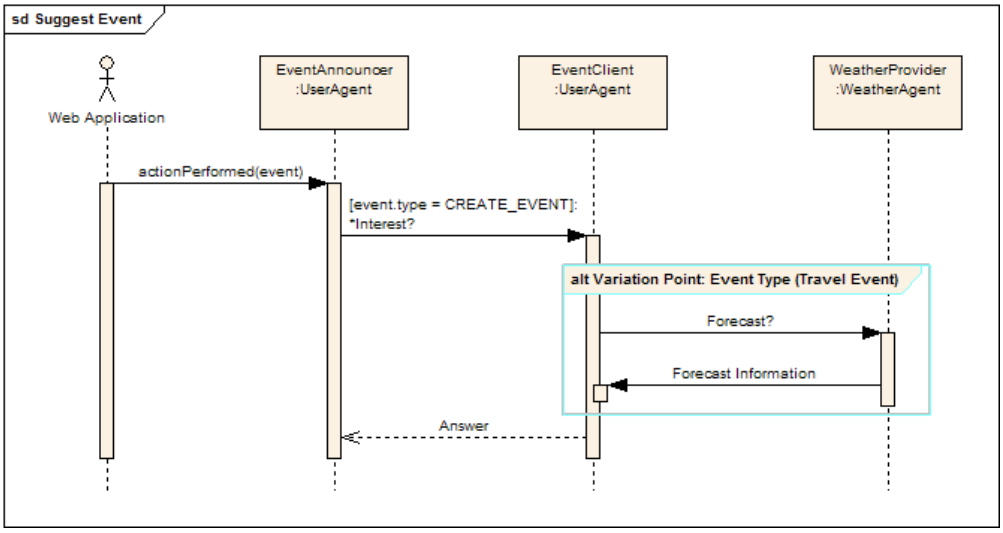
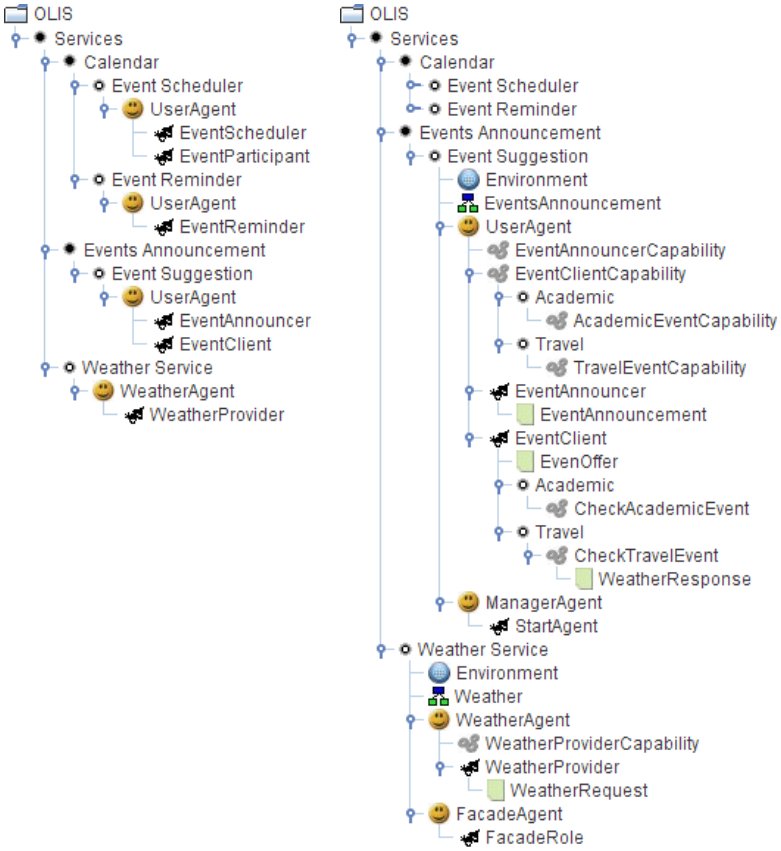


Figure 4.18: Role Identification Diagram - Event Suggestion.

After identifying agents and roles, they were related to features through the OLIS Feature/Agent Dependency model, which is presented in Figure 4.19(a). There, it can be seen the four agent features (*Event Scheduler*, *Event Reminder*, *Event Suggestion* and *Weather*) and their associated agents and roles. Differently of EC, the roles are not shared among features. The first three roles depend on the `UserAgent`, and the last one on the `WeatherAgent`.

Even though there is a communication between the `UserAgent` and the `WeatherAgent` in the *Event Suggestion* feature when the event type is *Travel*, there is no direct dependency between the *Event Suggestion* and the `WeatherAgent`. However, there is an indirect dependency because there is a constraint in the feature model that express this dependency; therefore if both *Event Suggestion* and *Travel* features are selected, the

Weather feature must be selected, and consequently the `WeatherAgent` will be present in the derived product.



4.19(a): Analysis Level 4.19(b): Design Level

Figure 4.19: OLIS Feature/Agent Dependency Models (Partial).

Task Specification. The tasks of all agents were specified in task specification diagrams. This Event Suggestion task specification diagram for the `UserAgent` is presented in Figure 4.20. As happened in the previous diagram (Figure 4.18), there is a variation point in the diagram due to the feature interaction between the *Event Suggestion* and the *Event Type* features. In this variation point, there are two alternative paths: one for Travel event and another for Academic.

Domain Design

The OLIS MAS-PL architecture has evolved over time. Its first version could only derive typical web applications and the unique variability was the event type. As new features were incorporated into the MAS-PL, its architecture was adapted.

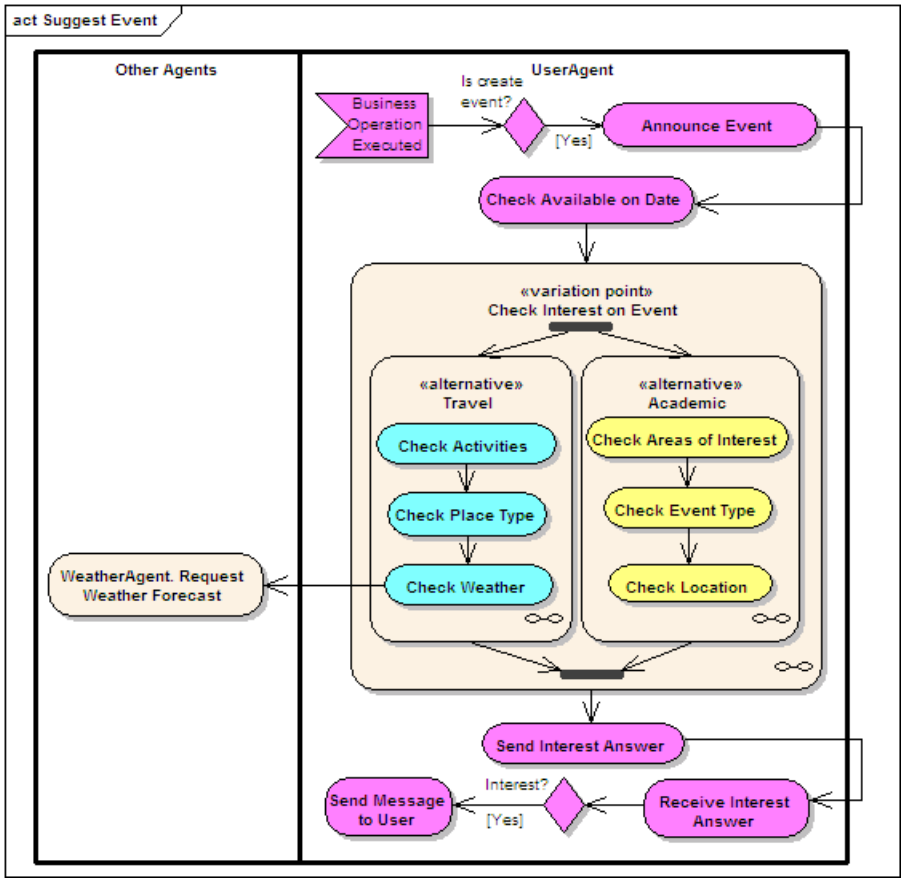


Figure 4.20: Task Specification Diagram - Event Suggestion.

Architecture Description and Implementation Platforms Selection. The OLIS architecture is very similar to the EC, being also structured according to the Layer architectural pattern. The layers that compose the web application component are exactly the same of the EC - `GUI`, `Business` and `Data` layers. The responsibilities attributed to each one of the layers are also the same. In addition to the two components – Business Layer Monitor and Agents Layer – that are part of the EC architecture, whose purpose is to incorporate agents and the web applications, one more component was added: the Agents Facade. This component is the access point for the web application retrieve information provided by agents. For instance, when a user requests the weather, a business service requests the facade agent to retrieve the weather, and it is responsible for sending a message for the weather agent to get it, and then pass the information back to the service that requested the weather.

Several frameworks were used to implement the OLIS MAS-PL, most of them are the same of the EC. One difference is that we used a newer version of the WAF in the GUI Layer: in EC we used the first version of the Struts framework, and Struts 2 in OLIS. An advantage of Struts 2 is that it allows the modularization of actions into packages, which are defined in an XML file.

This provides an easy configuration of the GUI layer.

The implementation of the agent layer was accomplished using a hybrid agent architecture: there are agents implemented with JADE and others with Jadex (Pokahr 2005). The main differences between JADE and Jadex is that the last follows the BDI model, specifies agents in an XML file in terms of their beliefs, goals and plans, and it provides a reasoning engine; and the former is task-oriented and agents are implemented by extending the `Agent` and `Behavior` classes of JADE framework.

Agent Modeling and Agent Society Modeling. In the OLIS architecture, there are two more agents then the ones identified at the Domain Analysis phase: the `ManagerAgent` and the `FacadeAgent`. Next we describe each one of the OLIS agents:

`ManagerAgent:` this agent is responsible for creating new `UserAgent`s when a new user is inserted in the database. It also starts a `UserAgent` for each user already stored in the database during the application start up. It is equivalent to the `UserDataAgent` from EC;

`FacadeAgent:` this agent is the access point of the web application to get information from the agents. It hides the other system agents from application, so that the application only needs to know about the `FacadeAgent` to get the information from the agents. This is part of the Web-MAS architectural pattern;

`WeatherAgent:` this agent provides the weather information. It looks for the current weather conditions and the forecast of a specific location;

`UserAgent:` each user of the system has a `UserAgent` that represents him/her. Each `UserAgent` can have at most five different roles according to the features selected for a product: (i) `EventReminder` role – reminds the user about events that are going to begin; (ii) `EventScheduler` role – invites other users to a calendar event and finds a time for the event that is compatible with the participants' schedule; (iii) `EventParticipant` role – accepts or rejects an invitation to participate of an event and, in case of reject, provides a time that is appropriate for the user according to his/her schedule; (iv) `EventAnnouncer` role – announces new events to the other user agents; (v) `EventClient` role – checks if the event announced is interesting according to the user preferences. This role also checks if the weather will be good according to the place type, consulting

the `Weather` agent. Eventually, the `UserAgent`s can access the business services to perform changes in the data model.

The structure of OLIS agents, roles and organization was modeled using MAS-ML diagrams. Figures 4.21, 4.22 and 4.23 present a partial view of the class, role and organization diagrams for the *Event Suggestion* feature.

In the class diagram (Figure 4.21), the are the two agents that are related to this feature – the `UserAgent` and the `WeatherAgent`. The `WeatherAgent` is colored in green because it is related to the *Weather* feature; however there is a constraint in the feature model expressing that if the *Event Suggestion* and *Travel* features are selected, it is mandatory the selection of the *Weather* feature; therefore, even though the `WeatherAgent` is not directly related to the *Event Suggestion* feature, it will be present in a product because of the constraint. In addition, capabilities are aggregated to the `UserAgent`, which is shared among other features, to provide the specific behavior for the *Event Suggestion* feature. Moreover, one of these capabilities has alternative capabilities aggregated to it in order to modularize the variable behavior of the `EventClientCapability`. Similar observations can be made about the role diagram (Figure 4.22).

Finally, the organization diagram (Figure 4.23) shows which agent plays which role in the organization. Only the agents and roles are present there, and not the capabilities.

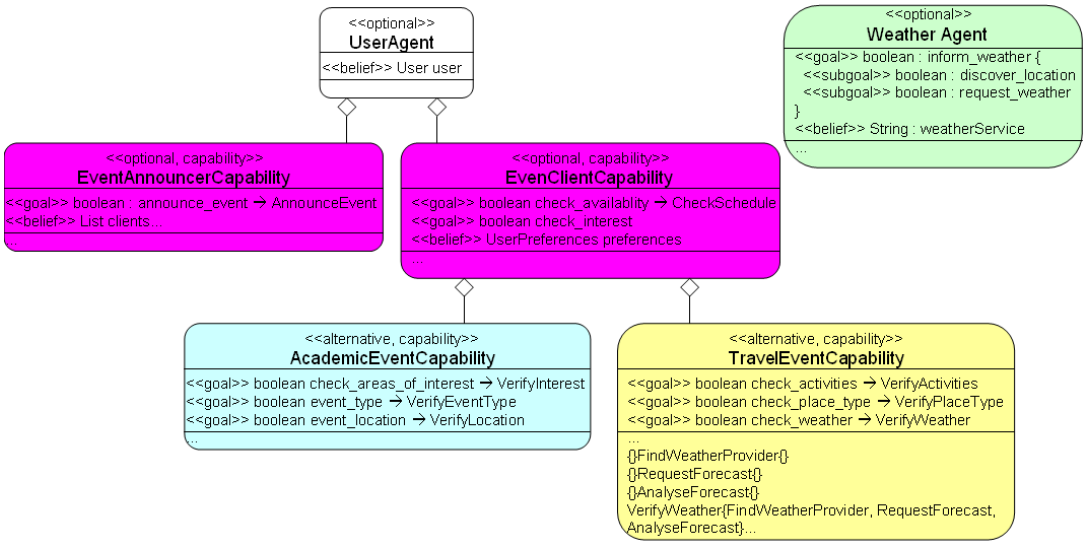


Figure 4.21: OLIS Class Diagram.

The dynamic behavior of OLIS agents, such as agent plans, was modeled with MAS-ML extended sequence diagrams. The plans necessary for the *Suggest Event* feature – `Announce Event`, `Check Availability`, and so on

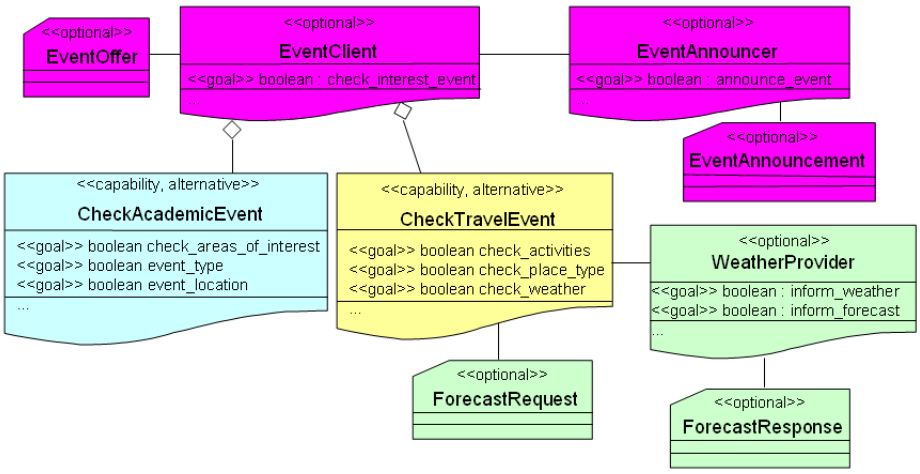


Figure 4.22: OLIS Role Diagram.

– were each one modeled in a separate sequence diagram. Even though this feature is crosscut by the *Event Type* feature, the agents' variable parts were modularized into capabilities. For instance, when a `UserAgent` has a goal of verifying interest in some event, this goal is achieved in a certain way according to the selected capability, which has specific subgoals, plans and beliefs. Therefore, there is no plan of the *Suggest Event* feature that has an alternative or optional behavior that required the use of UML 2.0 frames. As a consequence, this variability was supported by different plans that to be appropriately selected according to which event type was selected (*Academic* or *Travel*). As in the EC, non-agent features were modeled used typical sequence diagrams.

Several agent elements were introduced in the OLIS architecture during the Domain Design phase. These elements were added to the Feature/Agent Dependency model, which is presented in Figure 4.19(b). The `Environment` is present in all agent features, meaning if at least one of them is selected, the `Environment` will be present in the product.

Domain Realization

The two activities of the Domain Realization phase for the OLIS are described in this Section. First, we detail the techniques used to implement OLIS agents, and later we describe how assets were implemented and mapped onto design elements.

Agent Implementation Strategy Description. Jadex was the framework we chose to implement our agents, because the BDI model has been successfully used in large scale systems and is relatively mature for modeling cognitive

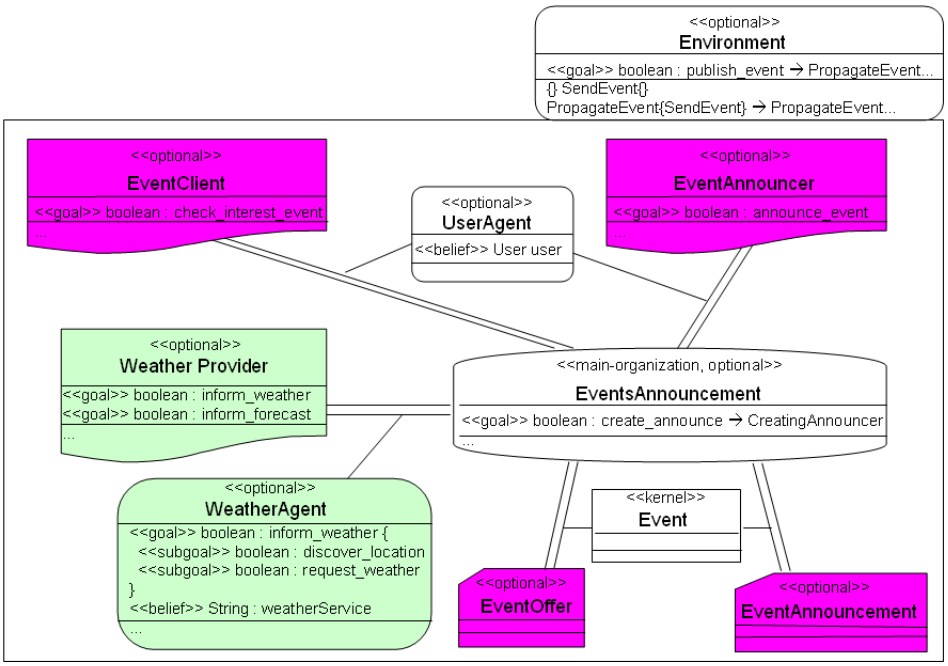


Figure 4.23: OLIS Organization Diagram.

agents. In addition it provides several concepts specified in MAS-ML models, facilitating the transition from the design models to implementation. However, there are two agents of the MAS-PL that needed to be accessed by the web application component – the `EnvironmentAgent` and the `FacadeAgent`. As JADE agents are Java classes, these agents were implemented with this framework and they have specific methods to the web application invoke.

Some techniques were used to implement OLIS architecture: (i) as in EC, the environment was implemented as an agent, the `EnvironmentAgent`; and (ii) agent roles were implemented as capabilities, and as in EC the roles also provide a certain behavior for the agent.

Assets Implementation. Mostly, variability was implemented using *polymorphism* and *design patterns*. OLIS MAS-PL can be configured through three different types of configuration files: (i) Struts XML file – this file configures the `GUI` layer, such as the actions/commands that it will have; (ii) Spring configuration files – these files configures the `Business` and `Data` layers, configuring the business services and entities that will compose the domain model; (iii) Jadex ADF files – ADFs files are XML files that define agents and capabilities.

The last task of the Assets Implementation activity is to map the implementation elements to the design elements. Figure 4.24 illustrates a partial view of this mapping for the OLIS. As the agents of this MAS-PL were implemented using JADE and Jadex frameworks, some agents have only classes associated to them and others have XML files too. The envi-

ronment was implemented with JADE, so three classes (`EnvironmentAgent`, `PropagateBusinessOperationBehavior` and `RegisterAgentBehavior`) are associated with it. The other agents in Figure 4.24 were implemented with Jadex, so they have an XML file that implements agents and some Java classes that implement plans. Similarly, capabilities have an XML file and classes associated with them. Jadex does not provide a way to model roles, so the capability XML implements this concept.

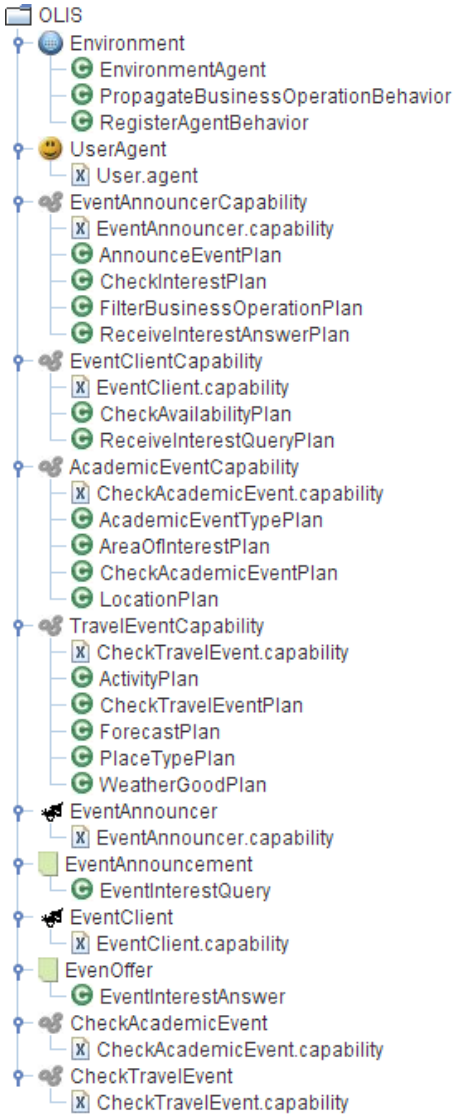


Figure 4.24: OLIS Design/Implementation Elements Mapping.

4.3 Final Remarks

This Chapter presented two MAS-PL case studies that illustrate our process. The first one is the ExpertCommittee, which is a product line of conference management systems that has agent features that automate user

tasks. This MAS-PL was developed using an extractive approach, so the product line architecture was built based on several system versions. The second one is the OLIS MAS-PL, which allows the derivation of web-based systems that provide personal services for the user, such as calendar. It was constructed using a reactive approach, so a first product line version was developed providing the core features and an alternative one, and, in sequel, it was incremented with new agent features.

Both case studies are MAS-PL for the web domain; however there is no evidence that the process is applicable only for this specific domain.

5
Lessons Learned

In this chapter, we present and discuss some lessons learned from our experience of development and evolution of the two case studies presented in the previous chapter, the EC and OLIS MAS-PLs. Our lessons learned are divided into two parts: the first one (Section 5.1) describes design and implementation guidelines for developing MAS-PLs, and the second details limitations of our approach (5.2).

5.1
Design and Implementation Guidelines

The implementation of modularized features is a very important issue in the SPL development in order to provide an architecture that supports the variability. In addition, some techniques can be used to improve this modularization allowing a better evolution of the SPL and a (semi-)automatic product derivation. In this section we present guidelines to design and implement agent features. These guidelines resulted from our experience while developing MAS-PLs. They are mainly related to: (i) how to structure an architecture that allows an integration of typical web applications architectures and agents in a decoupled way (Section 5.1.1); (ii) how to implement agent roles in some agent platforms that do not provide this concept (Section 5.1.2); and (iii) how to improve modularization using Aspect-oriented Programming (Section 5.1.3).

5.1.1
Web-MAS Architectural Pattern

In this section, we present the Web-MAS architectural pattern (Nunes 2008b). This pattern was derived from our MAS-PL case studies (Chapter 4) based on the common elements identified when integrating the web based systems and their respective software agents. The proposed pattern provides a general structure to add autonomous behavior to existing web applications using agent technology. This extension has a minimum impact on the architecture of web-based systems. Moreover, the agents can be easily

removed after being introduced on the system, allowing an (un)plugability of agent features.

When extending the web systems to incorporate the autonomous behavior, we have identified mainly two problems. The first problem is how software agents can perceive changes in the environment. We consider the environment the data model with its current data information. Changes on this model happen as a consequence of the user interactions with the system. So each time the user performs an action that changes the data model, the agents should detect it or be notified about this fact, and then they take the appropriate actions. In BDI agents, we can think the data model, or a part of it, as the beliefs of the agents, and the agents should perceive when their beliefs change. A possible solution to this problem is that agents can query the database in periodic times, as it is proposed in (Choy 2005). However, this solution can cause a big overhead in the system if it is done in short intervals, or changes will be perceived with a large delay if it is done in long intervals, which can cause undesired situations, such as missing a change if more than one change happen in the same data.

The second identified problem is that system functionalities may retrieve information from the agents. However, software agents exchange messages, and objects call methods; so it is a problem for an object from the system to retrieve information from the agents. Usually, the agent platforms do not provide an easy way for objects from the system to interact with agents. For example, in Jadex, the agents are specified in XML files, and an object is not able to access them and does not know their interface to call a specific method. Nevertheless, this is something desired, because agents can provide information that needs reasoning and learning to be produced. Furthermore, it is common that it takes some time to get information from the agents, as this can require a lot of processing and messages exchanges. Thus, delayed answers should also be considered.

The Web-MAS architectural pattern was proposed to solve both problems. The pattern addresses applications that follow the typical web application architecture, i.e. the Layer architectural pattern (Buschmann 1996). However, it would also be adapted to consider other alternative implementations of web-based systems. This pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction. The proposed solution is composed of the following components: (i) the presentation, business and data layers, which comprise the web application; (ii) the agents layer; (iii) the business layer monitor; (iv) and the agents layer facade. The structure of these components is depicted in the

Figure 5.1. Next we describe each one of these components:

Presentation Layer. This layer can also be called Graphical User Interface (GUI) layer. The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand. Usually, this layer follows the MVC pattern (Fowler 2002). This pattern considers three roles: (i) *model* – an object that represents some information about the domain; (ii) *view* – represents the display of the model in the user interface; and (iii) *controller* – takes user input, manipulates the model and causes the view to update appropriately. Commonly, WAFs are used to implement this layer;

Business Layer. This layer is also known as Logic layer. It coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers. Typically, there is transaction control in this layer;

Data Layer. In this layer, the information is stored and retrieved from a database or a file system. It is then passed back to the Business layer for processing, and then eventually back to the user. The information is represented in a data model, in which there are objects and relationships among them;

Agents Layer. This component is responsible for the autonomous behavior *de facto*. It is composed of software agents. The agents provide intelligent services and automate tasks that were previously done directly by users. Instead of being simple objects with attributes and methods, they have beliefs, goals and plans. The agents receive messages from the `Environment` agent about the execution processes that this agent detects by monitoring the services of the Business layer. According to the messages received, the agents take appropriate actions and can also perform changes in the data model by using the business services;

Business Layer Monitor. This component is responsible for monitoring the business operations of the web application. The business operations to be monitored are the ones that are related to the autonomous behavior. The Business Layer Monitor aggregates the `Environment` agent, which receives notifications about the operations executed in the Business layer and propagates them to the other agents;

Agents Layer Facade. This component is the access point of the web application to the Agents Layer. Besides the information that is stored in the data model, agents can also generate information through some processing and exchanging messages with other agents. Then, this facade provides an interface to the business services get information from the other agents of the system. This component is composed by the `Facade` agent, which receives a request from a business service, forward it to the appropriate agent and pass the result back to the service. When this agent starts up, it registers itself as a singleton instance. Then the business services can access this agent and make requests. There are three ways of communication: (i) *Synchronous* – the business service calls the `Facade` agent and waits for the response; (ii) *Asynchronous with pooling* – the business service calls the `Facade` agent, continues its processing, and periodically checks if the response arrived; and (iii) *Asynchronous with callback approach* – the business service calls the `Facade` agent, continue its processing, but it is notified through a callback function, which is passed as parameter when the `Facade` agent was called.

The communication between the business services and the `Environment` agent is accomplished by means of the introduction of the Observer design pattern (Gamma 1995). The intent of this pattern is to define a one-to-many dependency between objects so that when one object performs an action or changes state, all its dependents are notified automatically. By the use of this pattern, we keep a loose coupling between the application and the Agents layer. In the Observer pattern, the concrete subject is the object that sends a notification to its observers when its state changes or performs an action; thus all the services that compose the Business layer are concrete subjects. They must implement the `Observable` interface, which allows the observation of their actions. For each call of the business methods, the services not only execute the requested method, but they also notify their respective observers. The concrete observer implements an updating interface to receive notifications from the subject. In our architecture, there is only one concrete observer, which is the `Environment` agent. This agent registers itself as an observer of the services that compose the Business layer when it is initialized. When some action is performed in the Business layer, the `Environment` agent is notified about this event and it broadcasts the event to all other agents of the system.

An important implementation detail related to the Business layer is the transaction management. In typical web applications, the execution of each business method is under a transaction. Thus, if an error occurs during the method execution, the operations that were already executed are undone.

With transaction management, the information stored in the database cannot achieve an inconsistent state. The implementation issue is whether if the notification to the observers should be done inside or outside the transaction scope. If the business methods should be committed even though an error occurs during the notification to the agents, this notification should be outside the transaction or the exception thrown should be caught and treated. If the business method execution must rollback when something wrong happens during the notification to the agents, the notification must be inside the transaction.

Another design and implementation issue is related to the system performance. The inclusion of notifications on the business methods implies an overhead to the system, in particular when an entity is deleted, because its state is read and kept in memory before its deletion. Though, only the methods that impact in the behaviors of the agents should propagate their execution. Usually, methods that only retrieve information from the data model do not need to notify the observers.

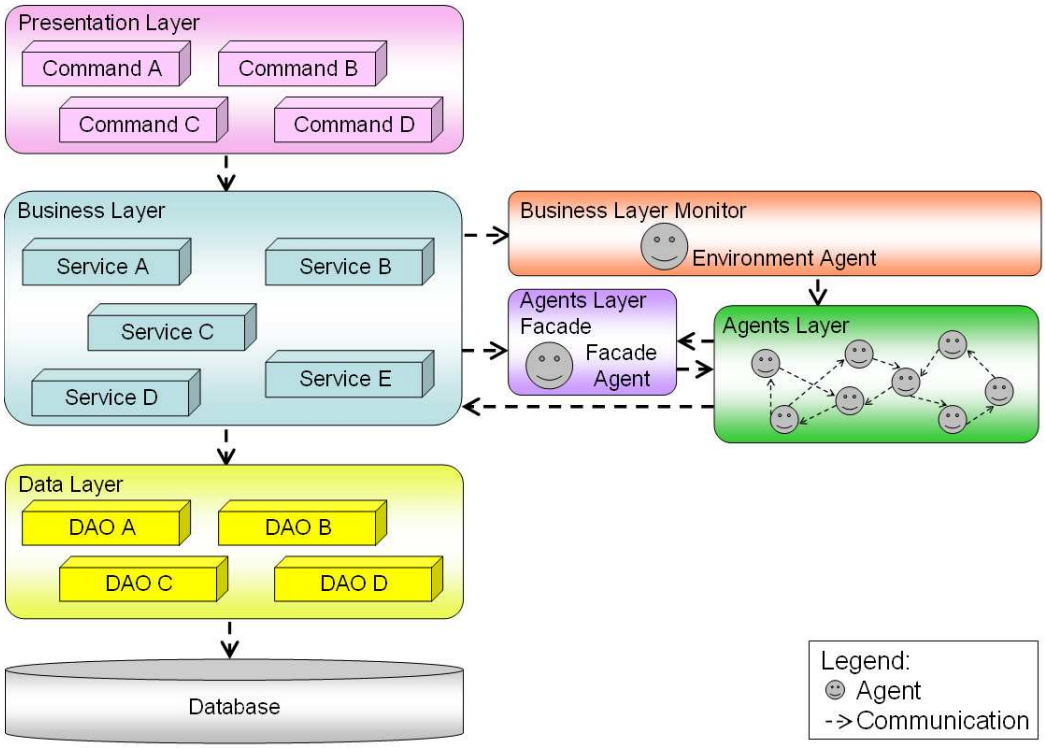


Figure 5.1: Web-MAS Architectural Pattern.

5.1.2 Roles Implementation

The concept of roles is adopted in most of MASs, meaning that an agent can play roles in organizations. Several MAS methodologies define

ways of modeling and specifying roles, such as Gaia (Wooldridge 2000a, Zambonelli 2003) and PASSI (Cossentino 2005), which were described in Chapter 2. However, not all frameworks for implementating agents provide this concept. In this Section, we describe how to implement the role concept with JADE (Bellifemine 2007) and Jadex (Pokahr 2005) frameworks in a modularized way, based on our experience while developing MAS-PLs. Both solutions model roles not only as an "interface" of the role that an agent can play, but also as an implementation of the necessary behavior to the agent play that role.

Implementing Roles with JADE

JADE is a framework implemented in Java language, which simplifies the implementation of MASs through a middleware. Basically, the developer should extend the `Agent` class to implement agents and extend the `Behavior` class to define behaviors that the agent should have. These are the main first class elements provided by the framework.

As agents are implemented with JADE framework only by extending Java classes, typical object-oriented techniques can be used when using this framework, such as design patterns and polymorphism. In this context, the Role Object pattern (Baumer 1997) is a design pattern whose purpose is "*to adapt an object to different client's needs through transparently attached role objects, each one representing a role the object has to play in that client's context. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified*". Besides allowing the implementation of roles using JADE, this pattern keeps the role modularization, which is an important characteristic of the SPL development.

According to the pattern, we have the following participants: (i) Component – models a particular key abstraction by defining its interface and specifies the protocol for adding, removing, testing and querying for role objects. This interface represents the agent interface; (ii) ComponentCore – implements the Component interface including the role management protocol. This is the agent itself, which extends the `Agent` class of JADE; (iii) ComponentRole – it is an abstract role of the agents; (iv) ConcreteRole – concrete roles are the agents' roles, which are dynamically attached to agents.

An example of the use of the Role Object pattern can be seen in the ExpertCommittee case study, described in Chapter 4.

Implementing Roles with Jadex

In the same way of JADE, Jadex framework is also implemented in Java. However, instead of extending classes, the developer specifies agents in XML files, which contain mainly beliefs, goals and declaration of plans. Plans are implemented in classes that must extend the `Plan` class. In addition, Jadex provides the capability concept, which is declared in a similar XML file and can be incorporated to agents and other capabilities. It provides a reuse mechanism. Moreover, Jadex follows the BDI model (Rao 1995) and has a reasoning engine that chooses plans in order to achieve goals. Nevertheless, Jadex does not provide the role concept.

The solution proposed for implementing roles with Jadex is based on capabilities. The beliefs and goals of a specific role are modularized into a capability, and optionally the plans necessary to achieve the goals. If the plans are not declared into the capability, the role will just define the "interface" that the agent will have while playing a certain role; and if the plans are declared, the role will provide both "interface" and behavior for the agent, like the solution adopted with JADE. The capabilities are statically declared into an agent; however it is possible to change the agent model on runtime, what allows to add and remove capabilities dynamically.

An example of implementing roles with Jadex can be seen in the OLIS case study, described in Chapter 4.

5.1.3 Improving Modularization using Aspect-Oriented Programming

Recent research work presents the benefits of adopting AOP techniques to improve the modularization of features in SPLs (Alves 2006, Figueiredo 2008), framework based (Kulesza 2006b) or MASs (Garcia 2004) architectures. The increasing complexity of agent-based applications motivates the use of AOP. AOP has been proposed to allow a better modularization of crosscutting concerns, and as a consequence to improve the reusability and maintenance of systems (Kiczales 1997). Among the problems of crosscutting variable features, we can enumerate: (i) *tangled code* – the code of variable features is tangled with the base code (core architecture) of a SPL; (ii) *spread code* – the code of variable features is spread over several classes; and (iii) *replicated code* – the code of variable features is replicated over many places. All these problems can cause difficulties regarding the management, maintenance and reuse of variable features in SPL.

In order to promote improved separation of concerns, some crosscutting features that present the problems mentioned above are natural candidates

to be designed and implemented using AOP. During the development of our MAS-PL case studies, we have found the following interesting situations to adopt AOP techniques:

– *Modularization of the glue-code that integrates the web-based system (base code) with the agent features.* In order to integrate agents with the web application, the Web-MAS architectural pattern (Section **??**) uses the Observer design pattern to observe/intercept the execution of business methods of the services of the Business layer. This pattern provides an abstract coupling between the subject and the observer; the services of the Business layer does not know who the concrete observers are. Furthermore, there are some Application Program Interfaces (APIs), such as the Java API, that already provide the `Observable` class and the `Observer` interface.

  However, AOP (Kiczales 1997) can be used to modularize the intercepted code that allows the agents monitor the execution of the web-based system. It facilitates the (un)plugging of the agent features in the system and makes the addition of agents even less intrusive. AOP enables to separate code that implements crosscutting concerns and modularize it into aspects. It provides mechanisms and techniques to compose crosscutting behaviors into the desired operations and classes during compile-time and even during execution. The source code for operations and classes can be free of crosscutting concerns and therefore easier to understand and maintain. The code of the Observer pattern presents an invasive nature, resulting on a scattered and tangled code among several classes. Therefore, this leads the pattern to "disappear into the code", and it also bring difficulties to the understanding, maintenance and documentation of the pattern, and consequently of the application; and

– *Modularization of the agent roles.* The use of the Role Object pattern (Baumer 1997) to implement agent roles with JADE framework provided the role modularization by the use of object-oriented techniques. However, the use of this pattern cannot provide an improved isolation of the agent role features, which is essential to SPL variability management. The implementation of the agent classes (e.g. `UserAgentCore` class) requires, for example, the activation and deactivation of the agent roles over different points of the execution of the agent behavior, such as agent initialization, execution of specific plans, etc. The adoption of AOP to modularize agent roles (Garcia 2005, Kendall 1999) is thus a

better option to improve the modularization and evolution of the agent roles features.

Empirical studies were performed with MAS-PLs in order to analyze the (dis)advantages of the use of AOP to implement agent features. The MAS-PLs were implemented using different agent platforms (JADE and Jadex) and different implementation techniques (conditional compilation, design patterns, configuration files and AOP). Results of these studies can be seen in (Nunes 2008c) and (Nunes 2009a).

5.2 Limitations of Our Approach

Our approach was developed in a bottom-up fashion. The development of the EC case study allowed us to identify the deficiencies in the MAS-PL development. Next, we have made comparison studies among SPL approaches and MAS approaches, and have modeled agent features of the EC MAS-PL with them. The result was the selection of PLUS, PASSI and MAS-ML to incorporate our process. In addition, new activities and adaptations to these approaches were proposed to address their drawbacks in the development MAS-PLs. Later, the EC was developed using our process, and this helped to refine our process. Finally, the OLIS case study was developed to evaluate our process. In addition, some activities of our approach, mainly related to the modeling of agent features, were adopted in the development of design and implementation projects in the Software Systems Design graduate course at PUC-Rio.

The proposed process aims at addressing the problems stated in Chapter 1, most of them identified with our exploratory studies. However, in this section we point out some limitations of our approach to developing MAS-PLs. First, our approach is the definition of a domain engineering process; nevertheless application engineering is the other process of SPLE, which defines a prescribed way in which all models and reusable assets will be used to derive applications. Although we provide means for deriving products by keeping features traceability, we do not explicitly define means of performing the application engineering. Even though this process of the SPLE is out of the scope of this dissertation, it is important to consider that it is also needed and should be addressed in future work.

In our process, we have adopted stereotypes to model variability in the models, and have proposed new models to trace variability. The stereotype concept is typically used for modeling variability, because it is the standard extension mechanism offered by the UML, and thus allows variability infor-

mation to be made explicitly visible in diagrams (Muthig 2002). Although the use of stereotypes has these advantages, one drawback of their use is keeping consistency among models. The variability information is spread along several models, and no technique is used to verify if this information is consistent. One scenario that this is important is when a mandatory feature becomes optional, and this change must be propagated to all models.

A potential solution for this problem is to incorporate techniques from Model-driven Development (MDD). According to (Beydeda 2005), in MDD, models are used to reason about a problem domain and design a solution in the solution domain. Relationships between these models provide a web of dependencies that record the process by which a solution is created, and help to understand the implications of changes at any point in that process. In addition to creating these models, we can define rules for automating many of the steps needed to convert one model representation to another, for tracing between model elements, and for analyzing important characteristics of the models. Exploiting MDD approaches to improve our process is part of future work.

Our case studies explored the scenario of incorporating features with autonomous and pro-active behavior in web applications with typical web architectures, which use widespread patterns. We have chosen this scenario, because we believe this is a potential application of agents, which can help with the automation of users' tasks. As a consequence, in all case studies, agents are not part of the MAS-PL core. However, MAS-PLs can have agents as part of their core; therefore it is important to validate if our approach covers this situation. Nevertheless, there is no restriction in our approach to have agents as part of the core – it is necessary only to use the appropriate stereotypes – but there is the need to explore this scenario to prove it.

5.3 Final Remarks

This chapter presented lessons learned with our experience with the MAS-PL development. We first described guidelines to help in agent features modularization in the development of MAS-PLs. We presented the Web-MAS architectural pattern (Nunes 2008b), whose purpose is to integrate traditional web applications architectures with software agents in such way that they can be easily (un)plugged. Therefore, the pattern provides the extension of existing web applications to incorporate new agent features with a low impact and provides an architecture that supports optional agent features. We also showed how the role concept can be implemented in two agent platforms: JADE

and Jadex. Both of them do not provide this concept. Finally, we presented and discussed the adoption of Aspect-oriented Programming in some situations in order to provide a better modularization. The second lesson learned is related with the limitations of our approach, mainly showing parts of the MAS-PL development, which were not covered by our approach.

6
Related Work

The main contribution of this dissertation is the proposal of a domain engineering process for developing MAS-PLs. Therefore, approaches that address the modeling and documentation of MAS-PLs are presented in this Chapter (Section 6.1). We have analyzed these approaches and pointed out their identified weaknesses.

In addition, we also describe architectures and frameworks that aim at incorporating agents into web applications (Section 6.2). These are alternative approaches to our Web-MAS architectural pattern. So, we discuss the problems we found on these approaches, and detail their differences compared to our proposal.

6.1
Multi-agent Systems Product Lines Approaches

In the literature, there are only few attempts that integrate SPL and MAS. Both existing approaches that address this integration – MaCMAS Extension (Pena 2006a) and GAIA-PL (Dehlinger 2007) – are based on existing MAS methodologies. The main problems that we have observed on these MAS-PL methodologies to model and document are: (i) they do not offer a complete solution to address the modeling of agent features in the whole engineering process; (ii) they suggest the introduction of complex and heavyweight notations that are difficult to understand when adopted in combination with existing notations (e.g. UML); and (iii) they do not capture explicitly the separated modeling of agent features.

6.1.1
MaCMAS Extension

Methodology Fragment for Analysing Complex MultiAgent Systems (MaCMAS) (Pena 2005) is an AOSE methodology that is designed to deal with complex unpredictable systems, which uses UML as a modeling language. In (Pena 2006a), it is proposed an approach to developing the core architecture of a MAS-PL. It consists of using goal-oriented requirement documents, role

models, and traceability diagrams in order to build a first model of the system, and later use information on variability and commonalities throughout the products to propose a transformation of the former models that represent the core architecture of the family.

The software process proposed is composed by four activities: (i) build acquaintance organization – consists of developing a set of models in different layers of abstraction where we obtain a MaCMAS traceability model and a set of role models showing how each goal is materialized. This is achieved by applying the MaCMAS software process; (ii) build features model – responsible for adding commonalities and variabilities to the traceability model; (iii) analyze commonalities – it is performed a commonality analysis to find out which features, called core features, are more used across products; and (iv) compose core features – the role models corresponding to these features are composed to obtain the core architecture.

In (Pena 2006c, Pena 2007), Pena et al. use this approach to describing, understanding, and analyzing evolving systems. The approach is based on viewing different instances of a system as it evolves as different "products" in a SPL. That SPL is in turn developed with an AOSE approach and views the system as a MAS-PL.

One deficiency that we have identified in this approach is that the variable aspects of the systems are analyzed after modeling the MAS, and this can lead to undesirable situations, such as, high coupling between mandatory and optional features and inadequate modularization of agent features. Inadequate feature modularization results in non-effective application engineering and this increases time and costs for deriving new products, which contradicts the main reason for SPLs. In addition, the approach does not detail how the MAS-PL assets can be implemented so that they can be assembled to derive a product.

6.1.2 GAIA-PL

Dehlinger & Lutz (Dehlinger 2007) have proposed an extensible agent-oriented requirements specification template for mission-critical, agent-based distributed software systems that supports safe reuse. Their proposal adapted portions of the Gaia (Wooldridge 2000a, Zambonelli 2003) agent-oriented methodology and integrated them with a product-line-like approach, which exploits component reuse during system evolution. It is rooted within the requirements engineering phase. The proposed template allows ready integration with an existing tool-supported, safety analysis technique sensitive to dynamic variations within the components (i.e., agents) of a system.

The requirements specification template allows to capture dynamically changing configurations of agents and reuse them in future similar systems. The requirements are documented in two schemata: (i) Role Schema – defines a role and the variation points that the role can assume during its lifetime; (ii) Role Variation Point Schema – captures the requirements of a role variation point's capabilities. A Role Schema has an associated set of Role Variation Point Schemata. These schemata specify all the possible variation points of the roles, thus a new member to be added to the distributed system can be instantiated by specifying each new member to be deployed in the Agent Deployment Schema, which describes how different members of the distributed system that are to be deployed shall be instantiated.

The scope of this approach just cover a small part of the domain engineering process, focusing only on the requirements engineering. In addition, even though different capabilities can be specified for an agent, it is not possible to combine them. Thus, if a role schema has two variation points and one wants to choose both of them to compose an agent, a third variation point must be created defining both capabilities combined. Furthermore, the approach does not explicitly account for feature/role interactions.

6.2 Web-based Systems and Agent Integration Approaches

Several agent frameworks and platforms (Bellifemine 2007, Bordini 2007, Pokahr 2005) have been developed to ease the implementation of MASs. However, most of them do not address applications whose interface is the web. In this Section, we present some approaches that emerged in order to integrate agents and web applications. We also detail weaknesses of these approaches, which we aimed to solve with the Web-MAS architectural pattern (Section 5.1.1).

6.2.1 TaMEX Framework

Stroulia & Hatch (Stroulia 2003) propose a software framework called TaMeX, which supports the development of intelligent multi-agent applications that integrate existing web-based applications offering related services in a common domain. The TaMeX applications rely on a set of specifications of the domain model, the integration workflow, their semantic constraints, the end-user profiles, and the services of the existing web applications; all these models are declaratively represented in the XML-based TaMeX integration-specification language. At run-time, the TaMeX agents use these models

to interact with the end users, monitor and control the execution of the underlying applications' services and coordinate the information exchange among them, and to collaborate with each other to react to failures and effectively accomplish the desired user request.

The TaMeX framework provides: (i) a language with well-defined semantics for specifying the application domain, the existing services and the service composition; (ii) a suite of intelligent methods for adapting existing web-based applications so that they "speak" the same – in terms of syntax and semantics-language; and (iii) a robust run-time environment for executing the integrated applications and deliver the desired overall service. The TaMeX architecture is a distributed multi-agent architecture consisting of two types of agents: task agents, which are responsible for interacting with the end users; and application wrappers, which are responsible for executing existing web applications and translating between the domain model of the integrated application and the individual domain models of the wrapped applications.

The Web-MAS architectural pattern also proposes the addition of software agents on existing web applications. However, we aimed at proposing a simple solution for that, which is founded on existing object-oriented design techniques and can be integrated with current adopted web technologies. On the other hand, the use of the TaMeX framework implies learning of a new programming language and specific methods. Besides, it obligates the use of its language to implement the software agents.

6.2.2 Choy et al. Approach

Choy et al. propose in (Choy 2005) the use of software agents to make the communication in a distance learning community more effective. They focus on the communication between teachers and students. Their software agents are designed to work on behalf of teachers, assisting them in communicating more effectively and closely with students, saving lot of time by delegating routine jobs. Basically, the software agents monitor the system and send alert e-mails in specific situations. The main elements that were introduced in the web application are: (i) `Schedule Control` class, which controls the agent's life and behavior; (ii) `Job Listener` classes that generate notifications when exceptional events occur; (iii) `Data Retrieval` class, which retrieves all required information at once for the `Job Central Processing` class; and (iv) `Job Central Processing` class, which hosts the criteria to generate e-mail alerts and sends them.

This work does not actually integrate agents into the web application.

The agents run in a parallel way with the system and consult the same database that is manipulated by the web application. Agents do not communicate with the rest of the system. Moreover, it does not allow the web application to access information directly from the agents. Both situations are addressed by the proposed Web-MAS architectural pattern.

6.2.3 Jadex Webbridge framework

A new architecture is proposed in (Pokahr 2007a) and (Pokahr 2007b), whose purpose is to seamlessly integrate agent technology and web applications. This architecture is in accordance with the Model 2 (Ford 2004) design pattern and the primary objective of the approach is to separate the agent-specific parts of an application from the web-specific parts such as HTML pages. An extra layer was introduced, performing the necessary mediation operations, to achieve the desired independence between the web front-end and the agent application.

The agentified Model 2 architecture refines only a small part of the original architecture by further developing the controller component, which serves as a connector that translates interactions with the view into actions to be performed on the model. This enables using agents for all aspects related to application functionality while preserving the usage of the existing and well suited Model 2 techniques for rendering (JSPs) and model representation (JavaBeans). One crucial aspect of this extended architecture is the partitioning of the controller into three distinct functionalities: delegate servlet, coordinator agent and application agents. The delegate has the main purpose to forward business tasks that originate from browser requests to the coordinator agent. The coordinator processes requests by distributing them to domain-dependent application agents.

This architecture is the basis for the Jadex Webbridge framework, which provides ready-to-use and extensible functionalities realizing the delegate servlet and the coordinator agent. Additionally, a web interaction module (capability) is provided that encapsulates the generic functionalities needed by application agents.

The Web-MAS architectural pattern also aims at seamlessly integrating agent technology and web applications; however this approach proposes that all the application logic is performed by software agents. Therefore, simple functionalities (e.g. create new system entities) that can be easily implemented with usual patterns and frameworks will not take advantage of these technologies. In addition, our pattern allows the integration with existing web applications de-

veloped with typical WAFs; furthermore Jadex Webbridge framework can not be integrated with this kind of frameworks, widely used in the web application development.

6.3 Final Remarks

This Chapter presented works related to two different topics of this dissertation: (i) approaches for developing MAS-PLs; and (ii) approaches that incorporate software agents into web applications. Only two recent research works have explored the integration synergy between SPLs and MASs. Pena et al. proposed a process based on the MaCMAS methodology to build the core architecture of MAS-PLs. Dehlinger & Lutz proposed a requirements specification template for agent-based applications that supports safe reuse. Both approaches present some deficiencies, which were described in this Chapter. Additionally, we have detailed approaches for integrating agents and web applications, indicating their weaknesses and how they were addressed in the Web-MAS architectural pattern.

7
Conclusion and Future Work

On the one hand, SPL is a new trend of software reuse, which investigates methods and techniques in order to promote reduced time-to-market, lower development costs and quality improvement on the development of applications that share common and variable features. On the other hand, AOSE is a new software engineering paradigm to allow the development of distributed complex applications, which are characterized by a system composed of many interrelated subsystems. In order to incorporate the benefits of these two software engineering disciplines and to help the industrial exploitation of agent technology, Multi-agent System Product Lines (MAS-PLs) have emerged integrating Software Product Lines and Agent-oriented Software Engineering techniques.

In this dissertation, we presented a domain engineering process for developing MAS-PLs. We specified our process according to SPEM, defining its phases, activities, tasks, roles and work products. We provided specific activities concerned in analyzing, designing and implementing agent features. Our approach was defined based on existing best practices of existing SPL and MAS approaches: PLUS is a SPL method based on UML that provides notations for documenting variability; PASSI is an agent-oriented methodology, which proposes some diagrams that are used to specify agent features at analysis level; and MAS-ML is the modeling language used in the domain design phase to model agents, roles and organizations. We also proposed adaptations and extensions to these approaches to address variability and traceability of agent features. Additionally, some design and implementation guidelines are proposed in order to help in agent features modularization. An advantage of the approach resides in the fact that we separate the modeling of agent features, and it makes it possible to evolve existing systems to incorporate new features that take advantage of agent abstraction. We have also discussed limitations of our approach, which include techniques to verify consistency among our process models. We developed our process with the experience of two cases studies: the ExpertCommittee case study, which is a product line of conference management systems and the OLIS case study, which is a product line of web

applications that provide several personal services to users. These case studies were modeled and documented with our domain engineering process and were used to evaluate and illustrate our approach.

7.1 Contributions

As the result of the work presented in this dissertation, the following contributions can be enumerated:

SPL and MAS-PL Approaches Comparison. Several SPL approaches have been proposed. In order to understand these approaches and analyze how they can deal with the documentation and modeling of agent features, we studied and compared some existing SPL and MAS-PL approaches (Nunes 2008a). In our comparative study (Chapter 2), we used the same evaluation framework of (Matinlassi 2004) to compare them. The goal of this evaluation was to obtain an overview of these approaches and not necessarily to rate them.

Domain Engineering Process. The main contribution of this dissertation is the definition of a systematic process (Nunes 2009b, Nunes 2009c) to perform domain engineering, which includes the phases of Domain Analysis, Domain Design, and Domain Realization, based on a set of activities, tasks, work products, roles. The domain engineering phases specify: (i) how to analyze, understand and document MAS-PL agent and non-agent features (Domain Analysis) (Nunes 2009d, Nunes 2009e); (ii) how to design a MAS-PL reference architecture by modeling its features (Domain Design); and (iii) how to implement MAS-PLs (Domain Realization). Moreover, the process was specified using SPEM (OMG 2008), which was proposed by OMG. It provides a way of documenting processes so that they may be studied, understood, and evolved in a logical, standardized fashion.

Documenting and Tracing Agent Features. In order to provide agent features documentation and tracing, we integrated notations from SPL and MAS approaches, and incorporated additional extensions. Stereotypes, UML 2.0 frames and structured activities were introduced in agent specific diagrams, such as role identification, task specification and organization diagrams, to provide explicit variability information. Additionally, the capability concept was used to modularize agent and role variabilities. A Feature/Agent dependency model was proposed providing agent features traceability.

Separated Modeling of Agent Features. MAS methodologies usually propose to distribute all system functionalities / responsibilities among agents. Agents are an abstraction that provides some particular characteristics, such as autonomy and pro-activeness. Therefore, in our process, we define an activity whose purpose is to identify agent features, which are features that the agent abstraction is appropriate, and then they are modeled in particular activities. We claim that features of MAS-PLs that do not take advantage from agent technology can be modeled and implemented using existing SPL approaches. This explicit separation allows non-agent features to take advantage of other existing SPL techniques adopted to implement variabilities.

Two MAS-PL Case Studies Development. In order to identify problems on MAS-PL development and deficiencies of current approaches to modeling MAS-PLs, we developed the ExpertCommittee case study (Nunes 2008d, Nunes 2009f) (Chapter 4). This case study was later used to evaluate our process using an extractive SPL development strategy. In addition, a second case study, the OLIS, was developed using a reactive approach with the purpose of evaluating and testing the effectiveness of our process. Given that little research effort has been made in MAS-PL development, there are only a few case studies in this context, which are essential to validate proposed approaches.

Design and Implementation Guidelines. Based on our experience on MAS-PL development, we stated design and implementation guidelines with the goal of providing a better agent features modularization. These guidelines were also derived from the development experience of our case studies. We presented guidelines in three directions: (i) integration of web applications with software agents; (ii) implementing roles with two widely used agent platforms (JADE and Jadex); and (iii) use of Aspect-oriented Programming to provide a better feature modularization. Among these guidelines, the major contribution is the proposal of the Web-MAS Architectural Pattern (Nunes 2008b), which defines a general structure to add autonomous behavior to existing web applications using agent technology.

Besides the contributions of this dissertation, we also conducted research in cooperation with other members of our research group related to the work presented here. The use of AOP to implement agent features is exploited in (Nunes 2008c) and (Nunes 2009a). (Nunes 2008c) presents a quantitative study of development and evolution of the EC case study. (Nunes 2009a)

presents a comparison of the stability of implementation techniques for MAS-PLs. In the context of application engineering for MAS-PLs, (Cirilo 2008c) proposes an extension of the GenArch tool (Cirilo 2008a) in order to provide automatic product derivation.

7.2 Future Work

These contributions represent a first effort at supporting the development of MAS-PLs. In spite of the benefits identified in the use of the proposed approach, there are many ongoing and future works, some of which are described in the following.

Dealing with Fine-grained Features. In the literature, there are many examples of SPLs with coarse-grained features. This means that these features can be implemented wrapped in a specific unit, such as a class, a method or an agent. However, fine-grained extensions, e.g. adding a statement in the middle of a method, usually require the use of techniques, like conditional compilation, which obfuscates the base code with annotations. Though many SPLs have been implemented at the coarse granularity level using existing approaches, fine-grained extensions are also essential when extracting features from legacy applications (Kästner 2008). Our scope in this dissertation was dealing with coarse-grained features; however it is important to study ways of completely modularizing fine-grained agent features at analysis, design and implementation levels.

Use of Aspect-oriented Analysis and Design Approaches. Our domain engineering process provides support to crosscutting features by using specific notations to indicate variability introduced by them. Nevertheless, some aspect-oriented analysis and design approaches (Chitchyan 2005) have been proposed in order to identify and modularize crosscutting features, which are in this case seen as crosscutting concerns. Crosscutting concerns (Rashid 2003) refers to such quality factors or functionalities of software that cannot be effectively modularized using existing software development techniques. So, we are currently investigating how existing aspect-oriented modeling approaches (Jacobson 2004, Clarke 2005) can help the visual documentation of the agent features.

Experimental Studies to Evaluate our Process. We applied and evaluated our process based on the development of two case studies. Furthermore, some

activities of our approach, mainly related to the modeling of agent features, were adopted in the development of design and implementation projects in the Software Systems Design graduate course at PUC-Rio. However, empirical studies are very important to be performed given that the progress in any discipline depends on our ability to understand the basic units necessary to solve a problem (Basili 1996). Moreover, experimentation provides a systematic, disciplined, quantifiable, and controlled way to evaluate new theories. As a consequence, our future work includes making experimental studies to see how and when our process really works, to understand its limitations and to understand how to improve it.

Framework for Integrating Web-applications and Software Agents. Based on the development of our case studies, we have derived the Web-MAS Architectural Pattern. Besides these case studies share the same architecture, there are several elements such as classes that are used in both of them. Examples are the environment and facade agents, and classes that implement the Observer pattern. Therefore, a framework for developing web applications that have software agents on their architecture may be built to facilitate the development of this kind of application.

Tool Support. Our domain engineering process proposes the use of MAS-ML to model agent features and some models to provide feature traceability. However, none of them have tool support to facilitate the generation of diagrams. This is an essential motivation for the adoption of an approach. In the Software Engineering Laboratory at PUC-Rio, a prototype for designing MAS-ML diagrams has already been developed. In addition, some model-based derivation tools have been proposed in order to automate the derivation process of SPLs. One of them is the GenArch (Cirilo 2008a) tool, which was recently extended (Cirilo 2008c) by the incorporation of a new domain-specific architecture model for Jadex to enable the automatic instantiation and customization of MAS-PLs. There are some ongoing research work, whose aim is to integrate our models with GenArch.

Bibliography

[Almeida 2007] DE ALMEIDA, E. S.. **RiDE: The RiSE Process for Domain Engineering**. PhD thesis, Federal University of Pernambuco (UFPE), Brazil, May 2007.

[Alur 2001] ALUR, D.; MALKS, D. ; CRUPI, J.. **Core J2EE Patterns: Best Practices and Design Strategies**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Alves 2006] ALVES, V.; GHEYI, R.; MASSONI, T.; KULESZA, U.; BORBA, P. ; LUCENA, C.. **Refactoring product lines**. In: GPCE '06: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, p. 201–210, New York, NY, USA, 2006. ACM.

[Alves 2007] ALVES, V.. **Implementing Software Product Line Adoption Strategies**. PhD thesis, Federal University of Pernambuco (UFPE), Brazil, March 2007.

[Antkiewicz 2004] ANTKIEWICZ, M.; CZARNECKI, K.. **Featureplugin: feature modeling plug-in for eclipse**. In: ECLIPSE '04: PROCEEDINGS OF THE 2004 OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, p. 67–72, New York, NY, USA, 2004. ACM.

[Apache 2008] APACHE SOFTWARE FOUNDATION. **Apache struts framework**, 2008. http://struts.apache.org/.

[Atkinson 2000] ATKINSON, C.; BAYER, J. ; MUTHIG, D.. **Component-based product line development: The KobrA approach**. In: Donohoe, P., editor, PROCEEDINGS OF theFirstSoftware PRODUCT LINE CONFERENCE, p. 289–309, 2000.

[Atkinson 2002] ATKINSON, C.; BAYER, J.; BUNSE, C.; KAMSTIES, E.; LAITENBERGER, O.; LAQUA, R.; MUTHIG, D.; PAECH, B.; WÜST, J. ; ZETTEL, J.. **Component-based product line engineering with UML**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Basili 1996] BASILI, V. R.. **The role of experimentation in software engineering: past, current, and future**. In: ICSE '96: PROCEEDINGS OF THE 18TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 442–449, Washington, DC, USA, 1996. IEEE Computer Society.

[Baumer 1997] BÄUMER, D.; RIEHLE, D.; SIBERSKI, W. ; WULF, M.. **The role object pattern**. In: PROCEEDINGS OF THE 4TH ANNUAL CONFERENCE ON THE PATTERN LANGUAGES OF PROGRAMS (PLOP, Monticello, Illinois, USA, 1997.

[Bauer 2001] BAUER, B.; MÜLLER, J. P. ; ODELL, J.. **Agent uml: a formalism for specifying multiagent software systems**. In: FIRST INTERNATIONAL WORKSHOP, AOSE 2000 ON AGENT-ORIENTED SOFTWARE ENGINEERING, p. 91–103, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[Bauer 2002] BAUER, B.. **Uml class diagrams revisited in the context of agent-based systems**. In: AOSE '01: REVISED PAPERS AND INVITED CONTRIBUTIONS FROM THE SECOND INTERNATIONAL WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING II, p. 101–118, London, UK, 2002. Springer-Verlag.

[Bayer 1999] BAYER, J.; FLEGE, O.; KNAUBER, P.; LAQUA, R.; MUTHIG, D.; SCHMID, K.; WIDEN, T. ; DEBAUD, J.-M.. **Pulse: a methodology to develop software product lines**. In: SSR '99: PROCEEDINGS OF THE 1999 SYMPOSIUM ON SOFTWARE REUSABILITY, p. 122–131, New York, NY, USA, 1999. ACM.

[Bellifemine 2007] BELLIFEMINE, F. L.; CAIRE, G. ; GREENWOOD, D.. **Developing Multi-Agent Systems with JADE**. John Wiley & Sons, 2007.

[Beydeda 2005] BEYDEDA, S.; GRUHN, V.. **Model-Driven Software Development**. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Bordini 2007] BORDINI, R. H.; WOOLDRIDGE, M. ; HÜBNER, J. F.. **Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)**. John Wiley & Sons, 2007.

[Bresciani 2004] BRESCIANI, P.; PERINI, A.; GIORGINI, P.; GIUNCHIGLIA, F. ; MYLOPOULOS, J.. **Tropos: An agent-oriented software devel-**

opment methodology. Autonomous Agents and Multi-Agent Systems, 8(3):203–236, 2004.

[Buschmann 1996] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M.; SOMMERLAD, P. ; STAL, M.. **Pattern-Oriented Software Architecture: A System of Patterns**. John Wiley Sons, 1996.

[Chella 2006] CHELLA, A.; COSSENTINO, M.; SABATUCCI, L. ; SEIDITA, V.. **Agile passi: An agile process for designing agents**. International Journal of Computer Systems Science & Engineering, 21(2), 2006.

[Chitchyan 2005] CHITCHYAN, R.; RASHID, A.; SAWYER, P.; GARCIA, A.; ALARCON, M. P.; BAKKER, J.; TEKINERDOGAN, B.; CLARKE, S. ; JACKSON, A.. **Survey of analysis and design approaches**. Technical report, AOSD-Europe, May 2005.

[Choy 2005] CHOY, S.-O.; NG, S.-C. ; TSANG, Y.-C.. **Building software agents to assist teaching in distance learning environments**. In: ICALT '05: PROCEEDINGS OF THE FIFTH IEEE INTERNATIONAL CONFERENCE ON ADVANCED LEARNING TECHNOLOGIES, p. 230–232, Washington, DC, USA, 2005. IEEE Computer Society.

[Ciancarini 1998] CIANCARINI, P.; NIERSTRASZ, O. ; TOLKSDORF, R.. **A case study in coordination: Conference management on the internet**, 1998. http:www.cs.unibo.it/cianca/wwwpages/case.ps.gz.

[Cirilo 2008a] CIRILO, E.; KULESZA, U. ; LUCENA, C.. **A Product Derivation Tool Based on Model-Driven Techniques and Annotations**. Journal of Universal Computer Science, 14:1344–1367, 2008.

[Cirilo 2008b] CIRILO, E.; KULESZA, U.; COELHO, R.; LUCENA, C. ; VON STAA, A.. **Integrating Component and Product Lines Technologies**. In: ICSR '08: 10TH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, China, 2008.

[Cirilo 2008c] CIRILO, E.; NUNES, I.; KULESZA, U.; NUNES, C. ; DE LUCENA, C. J.. **Automatic product derivation of multi-agent systems product lines**. In: PROCEEDINGS OF THE 2009 ACM SYMPOSIUM ON APPLIED COMPUTING (SAC 2009), p. 731–732, Hawaii, USA, March 2009.

[Clarke 2005] CLARKE, S.; BANIASSAD, E.. **Aspect-Oriented Analysis and Design: The Theme Approach (The Addison-Wesley Object Technology Series)**. Addison-Wesley Professional, March 2005.

[Clements 2002] CLEMENTS, P.; NORTHROP, L.. **Software Product Lines: Practices and Patterns**. Addison-Wesley, Boston, MA, USA, 2002.

[Cossentino 2002] COSSENTINO, M.; BURRAFATO, P.; LOMBARDO, S. ; SABATUCCI, L.. **Introducing pattern reuse in the design of multi-agent systems**. In: AGENT TECHNOLOGIES, INFRASTRUCTURES, TOOLS, AND APPLICATIONS FOR E-SERVICES, p. 107–120, 2002.

[Cossentino 2003] COSSENTINO, M.; SABATUCCI, L. ; CHELLA, A.. **Patterns reuse in the passi methodology**. In: ESAW'03, p. 29–31. Springer-Verlag, 2003.

[Cossentino 2005] COSSENTINO, M.. **From Requirements to Code with the PASSI Methodology**, chapter IV. Idea Group Inc., Hershey, PA, USA, 2005.

[Czarnecki 2000] CZARNECKI, K.; EISENECKER, U.. **Generative Programming: Methods, Tools, and Applications**. Addison Wesley Longman, 2000.

[Czarnecki 2006] CZARNECKI, K.; HELSEN, S.. **Feature-based survey of model transformation approaches**. IBM Systems Journal, 45(3):621–645, 2006.

[Dehlinger 2007] DEHLINGER, J.; LUTZ, R. R.. **A Product-Line Requirements Approach to Safe Reuse in Multi-Agent Systems**. In: SELMAS '05: PROCEEDINGS OF THE FOURTH INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS, p. 1–7, New York, NY, USA, 2005. ACM Press.

[Fayad 1999] FAYAD, M.; SCHMIDT, D. ; JOHNSON, R.. **Building application frameworks: object-oriented foundations of framework design**. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Figueiredo 2008] FIGUEIREDO, E.; CACHO, N.; SANT'ANNA, C.; MONTEIRO, M.; KULESZA, U.; GARCIA, A.; SOARES, S.; FERRARI, F.; KHAN, S.; FILHO, F. ; DANTAS, F.. **Evolving software product lines with aspects: An empirical study on design stability**. In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), May 2008.

[Ford 2004] FORD, N.. **Art of Java web development: Struts, Tapestry, Commons, Velocity, JUnit, Axis, Cocoon, InternetBeans, WebWorks**. Manning Publications Co., 2004.

[Fowler 2002] FOWLER, M.. **Patterns of Enterprise Application Architecture**. Addison-Wesley Professional, November 2002.

[Gamma 1995] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns: Elements of Reusable Object-oriented Software**. Addison-Wesley, 1995.

[Garcia 2004] GARCIA, A.; LUCENA, C. ; COWAN, D.. **Agents in object-oriented software engineering**. Software Practice Experience, 34(5):489–521, 2004.

[Garcia 2005] GARCIA, A.; CHAVEZ, C.; KULESZA, U. ; LUCENA, C.. **The role aspect pattern**. In: EUROPLOP2005, Isree, Germany., 2005.

[Girardi 2002] GIRARDI, R.. **Reuse in agent-based application development**.

[Gomaa 2004] GOMAA, H.. **Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures**. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Gonzalez-Palacios 2004] GONZALEZ-PALACIOS, J.; LUCK, M.. **A framework for patterns in gaia: A case-study with organisations**. In: AOSE, p. 174–188, 2004.

[Greenfield 2004] GREENFIELD, J.; SHORT, K.; COOK, S. ; KENT, S.. **Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools**. John Wiley and Sons, 2004.

[Griss 1997] GRISS, M. L.. **Software Reuse: Architecture, Process, and Organization for Business Success**. In: ICCSSE '97: PROCEEDINGS OF THE 8TH ISRAELI CONFERENCE ON COMPUTER-BASED SYSTEMS AND SOFTWARE ENGINEERING, p. 86, Washington, DC, USA, 1997. IEEE Computer Society.

[Griss 1998] GRISS, M. L.; FAVARO, J. ; METHODOLOGIST, C.. **Integrating feature modeling with the rseb**. In: ICSR'98: IN PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, p. 76–85. IEEE Computer Society, 1998.

[Holz 2008] Holz, H.; Hofmann, K. ; Reed, C., editors. **Personalization Techniques and Recommender Systems**, volumen 70. World Scientific Publishing, April 2008.

[Howden 2001] HOWDEN, N.; RÖNNQUISTA, R.; HODGSON, A. ; LUCAS, A.. **Jack intelligent agents^{TM}: Summary of an agent infrastructure**. In: THE FIFTH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS, Montreal, Canada, 2001.

[Jacobson 2004] JACOBSON, I.; NG, P.-W.. **Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)**. Addison-Wesley Professional, 2004.

[Jennings 2001] JENNINGS, N. R.. **An agent-based approach for building complex software systems**. Communications of the ACM, 44(4):35–41, 2001.

[Kang 1990] KANG, K.; COHEN, S.; HESS, J.; NOVAK, W. ; PETERSON. **Feature-oriented domain analysis (foda) feasibility study**. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.

[Kang 1998] KANG, K. C.; KIM, S.; LEE, J.; KIM, K.; SHIN, E. ; HUH, M.. **Form: A feature-oriented reuse method with domain-specific reference architectures**. Ann. Softw. Eng., 5:143–168, 1998.

[Kang 2002] KANG, K. C.; LEE, J. ; DONOHOE, P.. **Feature-oriented project line engineering**. IEEE Softw., 19(4):58–65, 2002.

[Kästner 2008] KÄSTNER, C.; APEL, S. ; KUHLEMANN, M.. **Granularity in software product lines**. In: ICSE '08: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 311–320, New York, NY, USA, 2008. ACM.

[Kendall 1999] KENDALL, E. A.. **Role model designs and implementations with aspect-oriented programming**. In: OOPSLA, p. 353–369, 1999.

[Kiczales 1997] KICZALES, G.; LAMPING, J.; MENHDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M. ; IRWIN, J.. **Aspect-Oriented Programming**. In: PROCEEDINGS EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, volumen 1241, p. 220–242, Berlin, Heidelberg, and New York, June 1997. Springer-Verlag.

[Krueger 2001] KRUEGER, C. W.. **Easing the transition to software mass customization**. In: PFE '01: REVISED PAPERS FROM THE 4TH INTERNATIONAL WORKSHOP ON SOFTWARE PRODUCT-FAMILY ENGINEERING, p. 282–293, London, UK, 2002. Springer-Verlag.

[Kulesza 2006a] KULESZA, U.; GARCIA, A. F. ; LUCENA, C.. **Composing object-oriented frameworks with aspect-oriented programming**. Technical report, PUC-Rio, Computer Science Department, April 2006.

[Kulesza 2006b] KULESZA, U.; ALVES, V.; GARCIA, A. F.; DE LUCENA, C. J. P. ; BORBA, P.. **Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming**. In: ICSR'06, p. 231–245, Torino, June 2006.

[Lee 2006] LEE, J.; MUTHIG, D.. **Feature-oriented variability management in product line engineering**. Communications of the ACM, 49(12):55–59, 2006.

[Lind 2002] LIND, J.. **Patterns in agent-oriented software engineering**. In: AOSE, p. 47–58, 2002.

[Matinlassi 2004] MATINLASSI, M.. **Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada**. In: ICSE '04: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 127–136, Washington, DC, USA, 2004. IEEE Computer Society.

[Muthig 2002] MUTHIG, D.; ATKINSON, C.. **Model-driven product line architectures**. In: SPLC 2: PROCEEDINGS OF THE SECOND INTERNATIONAL CONFERENCE ON SOFTWARE PRODUCT LINES, p. 110–129, London, UK, 2002. Springer-Verlag.

[Nunes 2008a] NUNES, I.; NUNES, C.; KULESZA, U. ; DE LUCENA, C. J.. **Documenting and modeling multi-agent systems product lines**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING (SEKE 2008), p. 745–751, Redwood City, San Francisco Bay, USA, July 2008.

[Nunes 2008b] NUNES, I.; KULESZA, U.; NUNES, C.; CIRILO, E. ; DE LUCENA, C. J.. **Extending web-based applications to incorporate autonomous behavior**. In: XIV BRAZILIAN SYMPOSIUM ON MULTIMEDIA AND THE WEB (WEBMEDIA 2008), p. 115–122, Vila Velha, Brazil, October 2008.

[Nunes 2008c] NUNES, C.; KULESZA, U.; SANT'ANNA, C.; NUNES, I. ; DE LUCENA, C. J.. **On the modularity assessment of aspect-oriented multi-agent systems product lines: a quantitative study**. In: SECOND BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES, AND REUSE (SBCARS 2008), p. 122–135, Porto Alegre, Brazil, August 2008.

[Nunes 2008d] NUNES, I.; NUNES, C.; KULESZA, U. ; DE LUCENA, C. J.. **Developing and evolving a multi-agent system product line: An exploratory study**. In: PROCEEDINGS OF 9TH INTERNATIONAL WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING (AOSE 2008), p. 177–188, Estoril, Portugal, May 2008.

[Nunes 2009a] NUNES, C.; KULESZA, U.; SANT'ANNA, C.; NUNES, I.; GARCIA, A. ; DE LUCENA, C. J.. **Comparing stability of implementation techniques for multi-agent system product lines**. In: 13TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR 2009), Kaiserslautern, Germany, March 2009.

[Nunes 2009b] NUNES, I.; KULESZA, U.; NUNES, C. ; DE LUCENA, C. J.. **Domain engineering process for developing multi-agent systems product lines (to appear)**. In: Sierra, C.; Castelfranchi, C.; Sichman, J. S. ; Decker, K. S., editors, PROCEEDINGS OF 8TH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS (AAMAS 2009), Budapest, Hungary, May 2009.

[Nunes 2009c] NUNES, I.; DE LUCENA, C. J.; KULESZA, U. ; NUNES, C.. **On the development of multi-agent systems product lines: A domain engineering process (to appear)**. In: PROCEEDINGS OF 10TH INTERNATIONAL WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING (AOSE 2009), Budapest, Hungary, May 2009.

[Nunes 2009d] NUNES, I.; KULESZA, U.; NUNES, C.; CIRILO, E. ; DE LUCENA, C. J.. **Extending passi to model multi-agent systems product lines**. In: PROCEEDINGS OF THE 2009 ACM SYMPOSIUM ON APPLIED COMPUTING (SAC 2009), p. 729–730, Honolulu, Hawaii, USA, March 2009.

[Nunes 2009e] NUNES, I.; KULESZA, U.; NUNES, C.; DE LUCENA, C. J. ; CIRILO, E.. **A domain analysis approach for multi-agent systems product lines (to appear)**. In: 11TH INTERNATIONAL CONFERENCE

ON ENTERPRISE INFORMATION SYSTEMS (ICEIS 2009), Milan, Italy, May 2009.

[Nunes 2009f] NUNES, I.; NUNES, C.; KULESZA, U. ; DE LUCENA, C. J.. **Developing and evolving a multi-agent system product line: An exploratory study**. In: Luck, M.; Sanz, J. J. G., editors, AGENT-ORIENTED SOFTWARE ENGINEERING IX, volumen 5386 de **Lecture Notes in Computer Science (LNCS)**, p. 229–242. Springer Berlin / Heidelberg, 2009.

[OMG 2008] OBJECT MANAGEMENT GROUP (OMG). **Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0**, April 2008.

[Padgham 2000] PADGHAM, L.; LAMBRIX, P.. **Agent capabilities: Extending bdi theory**. In: PROCEEDINGS OF THE SEVENTEENTH NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND TWELFTH CONFERENCE ON INNOVATIVE APPLICATIONS OF ARTIFICIAL INTELLIGENCE (AAAI 2000), p. 68–73. AAAI Press / The MIT Press, 2000.

[Parnas 1976] PARNAS, D. L.. **On the design and development of program families**. IEEE Trans. Software Eng., 2(1):1–9, 1976.

[Pena 2005] PENA, J.. **On improving the modeling of complex acquaintance organisations of agents. A method fragment for the analysis phase.** PhD thesis, University of Seville, 2005.

[Pena 2006a] PENA, J.; HINCHEY, M. G.; RUIZ-CORTÉS, A. ; TRINIDAD., P.. **Building the core architecture of a multiagent system product line: with an example from a future nasa mission.** In: AOSE '06: 7TH INTERNATIONAL WORKSHOP ON AGENT ORIENTED SOFTWARE ENGINEERING. LNCS, 2006.

[Pena 2006b] PENA, J.; HINCHEY, M. G. ; RUIZ-CORTÉS, A.. **Multi-agent system product lines: challenges and benefits**. Communications of the ACM, 49(12):82–84, 2006.

[Pena 2006c] PENA, J.; HINCHEY, M. G.; RESINAS, M.; STERRITT, R. ; RASH, J. L.. **Managing the Evolution of an Enterprise Architecture Using a MAS-Product-Line Approach.** In: Arabnia, H. R.; Reza, H., editors, SOFTWARE ENGINEERING RESEARCH AND PRACTICE, p. 995–1001. CSREA Press, 2006.

[Pena 2007] PENA, J.; HINCHEY, M. G.; RESINAS, M.; STERRITT, R. ; RASH, J. L.. **Designing and managing evolving systems using a MAS product line approach**. Science of Computer Programming, 66(1):71–86, 2007.

[Pohl 2005] POHL, K.; BÖCKLE, G. ; VAN DER LINDEN, F. J.. **Software Product Line Engineering: Foundations, Principles and Techniques**. Springer-Verlag, New York,USA, 2005.

[Pokahr 2005] POKAHR, A.; BRAUBACH, L. ; LAMERSDORF, W.. **Jadex: A bdi reasoning engine**. In: R. Bordini, M. Dastani, J. D.; Seghrouchni, A. E. F., editors, MULTI-AGENT PROGRAMMING, p. 149–174. Springer Science+Business Media Inc., USA, 9 2005. Book chapter.

[Pokahr 2007a] POKAHR, A.; BRAUBACH, L.. **An architecture and framework for agent-based web applications**. In: CEEMAS '07: 5TH INTERNATIONAL CENTRAL AND EASTERN EUROPEAN CONFERENCE ON MULTI-AGENT SYSTEMS, p. 304–306, 2007.

[Pokahr 2007b] POKAHR, A.; BRAUBACH, L.. **The webbridge framework for building web-based agent applications**. In: LADS '07: LANGUAGES, METHODOLOGIES AND DEVELOPMENT TOOLS FOR MULTI-AGENT SYSTEMS, p. 173–190, 2007.

[Prieto-Diaz 1999] PRIETO-DIAZ, R.; ARANGO, G.. **Domain Analysis and Software Systems Modeling**. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991.

[Rao 1995] RAO, A. S.; GEORGEFF, M. P.. **BDI-agents: from theory to practice**. In: PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON MULTIAGENT SYSTEMS (ICMAS 1995), San Francisco, 1995.

[Rashid 2003] RASHID, A.; MOREIRA, A. ; ARAÚJO, J.. **Modularisation and composition of aspectual requirements**. In: AOSD '03: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, p. 11–20, New York, NY, USA, 2003. ACM.

[Satyananda 2007] SATYANANDA, T. K.; LEE, D.; KANG, S. ; HASHMI, S. I.. **Identifying traceability between feature model and software architecture in software product line using formal concept analysis**. In: ICCSA '07: PROCEEDINGS OF THE THE 2007 INTERNATIONAL

CONFERENCE COMPUTATIONAL SCIENCE AND ITS APPLICATIONS, p. 380–388, Washington, DC, USA, 2007. IEEE Computer Society.

[Sellers 2005] Sellers, B. H.; Giorgini, P., editors. **Agent-Oriented Methodologies**. Idea Group Inc., 2005.

[Shaw 1996] SHAW, M.; GARLAN, D.. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice Hall, April 1996.

[Shehory 2001] SHEHORY, O.; STURM, A.. **Evaluation of modeling techniques for agent-based systems**. In: AGENTS '01: PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS, p. 624–631, 2001.

[Silva 2003a] DA SILVA, V. T.; DE LUCENA, C. J. P.. **Mas-ml: a multi-agent system modeling language**. In: OOPSLA '03: COMPANION OF THE 18TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, p. 126–127, New York, NY, USA, 2003. ACM.

[Silva 2003b] SILVA, V.; GARCIA, A.; BRANDÃO, A.; CHAVEZ, C.; LUCENA, C. ; ALENCAR, P.. **Taming agents and objects in software engineering**. In: SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS: RESEARCH ISSUES AND PRACTICAL APPLICATIONS, VOLUME LNCS 2603, p. 1–26. Springer-Verlag, 2003.

[Silva 2004a] DA SILVA, V. T.; CHOREN, R. ; DE LUCENA, C. J. P.. **A uml based approach for modeling and implementing multi-agent systems**. In: AAMAS '04: PROCEEDINGS OF THE THIRD INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, p. 914–921, Washington, DC, USA, 2004. IEEE Computer Society.

[Silva 2004b] DA SILVA, V. T.; DE LUCENA, C. J. P.. **From a conceptual framework for agents and objects to a multi-agent system modeling language**. Autonomous Agents and Multi-Agent Systems, 9(1-2):145–189, 2004.

[Silva 2007] DA SILVA, V. T.; DE LUCENA, C. J.. **Modeling multi-agent systems**. Communications of the ACM, 50(5):103–108, 2007.

[Silva 2008] DA SILVA, V. T.; CHOREN, R. ; DE LUCENA, C. J. P.. **Mas-ml: a multiagent system modelling language.** International Journal of Agent-Oriented Software Engineering (IJAOSE), 2(4):382–421, 2008.

[SpringSource 2008] SPRINGSOURCE. **Spring framework**, 2008. http://www.springsource.org/.

[Stroulia 2003] STROULIA, E.; HATCH, M. P.. **An intelligent-agent architecture for flexible service integration on the web**. IEEE Transactions on Systems, Man, and Cybernetics, Part C, 33(4):468–479, 2003.

[Sun 2008] SUN MICROSYSTEMS. **Javaserver faces technology**, 2008. http://java.sun.com/javaee/javaserverfaces/.

[Szyperski 2002] SZYPERSKI, C.. **Component Software: Beyond Object-Oriented Programming**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Weiss 1999] WEISS, D. M.; LAI, C. T. R.. **Software product-line engineering: a family-based software development process**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Wooldridge 1995] WOOLDRIDGE, M.; JENNINGS, N. R.. **Intelligent agents: Theory and practice**. Knowledge Engineering Review, 10:115–152, 1995.

[Wooldridge 1999] WOOLDRIDGE, M.. **Intelligent Agents**, chapter 1, p. 27–78. The MIT Press, London, England, 1999.

[Wooldridge 2000a] WOOLDRIDGE, M.; JENNINGS, N. R. ; KINNY, D.. **The gaia methodology for agent-oriented analysis and design**. Autonomous Agents and Multi-Agent Systems, 3(3):285–312, 2000.

[Wooldridge 2000b] WOOLDRIDGE, M.; CIANCARINI, P.. **Agent-Oriented Software Engineering: The State of the Art**. In: Ciancarini, P.; Wooldridge, M., editors, FIRST INT. WORKSHOP ON AGENT-ORIENTED SOFTWARE ENGINEERING, volumen 1957, p. 1–28. Springer-Verlag, Berlin, 2000.

[Yu 1996] YU, E. S.-K.. **Modelling strategic relationships for process reengineering**. PhD thesis, Toronto, Ont., Canada, Canada, 1996.

[Zambonelli 2000] ZAMBONELLI, F.; JENNINGS, N. R. ; WOOLDRIDGE, M.. **Organizational abstractions for the analysis and design of multi-agent systems**. In: AOSE, p. 235–251, 2000.

[Zambonelli 2003] ZAMBONELLI, F.; JENNINGS, N. R. ; WOOLDRIDGE, M.. **Developing multiagent systems: The gaia methodology**. ACM Trans. Softw. Eng. Methodol., 12(3):317–370, 2003.