PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 47/08

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

Ingrid Oliveira de Nunes

Uirá Kulesza

Camila Patrícia Bazílio Nunes

Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 47/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena November, 2008

A Domain Engineering Process for Developing Multi-agent Systems Product Lines [1]

Ingrid Oliveira de Nunes[1], Uirá Kulesza[2], Camila Patrícia Bazílio Nunes[1], Carlos José Pereira de Lucena[1]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro - Brazil
[2] Federal University of Rio Grande do Norte (UFRN) - Natal - Brazil

{ioliveira,camilan,lucena}@inf.puc-rio.br, uira@dimap.ufrn.br

Abstract. Multi-agent Systems Product Lines (MAS-PLs) have emerged to integrate two promising trends of software engineering: agent-oriented software engineering, which is a new paradigm to support the development of complex and distributed systems based on agent abstraction, and software product lines, a systematic form of reuse that addresses the development of system families that share common and variable features. In this paper, we propose a domain engineering process to develop MAS-PLs, built on top of agent-oriented and software product line approaches, addressing agency features modeling and documentation. We describe each one of the process stages, and their respective activities. Our approach is illustrated with a case study, OLIS (OnLine Intelligent Services).

Keywords: Software Product Lines, Multi-agent Systems, Process, Agent-oriented software engineering, Domain Engineering.

Resumo. Linhas de Produto de Sistemas Multi-agentes (LP-SMAs) surgiram para integrar duas promissoras tendências da engenharia de software: engenharia de software de sistemas multi-agentes, que é um novo paradigma para suportar o desenvolvimento de sistemas complexos e distribuídos baseado na abstração de agente, e linhas de produto de software, uma forma sistemática de reuso que visa o desenvolvimento de famílias de sistemas que compartilham *features* comuns e variáveis. Neste artigo, propomos um processo de engenharia de domínio para o desenvolvimento de LP-SMAs, construído com base em abordagens orientadas a agentes e de linhas de produtos de software, permitindo a documentação e modelagem de *features* de agência. Nós descrevemos cada um dos estágios do processo, e suas respectivas atividades. Nossa abordagem é ilustrada com um estudo de caso, o OLIS (OnLine Intelligent Services).

Palavras-chave: Linhas de Produto de Software, Sistemas Multi-agentes, Processo, Engenharia de Software Orientada a Agentes, Engenharia de Domínio.

[1]This work has been sponsored by Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Contents

1 Introduction

Over the past decade, software agents have become a powerful abstraction to support the development of complex and distributed systems. 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). Agent-oriented software engineering (AOSE) has emerged as a new software engineering paradigm to help on the development of MASs, and then new works were proposed in this direction, such as methodologies (Wooldridge, Jennings & Kinny 2000, Cossentino 2005, Bresciani, Perini, Giorgini, Giunchiglia & Mylopoulos 2004) and modeling languages (da Silva & de Lucena 2007). However, most of the agent-oriented methodologies do not take into account the adoption of extensive reuse practices, which have been widely used in the software engineering context to provide reduced time-to-marked, quality improvement and lower development costs.

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 product-line engineering. SPLs (Clements & Northrop 2002) refer to a family of systems sharing a common, managed set of features to satisfy the needs of a selected market and that are developed from a common set of core assets in a prescribed way. Although many SPL methodologies have been proposed (Pohl, Böckle & van der Linden 2005, Gomaa 2004), they do not detail or barely detail the modeling and documentation of SPLs that take advantage of agent technology. Only some recent research (Dehlinger & Lutz 2005, Pena, Hinchey & Ruiz-cortés 2006) has explored the integration of SPLs and MASs technologies, by incorporating their respective benefits and helping the industrial exploitation of agent technology. However, there are still many challenges (Pena, Hinchey & Ruiz-Cortés 2006) to be overcome in the Multi-agent Systems Product Lines (MAS-PLs) development.

This paper presents a domain engineering process for developing MAS-PLs. This 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-PLs. Based on this experience, we propose our process, which is built on top of some MAS and SPL approaches. We combined some techniques and notations of these approaches (Gomaa 2004, Cossentino 2005, da Silva & de Lucena 2007) and added some extensions/adaptations. The scenario that we are currently exploring is the development of MAS-PLs that have some features implemented with typical web implementation techniques, such as the use of object-oriented frameworks, and others that take advantage of agent technology providing an autonomous or pro-active behavior, denominated agency features. There are examples that illustrate the incorporation of agency features into web applications, and a typical one is the recommendation of products and information to users (Holz, Hofmann & Reed 2008). Due to the existence of lots of web applications already developed, the use of a SPL approach can provide a better modularization of these features, making this evolution with a minimum impact. In this context, two case studies were developed: the OLIS case study, which is used to illustrate our approach, and the ExpertCommittee case study, a SPL of conference management systems.

The main contributions of this paper are: (i) we provide a way to model and document agency variability; (ii) we define the activities of the whole domain engineering process for MAS-PLs; and (iii) our approach models agency features independently, making the incorporation of agents in existing systems designed with other technologies, such as object-oriented, with a low impact.

The structure of this paper is as follows: Section 2 gives an overview of our approach, describing its key concepts. In Section 3, we present the OLIS case study, which is used to illustrate our approach. The process we propose is described in Section 4. Related works are presented in Section 5, followed by Section 6 that concludes this paper.

2 Approach Overview

This section presents an overview of our approach for developing MAS-PLs, detailing its basic concepts. Current SPL methodologies (Clements & Northrop 2002, Gomaa 2004, Pohl et al. 2005) cover a great variety of SPL development activities, related to domain and application engineering, and to the processes management as well. Nevertheless, they either are too abstract, lacking of 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 key concepts that guided the elaboration of our approach are presented in Section 2.1. We have also defined the agency features granularity that we are dealing in MAS-PLs development, which is described in Section 2.2.

2.1 Key Concepts

Our domain engineering process was conceived by the integration 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 (da Silva & de Lucena 2007) modeling language. In addition, we propose additional adaptations and extensions for them. 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. Figure 1 illustrates our approach by showing how MAS-PLs are modeled in different abstraction levels.

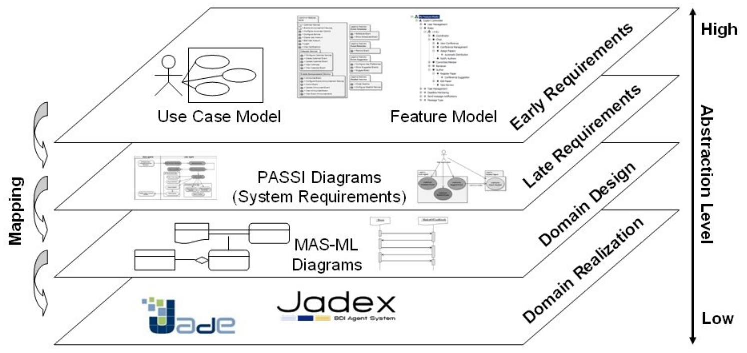

Figure 1: Approach Overview.

In (Nunes, Nunes, Kulesza & Lucena 2008), Nunes et al. investigated the use of current SPLs and MAS-PLs approaches for modeling and documenting agency features in MAS-PLs. SPL methodologies provide useful notations to model the agency features. However, none of them completely covers their specification. Particularly, the PLUS (Product Line UML-based Software engineering) (Gomaa 2004) approach was very useful for documenting agency features. PLUS provides a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle SPLs. As a consequence, this approach was the base for the elaboration of 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 agency features in MAS-PL, we have adopted the phases of the System Requirements model of PASSI methodology. We made some adaptations in these phases and they were incorporated into the domain analysis stage. PASSI (Process for Agent Societies Specification and Implementation) (Cossentino 2005) is an agent-oriented methodology that specifies five models with their respective phases for developing MASs. This methodology covers all the development process from requirements to code. PASSI follows the guideline of using standards whenever possible; and this justifies the use of UML as modeling language. This helped us to incorporate some PLUS notations into PASSI diagrams.

At the domain design stage, PASSI uses convetional 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 BDI (belief-desire-intention) model. This model proposes that agents are described in terms of three mental attitudes - believes, desires and intentions - which determine the agent's behavior. Moreover, some important agent-oriented concepts, such as environment, cannot be modeled with UML and the use of stereotypes is not enough because objects and agency elements have different properties and different relationships. Thus, our process uses the MAS-ML (da Silva & de Lucena 2007) modeling language, with some extensions, to model agents. MAS-ML extends the UML meta-model in order to express specific agent properties and relationships. As discussed in (da Silva & de Lucena 2007), others MAS modeling languages do not allow to model some agency concepts. For instance, AUML (Bauer, Müller & Odell 2001) does not define organizations and environments and as a consequence the relationships between agents and these elements cannot be modeled.

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 the models. Additionally the <<*agency feature*>> stereotype is used to indicate the use cases related to an agency feature (Section 4.1.1); (ii) use of colors to structure models in terms of features; (iii) model an agency feature in one or more models, but never two agency features in the same model; and (iv) provide the features traceability.

2.2 Agency 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 of a MAS-PL architecture. We have excluded the possibility an optional belief, for instance.

A capability (Padgham & Lambrix 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 and Jadex 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 believes, goals and plans to vary in a MAS-PL is that we believe that variations in these finegrained elements can be encapsulated into a capability.

Modularity is very important in this context because an essential engineering principle of SPLs 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.

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, Apel & Kuhlemann 2008). An example is considering two versions of a MAS, on which a belief of an agent varied between the two versions. Nevertheless, dealing with fine-grained features is out of the scope of this paper.

3 The Motivating Example: OLIS Case Study

This section briefly describes the OLIS (**O**n**L**ine **I**ntelligent **S**ervices) case study, which is used to illustrate the phases of our approach in the next section. OLIS is a MAS-PL built using a reactive development approach (Krueger 2002). This approach advocates the incremental development of SPLs. Initially, the SPL artifacts address only a few products. When there is a demand to incorporate new requirements or products, the common and variable artifacts are incrementally extended in reaction to them.The products that can be derived from OLIS MAS-PL are web applications customized to provide several personal services to users. OLIS is composed by four services: (i) User Management - allows the creation and configuration of user accounts; (ii) Events Announcement - allows the user to announce events to other system users through an events board; (iii) Calendar services - lets the user to schedule events in his/her calendar; and (iv) Weather - provides information about the current weather conditions and forecast. Additionally, OLIS provides an alternative feature: the event type, thus products can deal with generic, academic or travel events. Figure 2 depicts the OLIS features diagram.

OLIS services can be customized by the addition of optional features that automate some tasks previously done by users, which are: (i) *Events reminder* - the system sends messages to notify the user about events that are about to begin; (ii) *Events scheduler* - when an user adds a new calendar event that involves more participants, the system checks the schedule of the other participants to verify if this event conflicts with other existing ones. If so, the system suggests a new date for the calendar event that is appropriate according to schedules from all participants; and (iii) *Events Suggester* - when a new event is announced, the system automatically recommends the event after checking if it

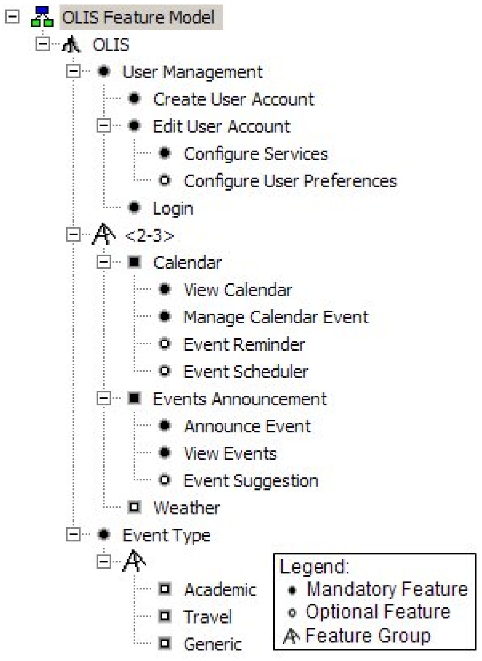

Figure 2: OLIS Features Diagram.

is interesting to the users based on their preferences. This feature is also responsible for checking if the weather is going to be appropriate according to the place type where the event is going to take place. These new features are characterized by a pro-active behavior, therefore software agents and their respective roles were useful abstractions to implement these features.

4 The Domain Engineering Process

Software product line engineering (SPLE) is in general organized in two main processes (Pohl et al. 2005): domain engineering and application engineering. Domain engineering is the process of SPLE in which the commonalities and variabilities of the SPL are identified, defined and realized. Application engineering is the process of SPLE in which applications of the SPL are built by reusing domain artifacts produced during domain engineering. Our proposal is a domain engineering process that defines the stages and their respective activities to develop MAS-PLs. It aggregates some activities that are specific to model software agents and their variabilities.

Typical domain engineering processes encompass three stages: (i) domain analysis - where the main concepts and activities in a domain are identified and modeled using adequate modeling techniques. The common and variable parts of a system family 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 - which involves implementing the architecture, components, and the production plan using appropriate technologies. The qualifier “domain” emphasizes the multisystem scope of these stages.

Table 1 summarizes the stages and activities that compose our process, and the artifacts produced in each one of them as well. Next sections detail the stages of our domain engineering process.

Stage	**Activity**		**Artifacts**
Domain Analysis	Early Requirements		
		Feature Modeling	Feature Model
		Use Case Modeling	Use Case Diagram, Use Case Descriptions
		Feature/Use Case Dependency Modeling	Feature/Use Case Dependency Model
	Late Requirements		
		Agent Identification	Agent Identification Diagram
		Role Identification	Role Identification Diagram
		Task Specification	Task Specification Diagram
		Feature/Agent Dependency Modeling	Feature/Agent Dependency Model
Domain Design	Static Modeling		Class Diagrams, Role Diagrams, Organization Diagrams
	Dynamic Modeling		Sequence Diagrams
	Feature/Agent Dependency Modeling		Refined Feature/Agent Dependency Model
Domain Realization	Assets Implementation		Reusable Software Elements
	Design/Implementation Elements Mapping		Implementation Model, Configuration Model

Table 1: The Domain Engineering Process.

4.1 Domain Analysis

The domain analysis stage 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 stage in our process is divided in two phases: Early Requirements and Late Requirements[2].

4.1.1 Early Requirements

In the Early Requirements phase, the system family is analyzed and its common and variable features are identified to establish the scope of the SPL. A feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a SPL. Posteriorly, the requirements are described in terms of use case diagrams and descriptions. Next, we detail each one of the activities that compose this phase.

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, and during the feature modeling, they are assigned to different variability categories, including mandatory, alternative and optional features. After, they are organized into a tree representation called features diagrams. Figure 2 illustrates the OLIS features diagram. A feature model refers to a features diagram accompanied by additional information such as constraints, and it represents the variability within a system family in an abstract and explicit way.

[2]These terms were inspired by Tropos(Bresciani et al. 2004) methodology.

Use Case Modeling. In this activity, the SPL functional requirements are described in terms of use cases. In our process, we have adopted the use case extended proposed by PLUS approach. In this approach, stereotypes are used to indicate that a use case: (i) is part of the SPL kernel being present in all products (<<*kernel*>>); (ii) is present only in some products (<<*optional*>>); or (iii) varies among the SPL products (<<*alternative*>>). Besides the stereotypes, we also use colors in the use cases to indicate to which feature they are related to. This color indication is used in almost all artifacts. Use case descriptions are widely used in the literature, thus we adopted them in our process using PLUS template.

According to separation of concerns principle (mentioned in Section 2.2), each use case should correspond to only one feature. 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. However, there are some features, named crosscutting features, which have impact in several use cases/features of the SPL, e.g. the event type feature in the OLIS MAS-PL impacts almost all other features and use cases of the MAS-PL. Therefore, creating separated use cases for different event types could lead to (i) a high number of use cases; and (ii) the impossibility of grouping use cases in only one categorization, for instance, the use cases could be grouped according to the functional features or the event type feature in the OLIS. So, instead of modularizing crosscutting features in specific use cases, we propose the use of variation points in the use case descriptions. PLUS does not provide explicit guidelines for modeling crosscutting features and use of variation points to describe small variations.

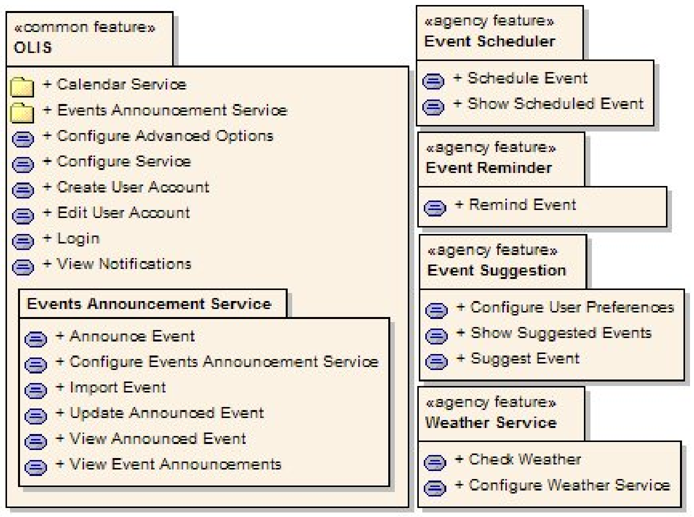

Figure 3: Feature/Use Case Dependency Diagram (Partial).

Feature/Use Case Dependency Modeling. A particularity of SPLE is that domain models should contain traceability links from the features and variation points in the feature models to their realizations in the other analysis models. So, in this activity of domain analysis, we provide another use case view (see Figure 3), which adapted from PLUS. This view maps use cases to features: use cases are grouped into features with the UML package notation plus stereotypes. In addition, we adopt a new stereotype (<<*agency feature*>>)

to indicate that the use cases of a specific package are related to an agency feature. We define an agency feature as a feature that presents some particular characteristics, such as pro-activity and autonomy, and the agent abstraction is indicated to model that feature.

4.1.2 Late Requirements

The purpose of the Late Requirements phase is to better describe the pro-active and autonomy concerns with respect to the current problem domain. A particularity of this kind of concerns is that they do not need a user that supervises their execution. Furthermore, they are not well described in use cases, and consequently they need a more precise specification. Software agents are an abstraction of the problem space that are a natural metaphor to model pro-active or autonomous behaviors of the system. Therefore, we incorporated some phases of the Domain Requirements model of PASSI methodology to our process in order to specify agency features. The Domain Requirements model generates a model of the system requirements in terms of agency and purpose. The incorporated phases with our extensions are described in Table 2. For details of the PASSI extension to model MAS-PLs in the domain analysis stage, refer to (Nunes, Kulesza, Nunes, Cirilo & Lucena 2008*a*). Next we describe the activities that encompass the Late Requirements phase.

Table 2: Our extensions to the PASSI Approach.

Phase	**Extensions**
Agent Identification	Only use cases in <<agency feature>> stereotyped packages are distributed among agents Use of Stereotypes (kernel, alternative or optional) Use of colors to trace features
Role Identification	Use of UML 2.0 frames (crosscutting features) Use of colors to trace features
Task Specification	One diagram per agent and feature Use of UML 2.0 frames (crosscutting features) Use of colors to trace features

Agent Identification. In this activity, responsibilities are attributed to agents, which are represented as stereotyped UML packages. The input of this phase is the use case diagrams generated in the Use Case Modeling activity. According to PASSI methodology, all the use cases are grouped to be performed by agents; however, we propose that only the use cases that are into a UML package stereotyped with <<*agency feature*>> will be considered. These use cases are grouped into <<*agent*>> stereotyped packages so as to form a new diagram. Each one of these packages defines the functionalities of a specific agent.

Figure 4 shows the OLIS agent identification diagram. The use case stereotypes used in the previous activity are still present. As a result, if only optional use cases are given to an agent, this agent will be also optional, for example. Additionally, the 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 stage (Section 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

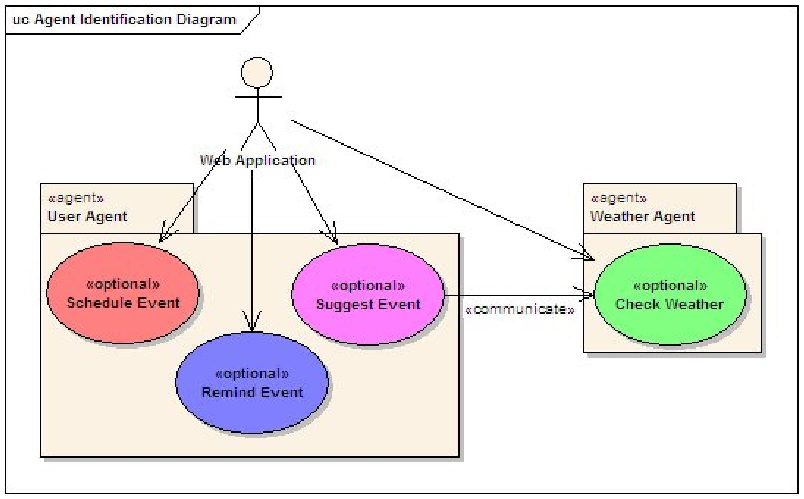

Figure 4: OLIS Agent Identification Diagram.

scenario of interacting agents working to achieve a required behavior of the system. The agent interactions are expressed through sequence diagrams.

Each role identification diagram usually corresponds to only one feature of the SPL due to the fact that use cases were already modularized in that way. Consequently, all identified roles and described interactions in the diagram are related to one specific feature. However, use cases impacted by crosscutting features are an exception for this, and the interactions related to the crosscutting feature should be explicit documented. For these cases, we propose the use of UML 2.0 frames to express optional and alternative paths in the sequence diagrams.

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.

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 the features; (ii) use of UML 2.0 frames to show different paths when there is a crosscutting feature; (iii) a colored indication showing with which feature the task is related to. These adaptations can be seen in Figure 5. The main objective of these adaptations is to provide a better feature modularization and traceability. Splitting the diagram in that way, we allow the selection of the necessary diagrams during the application engineering according to selected features.

Feature/Agent Dependency Modeling. In the Late Requirements phase, the agency features of the SPL are described in terms of agents and their respective roles. In this activity, a model is generated describing the relationships between features and these agent concepts in the SPL. This model is organized into a tree, in which each feature has agents and roles as its children indicating that these elements must be present in the product being derived if the feature is selected. Figure 6(a) shows the OLIS feature/agent dependency model.

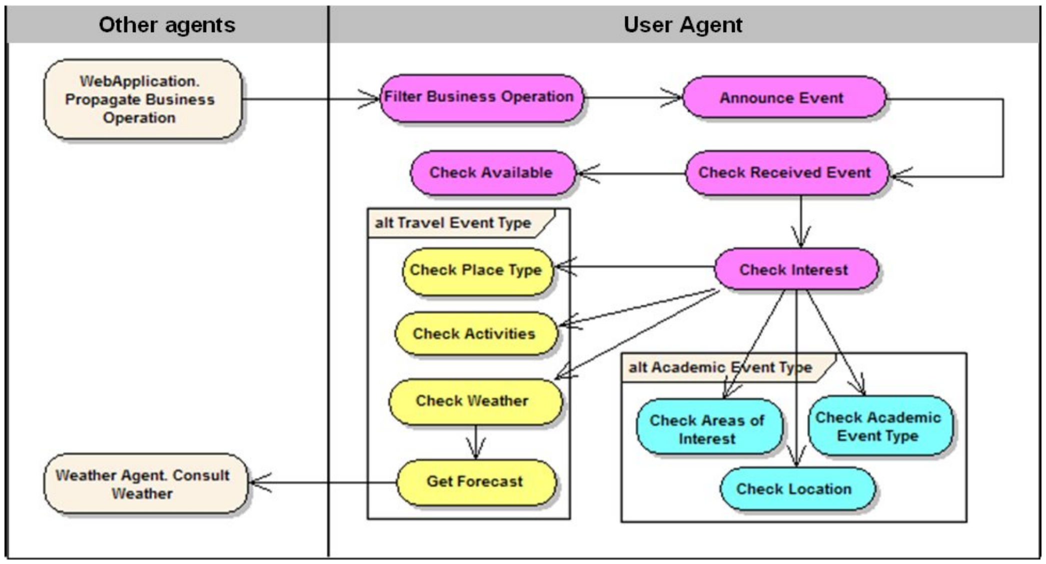

Figure 5: OLIS Task Specification Diagram.

4.2 Domain Design

The main purpose of the domain design stage is to define an architecture that addresses both the common and variable features of a SPL. Based on the SPL requirements identified on the previous stage, designers should model the SPL architecture, determining how these requirements, including the variability, are reflected 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 the design elements providing a traceability of the features. The next sections detail the activities that compose the domain design stage.

Static Modeling. The static modeling of a system determines how its elements, e.g. objects and agents, are structured. In SPLs, static modeling has also the purpose of capturing the structural aspects of the SPL; however it has additional notations to indicate the common and variable parts of the system. Moreover, when structuring a SPL, techniques, such as generalization/specialization and design patterns, should be used to model variable parts of the SPL in order to modularize features.

Instead of using UML to model MAS-PLs, we propose the use of MAS-ML (da Silva & de Lucena 2007), a MAS modeling language. It is a UML extension based on the Taming Agents and Objects (TAO) conceptual framework (meta-model). 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.

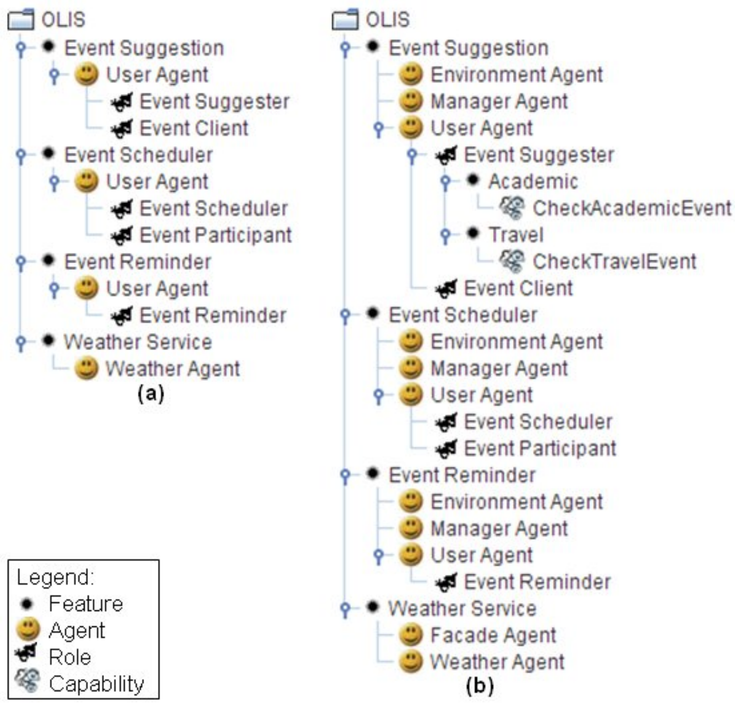

Figure 6: OLIS Feature Agent Dependency Model.

To address variability in MAS-ML diagrams, we adopted four different adaptations: (i) use of the <<*kernel*>> , <<*optional*>> and <<*alternative*>> 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 & Lambrix 2000) concept to allow the modularization of variable parts in agents and roles. We represented a capability in MAS-ML by the agent role notation with the <<*capability*>> stereotype. An aggregation relationship can be used between capabilities and agents, and capabilities and roles.

We illustrate some of the artifacts generated in the static modeling activity with two diagrams: OLIS Organization Diagram (Figure 7) and OLIS Role Diagram (Figure 8). These diagrams are related to the Event Suggestion feature. When an event is inserted by a user on the system, the *User Agent* associated with that user asks for the other user agents if they have interest on that particular event. So, the agent checks the availability of the user on the event date. Besides, if the event type is academic, the agent checks the areas of interest and the location of the event according to the user `AcademicPreferences`. And if the event type is travel, the agent checks the place type where the event is going to happen and the activities that can be done according to the user `TravelPreferences`. Finally, the user agent also consults the *Weather Agent* to get the weather forecast and check if it will be good in the event date. The implementation of this feature is accomplished by two roles: *Event Announcer* and *Event Client*. Both roles are played by the *User Agent*. The

Event Client role varies according to the event type feature. It was modularized into two capabilities: *Check Academic Event* and *Check Travel Event*. The last communicates with the *Weather Provider* role to request the forecast, what is represented by an association relationship.

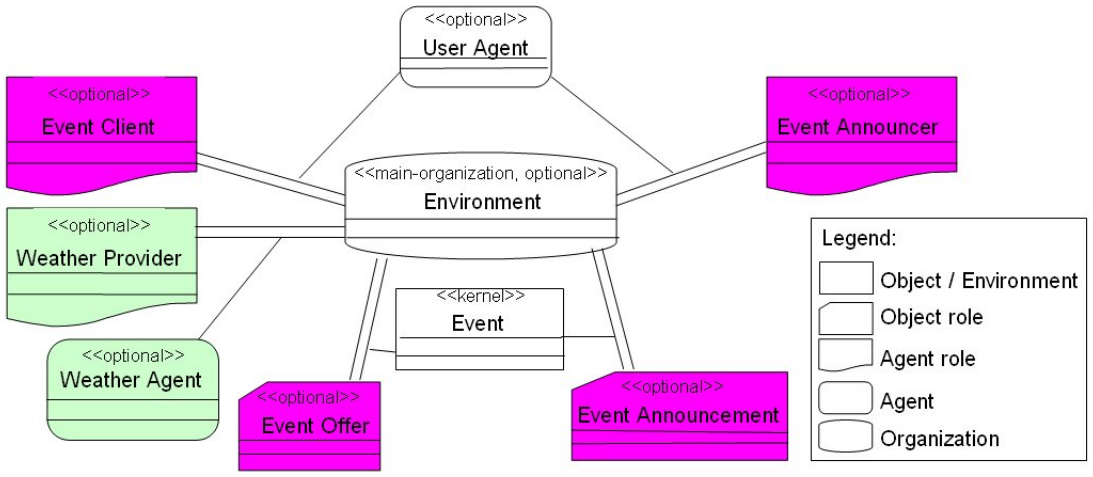

Figure 7: OLIS Organization Diagram.

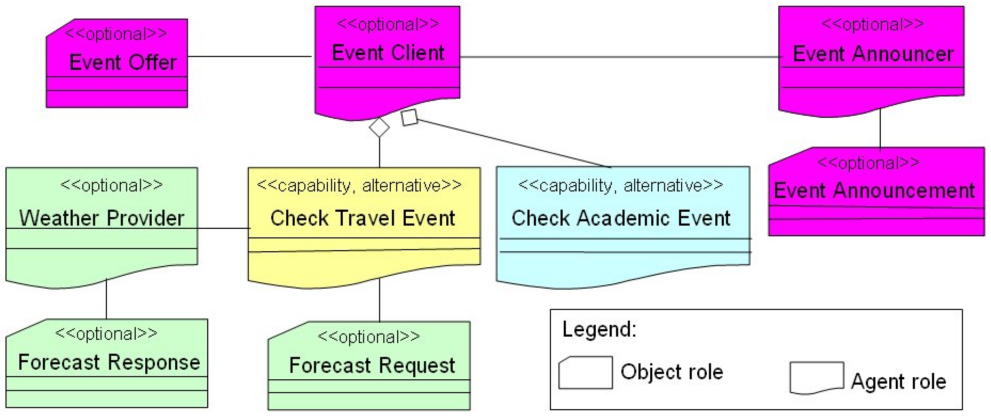

Figure 8: OLIS Role Diagram.

Dynamic Modeling. The dynamic modeling addresses interaction between objects, agents and roles, describing how these elements interact with each other. It is based on the use cases, role identification diagrams and task identification diagrams developed during domain analysis stage. For each use case, the elements that participate in the use case are determined, and the way in which the elements interact are shown in order to satisfy the requirements described in the use case. Information about the autonomous and pro-active behavior of the system is provided by the role identification and task identification diagrams.

To model the dynamic behavior in MAS-PLs, we use UML sequence diagrams, which present a set of interactions between objects playing roles in collaborations, extended by MAS-ML. 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 Task Specification activity (Section 4.1.2).

Feature/Agent Dependency Modeling. In this activity, we refine the model generated in the Feature/Agent Dependency Modeling activity of the domain analysis stage. New agents and roles can be introduced in the MAS-PL during the domain design stage to model agency features. An example is the *Environment Agent* and the *Facade Agent* in the OLIS MAS-PL. The former receives notifications about the execution of business operations and it propagates them to the other agents of the system, and the last is the access point of the web application to get information from the agents. These agents were not identified in the domain analysis stage, but were introduced only in the domain design stage. So the Feature/Agent Dependency Modeling is refined to incorporate new agents and roles, as also capabilities. The OLIS feature/agent dependency model generated after the domain design stage is depicted in Figure 6(b). Note that some agents appear as a child of more than one feature, meaning that these agents are present in a certain product if at least one of these features is selected for this product.

4.3 Domain Realization

The purpose of the domain realization stage is to implement the reusable software assets, according to the design diagrams generated in the previous stage. 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 stage: Assets Implementation and Design/Implementation Elements Mapping.

Assets Implementation. In this activity, elements designed in the previous stage are coded in some programming language. In (da Silva & de Lucena 2007), it is proposed a code generation from MAS-ML models. Nevertheless, the implementation of software agents are usually accomplished by the use of agent platforms, such as JADE (agents are implemented with Java classes) and Jadex (agents are implemented with XML files and Java classes). Consequently, the design elements can differ from implementation elements, e.g. roles are concepts that are not present in JADE framework, so they can be implemented using classes structured according to the Role Pattern. Therefore, implementation elements should be mapped into design elements, what is done in the next activity.

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. Recent works have explored modularization techniques related to MASs. An architectural pattern is proposed in (Nunes, Kulesza, Nunes, Cirilo & Lucena 2008*b*) to integrate software agents and web applications in a loosely coupled way. In (Nunes, Kulesza, Sant'Anna, Nunes & Lucena 2008), Nunes et al. presented a quantitative study of development and evolution of a MAS-PL, consisting of a systematic comparison between two different versions of a MAS-PL: (i) one version implemented with object-oriented techniques and conditional compilation; and (ii) the other one using aspect-oriented techniques. Furthermore, the are some research work (Garcia, Kulesza & de Lucena 2004) that studies separation of concerns in MAS by means of aspect oriented techniques.

Design/Implementation Elements Mapping. To provide the mapping from the implementation elements to the design elements, we use the models proposed by the GenArch (Cirilo, Kulesza & Lucena 2008) tool. It is a model-based product derivation tool, which encompasses three models: (i) the implementation model; (ii) the feature model; and (iii) the configuration model. This tool was also extended (Cirilo, Kulesza, Nunes, Nunes & Lucena 2008) by the incorporation of a new domain-specific architecture model for Jadex in order to enable the automatic instantiation and customization of MAS-PLs. For further details about GenArch, please refer to (Cirilo, Kulesza & Lucena 2008) and (Cirilo, Kulesza, Nunes, Nunes & Lucena 2008).

5 Related Work

Only few attempts have explored the integration synergy of MASs and SPLs technologies. Pena et al. (Pena, Hinchey & Ruiz-cortés 2006) propose an approach that 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. Their approach is based on MaCMAS, an agent-oriented methodology. One main difference between our approach and theirs is they consider goals as the features of the SPL. Goals are not a detail of the system that is visible to the end user; therefore, they should not appear in a feature model. In addition, the approach first proposes modeling the system, then analyzes the variabilities. This can result, for instance, in an optional feature designed together with a mandatory feature, not keeping the separation of concerns. Finally, the approach does not detail how the SPL assets can be implemented in such way that they can be assembled together to derive a product.

Dehlinger & Lutz (Dehlinger & Lutz 2005) have proposed an extensible agent-oriented requirements specification template for distributed systems that supports safe reuse. Their proposal adopts a product line to promote reuse in MASs, which was developed using the Gaia methodology. The requirements are documented in two schemas: (i) role schema - a role and the variation points that a role can play during its lifetime; and (ii) role variation point - captures the requirements of role variation points capabilities. The proposed approach allows the reuse of agent configuration along the system evolution. Each agent configuration can be dynamically changed and reused in similar applications. Although this approach provides a template to capture agency variability, it does not offer a complete solution to address the modeling of agency features in the domain design and implementation.

6 Conclusion and Future Work

Software product lines can bring many advantages to the development of multi-agent systems, providing benefits such as reduced time-to-marked and lower development costs. In addition, agent abstraction is very useful to model features that present an autonomous or pro-active behavior.

In this paper, we presented a process for domain engineering to develop Multi-agent Systems Product Lines (MAS-PLs), describing the activities to be performed in each one of the stages that compose the process. The definition of our approach incorporates activities and notations of different well-succeeded works in the context of SPLs and MASs: PLUS provides notations for documenting variability; PASSI methodology diagrams are used to specify agency features; and MAS-ML is the modeling language used in the domain analysis stage. We also proposed some adaptations and extensions to these approaches to address agency features. An advantage of the approach resides in the fact that we separate the modeling of agency features, and it makes possible to evolve existing systems, for instance object-oriented, to incorporate new features that take advantage of agent abstraction. We have developed our process with the experience of two cases studies: the OLIS case study (presented in this paper) and the ExpertCommittee case study, which is a product line of conference management systems.

We are currently extending our research work in several directions. First, we are working on the development of other case studies to evaluate our process. We are also investigating how model-driven and aspect-oriented approaches can help to model and implement crosscutting features in order to provide a better modularization. In addition, we are exploring scenarios in which the SPL is built from legacy systems. Finally, we aim at extending our process to address other agent characteristics, such as self-* properties.

References

Alves, V. (2007), Implementing Software Product Line Adoption Strategies, PhD thesis, UFPE, Brazil.

Bauer, B., Müller, J. P. & Odell, J. (2001), Agent uml: a formalism for specifying multiagent software systems, *in* 'AOSE'00'.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. & Mylopoulos, J. (2004), 'Tropos: An agent-oriented software development methodology', *AAMAS* **8**(3), 203–236.

Cirilo, E., Kulesza, U. & Lucena, C. (2008), 'A Product Derivation Tool Based on Model-Driven Techniques and Annotations', *JUCS* **14**, 1344–1367.

Cirilo, E., Kulesza, U., Nunes, I., Nunes, C. & Lucena, C. (2008), Automatic product derivation of multi-agent systems product lines, Technical report, PUC-Rio.

Clements, P. & Northrop, L. (2002), *Software Product Lines: Practices and Patterns*, Addison-Wesley, Boston, MA, USA.

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

da Silva, V. T. & de Lucena, C. J. (2007), 'Modeling multi-agent systems', *Comm. of the ACM* **50**(5), 103–108.

Dehlinger, J. & Lutz, R. R. (2005), A Product-Line Requirements Approach to Safe Reuse in Multi-Agent Systems, *in* 'SELMAS'05'.

Garcia, A. F., Kulesza, U. & de Lucena, C. J. P. (2004), Aspectizing multi-agent systems: From architecture to implementation, *in* 'SELMAS', pp. 121–143.

Gomaa, H. (2004), *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison Wesley Longman Publishing Co., Inc., USA.

Holz, H., Hofmann, K. & Reed, C., eds (2008), *Personalization Techniques and Recommender Systems*, Vol. 70, World Scientific Publishing.

Kästner, C., Apel, S. & Kuhlemann, M. (2008), Granularity in software product lines, *in* 'ICSE '08', ACM, USA, pp. 311–320.

Krueger, C. W. (2002), Easing the transition to software mass customization, *in* 'PFE'01', Springer-Verlag, London, UK, pp. 282–293.

Nunes, C., Kulesza, U., Sant'Anna, C., Nunes, I. & Lucena, C. (2008), On the modularity assessment of aspect-oriented multi-agent systems product lines: a quantitative study, *in* 'SBCARS '08', Porto Alegre, Brazil, pp. 122–135.

Nunes, I., Kulesza, U., Nunes, C., Cirilo, E. & Lucena, C. (2008*a*), Extending passi to model multi-agent systems product lines, Technical report, PUC-Rio.

Nunes, I., Kulesza, U., Nunes, C., Cirilo, E. & Lucena, C. (2008*b*), Extending web-based applications to incorporate autonomous behavior (to appear), *in* 'WebMedia 2008', Vila Velha, Brazil.

Nunes, I., Nunes, C., Kulesza, U. & Lucena, C. (2008), Documenting and modeling multi-agent systems product lines, *in* 'SEKE '08', Redwood City, San Francisco Bay, USA, pp. 745–751.

Padgham, L. & Lambrix, P. (2000), Agent capabilities: Extending bdi theory, *in* 'AAAI '00', AAAI Press / The MIT Press, pp. 68–73.

Pena, J., Hinchey, M. G. & Ruiz-Cortés, A. (2006), 'Multi-agent system product lines: challenges and benefits', *Communications of the ACM* **49**(12), 82–84.

Pena, J., Hinchey, M. G. & Ruiz-cortés, A. (2006), Building the core architecture of a nasa multiagent system product line, *in* 'AOSE'06'.

Pohl, K., Böckle, G. & van der Linden, F. J. (2005), *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag, USA.

Wooldridge, M., Jennings, N. R. & Kinny, D. (2000), 'The gaia methodology for agent-oriented analysis and design', *AAMAS* **3**(3), 285–312.