On the Development of Multi-agent Systems Product Lines: A Domain Engineering Process

Ingrid Nunes[1], Carlos J.P. de Lucena[1], Uirá Kulesza[2], and Camila Nunes[1]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ioliveira,lucena,cnunes}@inf.puc-rio.br
[2] Federal University of Rio Grande do Norte (UFRN) - Natal, Brazil
uira@dimap.ufrn.br

Abstract. Multi-agent System Product Lines (MAS-PLs) are the integration of two promising technologies: Multi-agent Systems (MASs), which provides a powerful abstraction to model features with autonomous and pro-active behavior, and Software Product Lines (SPLs), whose aim is to reduce both time-to-market and costs in the development of system families by the exploitation of commonalities among family members. This paper presents a domain engineering process for developing MAS-PLs. It defines activities and work products, whose purposes include allowing agent variability and providing agent features traceability, both not addressed by current SPL and MAS approaches.

Key words: Multi-agent Systems, Software Product Lines, Domain Engineering, Software Process

1 Introduction

Complex modern software systems tend to be situated, open, autonomous and highly interactive [1]. Agent-oriented Software Engineering (AOSE) has emerged as new paradigm that addresses the development of complex and distributed systems based on their decomposition into autonomous and pro-active agents, which together compose a Multi-agent System (MAS). However, MAS methodologies have not addressed so far the need of developing large scale customized systems and little effort has been done to take advantage of software reuse techniques. Software Product Lines (SPLs) manage to promote reduced time-to-market, lower development costs and higher quality to the development of applications that share common and variable features. Based on the exploitation of application commonalities and large-scale reuse, these applications are derived in a systematic way and are customized to specific user needs. In order to fulfil the increasing demand of large-scale and customized MASs, Multi-agent System Product Lines (MAS-PLs) have emerged to integrate these two promising trends of software engineering. The main goal of MAS-PLs is to incorporate their respective benefits and to help the industrial exploitation of agent technology.

In this context, this paper presents a domain engineering process for developing MAS-PLs. The two main issues in the MAS-PL development that we

aim at addressing and have not been addressed by current approaches are: (i) *documenting agent variability* – explicit variability documentation is essential in SPLs [2]. Nevertheless, SPL approaches do not cover variability documentation in agent models; they focus on specific models, e.g. object-oriented and component-oriented; and (ii) *tracing agent features* – feature traceability allows to specify the configuration knowledge between problem and solution space thus enabling the selection of appropriate artifacts of a SPL in the product derivation process. In defining our process we introduce new and modified models as well as leverage some activities and notations that consist of parts of methods of existing SPL and MAS approaches [3–5].

Besides providing customized applications derived in a systematic way, the scenario we are currently exploring is the incorporation of autonomous and proactive behavior to existing web systems. This is becoming common practice for several web-based systems, for instance recommending products in online stores and displaying personalized ads in search engines according to previous searches. As a consequence, we aim at explicitly separating agent and non-agent features, mainly due to two reasons: (i) agent abstraction provides some particular characteristics, such as autonomy and pro-activeness. Features that do not require them can be modeled and implemented using other technologies (e.g. object-oriented) taking benefit from frameworks and approaches already proposed; and (ii) we aim at supporting the evolution of existing applications and SPLs that have been developed using other existing technologies by the incorporation of agent features. Given that we are adopting a feature-oriented development approach, features are modeled independently of each other. Therefore, it is possible to modeling agent and non-agent features in different ways.

The paper is structured as follows. Related work is presented in Section 2. Section 3 presents the proposed domain engineering process, first giving an overview of it, and later detailing each of its phases. Section 4 concludes this paper and points out directions for future work.

2 Existing MAS-PL Approaches

Several approaches have been published to address problems and challenges of both SPL and MAS engineering [3–6]. Even though many MAS methodologies have been proposed, 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. They do not consider variability on agent models and do not take into account feature modularization and traceability. Despite the fact that SPL approaches provide useful notations to model agent features, none of them completely covers all their properties and concepts [7]. They do not provide models to design agent concepts and map them to features.

Only few attempts have explored the integration synergy of MASs and SPLs. Pena et al. [8] propose an approach based on MaCMAS methodology, which consists of using goal-oriented requirement documents, role models, and traceability diagrams in order to build a first model of the system. A principle of SPLs is

to design and implement features as modularized as possible in order to allow an effective application engineering. However, their approach proposes that variabilities are analyzed after modeling the MAS, and this can lead to undesired situations, such as, the high coupling between mandatory and optional features and inadequate modularization of agent features.

Dehlinger & Lutz [9] have proposed an extensible agent-oriented requirements specification template for distributed systems that supports safe reuse. Their proposal adopts a SPL approach to promote reuse in MASs, which was developed using the Gaia methodology. Although this approach provides a template to capture agent variability, it covers only the requirements engineering phase, and therefore it does not offer a complete solution to address the modeling of agent features in the domain design and implementation.

3 A Domain Engineering Process for MAS-PL

Domain engineering is the process of SPL engineering in which SPL commonalities and variabilities are identified, defined and realized. This section first introduces our process and describes its key characteristics (Section 3.1). Later, it presents and analyzes approaches that led to some activities and notations incorporated to our process. (Section 3.2). Finally, it details each phase of our domain engineering process (Sections 3.3, 3.4 and 3.5).

We illustrate our process phases with our ExpertCommittee (EC) case study [10]. It is a MAS-PL of conference management systems, whose aim is to manage paper submission and reviewing processes from conferences. The multi-agent version of this kind of system was first proposed in [11] and since has been widely used to the elaboration and application of MAS methodologies. We assume the readers of this paper are mostly knowledgeable about the domain, but a complete description about this case study can be seen in [10]. In addition to the traditional functionalities provided by conference management systems, we have incorporated features to the EC, whose goal is automating tasks previously executed by users and each of them can be or not included in a specific conference management application. Examples are the automatic suggestion of conferences to authors and automatic assignment of papers to committee members.

Due to space restrictions, we focus on describing activities purposes and their main output work products, suppressing some details such as tasks of each activity and roles. Additional details and artifacts produced for the EC case study can be found in [12].

3.1 Process Overview

Our process is structured according to the SPEM [13], which provides a common syntax and modeling structure to construct software process models. It is based on three main levels: (i) phases – significant periods in a process; (ii) activities – general units of work; and (iii) tasks – define work being performed by roles and are associated with input and output work products. Figure 1 summarizes our

process. Following typical domain engineering processes, our approach encompasses three phases: Domain Analysis, Domain Design and Domain Realization. The qualifier "domain" emphasizes the multisystem scope of these phases.

Our process aggregates activities and notations that are specific to model agents and their variabilities to address MAS-PLs. Notations and guidelines that have been adopted alongside all our process are: (i) use of ≪*kernel*≫, ≪*optional*≫ and ≪*alternative*≫ stereotypes to indicate variability in different model elements (e.g. use cases, classes and agents) of several models; (ii) separated modeling of features, to stress the fact that diagrams are split accordingly; (iii) specific models to provide features traceability along all the process. Most activities have a specific task for generating the traceability model; 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.

3.2 Method Fragments Incorporated to our Process

Even though SPL and MAS approaches present deficiencies to develop MAS-PLs, they provide useful notations and activities that can be integrated to model MAS-PLs. Consequently, instead of proposing an approach from scratch, we have incorporated fragments of existing approaches into 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. Basically the reasons for adopting PLUS are [7]: (i) PLUS explicitly models the commonality and variability in a SPL, mainly through the use of UML stereotypes; (ii) it uses feature modeling to address variability in the domain analysis, as it is commonly done in SPL approaches. On the other hand, other SPL approaches have the following drawbacks: they either focus mainly on management aspects of SPLs [6], also lack design details, or just provide high level guidelines [14].

In order to model agent features at the Domain Analysis phase, we have adopted some phases of PASSI [4], an agent-oriented methodology. It specifies models with their respective phases for developing MASs, covering all the development process. PASSI integrates concepts from object-oriented software engineering and artificial intelligence, and it follows the guideline of using standards whenever possible. This justifies the use of UML as modeling language. One of the key reasons for choosing PASSI is that the use of an UML-based notation enables the merging of complementary notations proposed in PLUS and PASSI, while establishing a standard for modeling agent and non-agent features.

Instead of using UML for modeling agents in the Domain Design phase, as PASSI proposes, we use an extended version of it, the MAS-ML modeling language [5, 15]. Our focus is to allow the design of agents that follow the belief-desire-intention (BDI) [16] model, whose 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, Jason, JACK and 3APL; and it is based

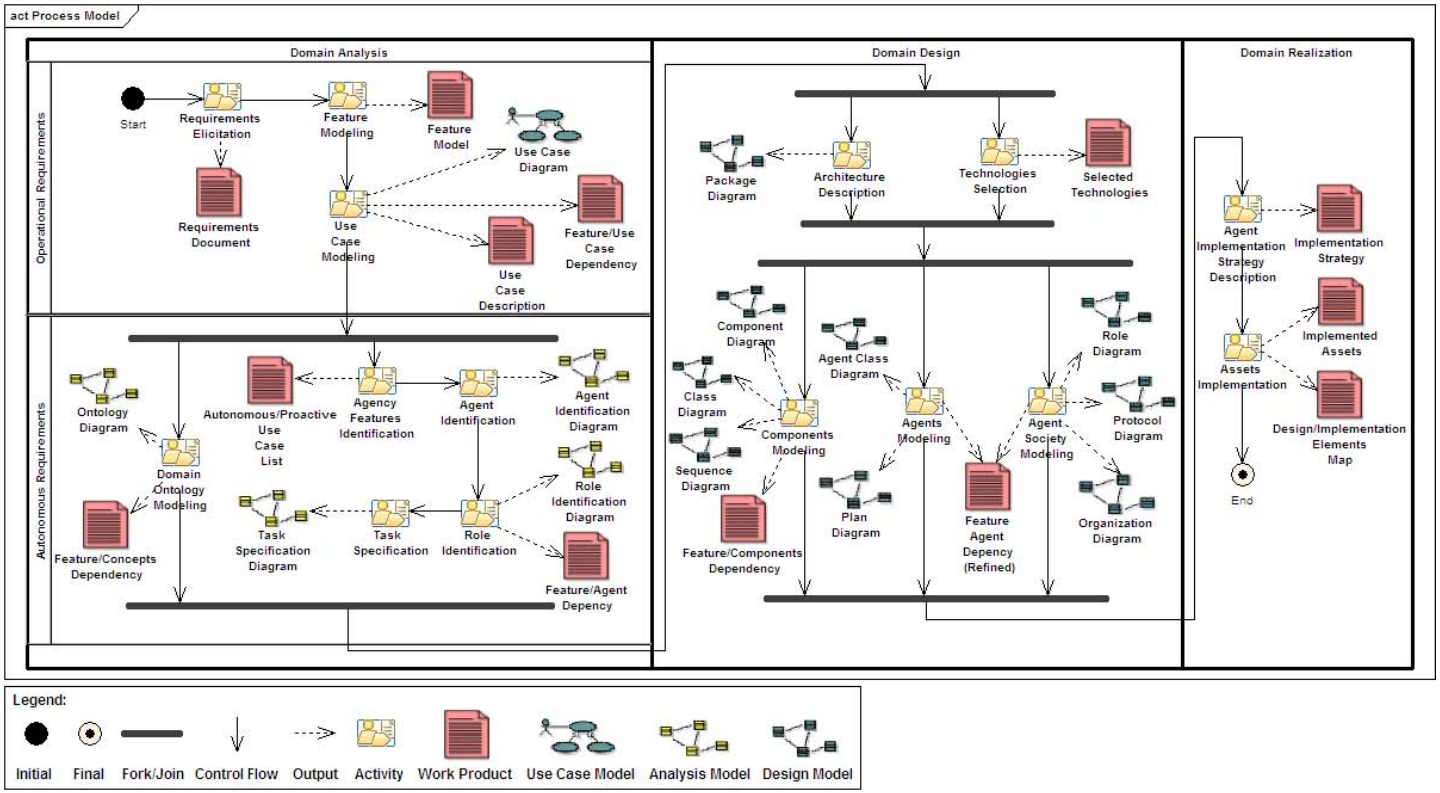


Fig. 1. The Domain Engineering Process.

on solid philosophical foundations. As discussed in [15], 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. As a consequence, other MAS modeling languages do not allow the modeling of some agent concepts. 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.

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 Domain Analysis

The Domain Analysis phase defines activities for eliciting and documenting the common and variable requirements of a SPL. This phase comprises two sub-phases: Operational Requirements and Autonomous Requirements.

In the Operational Requirements sub-phase, the systems family is analyzed and its common and variable features are identified therefore defining the SPL scope. 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. In sequel, requirements are described in terms of use case diagrams and descriptions. The first activity is the Requirements Elicitation, which captures requirements in documents based on interactions with domain specialists and stakeholders. The next activity is the Feature Modeling, which was originally proposed by the FODA [14] method and is the activity of modeling the common and variable properties of concepts and their interdependencies in SPLs. Features are organized into a tree representation, called features diagram, with a specific notation for each variability category (mandatory, alternative and optional). EC features diagram is depicted in Figure 2(a), showing the optional automatic conference suggestion feature. A feature model refers to a features diagram accompanied by additional information such as dependencies among features. It represents the variability within a system family in an abstract and explicit way.

After the Feature Modeling activity, SPL functional features are described in terms of use cases, in the Use Case Modeling activity. First, use cases are identified and described, resulting in both a use case diagram and use case descriptions. Later, the use case diagram should be refined by: (i) refactoring use cases to provide feature modularization (each use case should correspond to only one feature). It is accomplished by the use of the generalization and the extend relationships; and (ii) adding stereotypes to give variability information (≪*kernel*≫,≪*optional*≫ and ≪*alternative*≫). For instance, the *Suggest*

Conferences is an optional use case and extends the *Register Paper* use case, which is part of the SPL kernel. To complete the Operational Requirements, another use case view is modeled to map use cases to features, resulting in the Feature/Use Case Dependency model. Use cases are grouped into features with the UML package notation. 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*≫ – aggregates alternative features and groups use cases related to a specific alternative feature. Figure 2(b) shows the EC Feature/Use Case Dependency model with two optional features: *Automatic Distribution* and *Conference Suggestion*.

The purpose of the Autonomous Requirements sub-phase is to understand better 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 well described in use cases, and consequently they need a more precise specification. Agents are an abstraction of the problem space that are a natural metaphor to model pro-active or autonomous behavior. Therefore, it is identified and specified in models in terms of agents and roles.

The domain concepts are captured through the Domain Ontology Modeling activity, in which the MAS-PL domain is modeled through an ontology. The concepts should be modeled taking into account features, by using techniques such as generalization to modularize features. The ontology is represented by UML class diagrams, on which classes and their attributes represent concepts and slots, respectively. 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. In parallel to this activity, features that present pro-active or autonomous behavior (agent features) are identified and specified in models in terms of agents and roles. The Agent Features Identification activity is responsible for such identification. So, a new stereotype (≪*agent feature*≫) is added to the packages of the Feature/Use Case Dependency model to indicate which features are agent features. In Figure 2(b), both optional features are classified as agent features. In order to specify the identified agent features, our process incorporates some activities that correspond to some phases of the System Requirements model of PASSI methodology [17]. Next, we briefly describe PASSI phases used in our process and our proposed extensions to them.

- Agent Identification: use cases are attributed to agents (represented as stereotyped UML packages). Use case stereotypes used in the Use Case diagram are still present. *Our extensions:* only pro-active or autonomous use cases are distributed among agents, while PASSI proposes the realization of all use cases performed by agents; adoption of stereotypes (kernel, alternative or optional); and use of colors to trace features.

- Role Identification: agent interactions are explored and expressed through sequence diagrams to identify agent roles. *Our extensions:* diagrams split accord-

ing to features; use of UML 2.0 frames for representing crosscutting features; use of colors to trace features; and Feature/Agents Dependency model.

- Task Specification: activity diagrams are used to specify the capabilities of each agent. *Our extensions:* one diagram per agent and feature; use of UML 2.0 structured activities for representing crosscutting features (Figure 2(c)) and use of colors to trace features.

Most of these adaptations are meant to provide variability information in models. Some proposed notations aims at modularizing crosscutting features, which are characterized by having impact in several other features. So, instead of creating new models for them, notations indicate the behavior introduced by this kind of feature. An additional model (Feature/Agents Dependency model) provides support to trace from features to agents (and vice-versa). This model is organized into a tree, whose root represents 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 plays that roles for a certain feature. Agents and roles may appear more than once in the model, meaning that they will be present in a product if at least one agent feature that depends on them is selected.

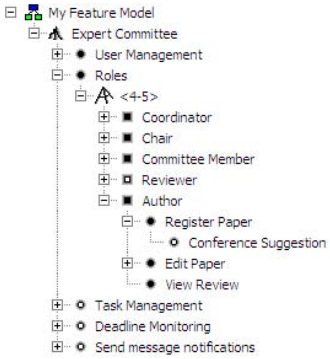
3.4 Domain Design

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 analysis models, designers must model the SPL architecture, determining how these models, including the variability, are implemented in this architecture. Features modularization must be taken into account during the design of 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 traceability information.

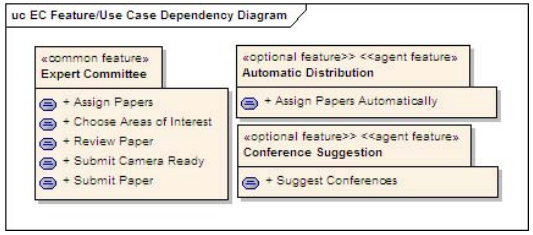
The Domain Design phase has mainly two parts. First, the SPL architecture is defined and technologies (e.g. frameworks, libraries and agent platforms) that will be used are selected. The SPL architecture is specified in an UML package diagram and defined by its decomposition into subsystems and their components. This helps to reduce the complexity and to allow several design teams to work independently. Choosing appropriate technologies for designing and implementing a SPL is a very important step of its development, because they have an impact on how features are modularized.

On the second part, each feature is statically and dynamically designed in three different activities: Components Modeling, Agents Modeling and Agent Society Modeling. The first activity concerns the design of non-agent features. This design step is basically provided by the PLUS approach.

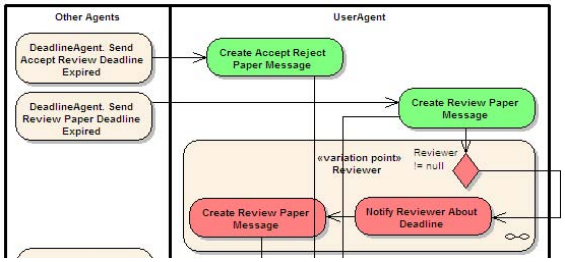
Agent features are modeled in two activities of our process. They are performed in parallel and may contribute with each other. In the Agents Modeling activity, agents with their beliefs, goals and plans are modeled; and in the Agent Society Modeling activity, roles and organizations are modeled. As mentioned in Section 3.2, we use MAS-ML to model agents. Its structural diagrams are



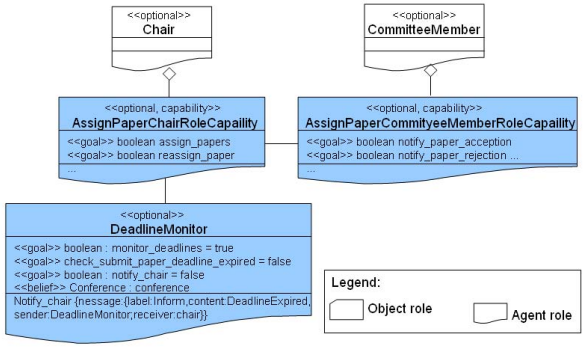
(a) Features Diagram



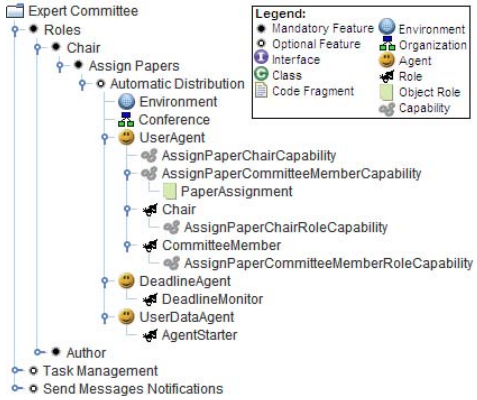
(b) Feature/Use Case Dependency model



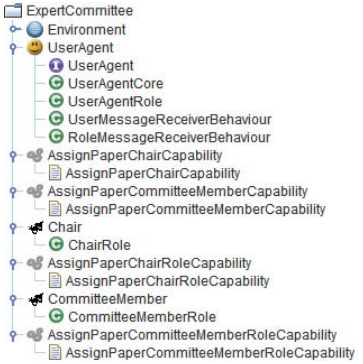
(c) Task Specification Diagram



(d) Role Diagram



(e) Feature/Agent Dependency



(f) Design/Implementation Elements Map

Fig. 2. EC Work Products (Partial).

the extended UML class diagram and two new diagrams: organization and role. MAS-ML extends the UML class diagram to represent the structural relationships among agent concepts. The organization diagram models system organizations and relationships between them and other system elements. Finally, the role diagram is responsible for modeling relationships between roles defined in organizations.

To address variability in MAS-ML diagrams, we have adopted four different adaptations: (i) use of ≪*kernel*≫ , ≪*optional*≫ and ≪*alternative*≫ stereotypes to indicate variability; (ii) model elements are colored according to the feature they are related to, based on the color assigned for each feature; (iii) model each feature in a different diagram. However, crosscutting features impact in several features, so their specification is spread in other features' diagrams. Although crosscutting features are not modeled in specific diagrams, the use of colors helps to distinguish different model elements related to them; and (iv) the introduction of the capability [18] concept to modularize variable parts in agents and roles. A capability is essentially a set of plans, a fragment of the knowledge base and a specification of the interface to the capability. Capabilities have been introduced into some MASs as a software engineering mechanism to support modularity and reusability. We represent a capability in MAS-ML by the agent or agent role notation with the ≪*capability*≫ stereotype. An aggregation relationship is used between capabilities and agents, and capabilities and roles. Figure 2(d) shows the *Assign Papers* role diagram, on which the colored elements are capabilities that defines parts of the `Chair` and `CommitteeMember` roles and a role (`DeadlineMonitor`) that are specific for this feature.

Agents' dynamic behavior is modeled by means of extended sequence diagrams by MAS-ML. The extended version of this diagram represents the interaction between agents, organizations and environments. The only differences in modeling dynamic behavior for single systems and MAS-PLs are: (i) different features are modeled in different 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 3.3).

Besides modeling agents with appropriate feature modularization and variability notations, the Feature/Agent Dependency model is refined by introducing new agent concepts that were identified in both activities. Figure 2(e) shows a partial view of the EC Feature/Agent Dependency model.

3.5 Domain Realization

The purpose of the Domain Realization phase is to implement the reusable software assets, according to the design diagrams. In addition, it incorporates configuration mechanisms that enable the product instantiation process. Two activities compose this phase: Agent Implementation Strategy Description and Assets Implementation.

Implementing software agents is usually accomplished by the use of agent platforms, such as JADE (the two main provided concepts are agent and behaviors) and Jadex (implements the BDI architecture, providing goal, belief and

plan concepts). Consequently, there are different ways of implementing agents. In addition, the concepts adopted to model them may be different of the ones provided by the target implementation platform. So, the goal of the Agent Implementation Strategy Description activity is to define a strategy for implementing agents. For instance, mapping agent concepts (agents, beliefs and plans) to object-oriented concepts (classes, attributes and methods).

In the Assets Implementation activity, designed elements are codified in some programming language. So, the first task of this activity is to implement SPL assets. Different implementation techniques can be used to modularize features in the code, e.g. polymorphism, design patterns, frameworks, conditional compilation and aspect-oriented programming. In [19], we presented a quantitative study of development and evolution of the EC MAS-PL, consisting of a systematic comparison between two different versions: (i) one version implemented with object-oriented techniques and conditional compilation; and (ii) the other one using aspect-oriented techniques. Additionally, in [20], we have proposed an architectural pattern to integrate software agents and web applications in a loosely coupled way.

As stated previously, the target implementation platform may force transforming some design elements into platform specific ones. Hence, it is necessary to specify how the design elements were implemented in the selected agent platform for traceability purposes. The Design/Implementation Elements Map is responsible for providing this information. It defines which elements implement which design elements. Figure 2(f) presents a partial view of this model for the EC, showing which classes implemented the User agent, for instance.

4 Conclusions and Future Work

In this paper, we have proposed a domain engineering process for developing MAS-PLs. Using a SPL approach for building MASs allows meeting the need of producing software with mass customization while taking advantage of agent abstraction to model modern software systems that tend to be situated, open, autonomous and highly interactive. Our process was modeled according to SPEM and includes specific activities and work products to address agent features and their traceability, and also provide notations for documenting agent variability. It was also defined based on existing best practices of existing SPL and MAS approaches: PLUS provides notations for documenting variability; PASSI methodology diagrams are used to specify agent features in the Domain Analysis phase; and MAS-ML is the modeling language used in the Domain Design phase. A new contribution of our approach resides in the fact that we completely separate the modeling of agent features. Therefore, it makes it possible to evolve existing systems developed with different technologies to incorporate new features that take advantage of agent abstractions. Our process has emerged based on the experience of development of two web-based MAS-PLs: the ExpertCommittee (presented in this paper) and the OLIS case study, which is a SPL of web applications that provide personal services to users. In addition, the process was

used in a graduate Agent-oriented Software Engineering course at PUC-Rio in order to provide further evaluation of the approach.

We are currently developing other case studies to evaluate our process. In addition, we are investigating how model-driven and aspect-oriented approaches can help to model and implement crosscutting features to provide for a better modularization. Finally, we aim at extending our process to address other agent characteristics, such as self-* properties.

References

1. Zambonelli, F., Omicini, A.: Challenges and research directions in agent-oriented software engineering. JAAMAS **9**(3) (2004)
2. Muthig, D., Atkinson, C.: Model-driven product line architectures. In: SPLC 2. (2002) 110–129
3. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley (2004)
4. Cossentino, M.: IV. In: From Requirements to Code with the PASSI Methodology. Idea Group Inc. (2005)
5. da Silva, V.T., de Lucena, C.J.P.: From a conceptual framework for agents and objects to a multi-agent system modeling language. JAAMAS **9**(1-2) (2004)
6. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
7. Nunes, I., Nunes, C., Kulesza, U., Lucena, C.: Documenting and modeling multi-agent systems product lines. In: SEKE 2008. (2008) 745–751
8. Pena, J., Hinchey, M.G., Ruiz-Corts, A., Trinidad, P.: Building the core architecture of a multiagent system product line: with an example from a future nasa mission. In: AOSE '06. LNCS. (2006)
9. Dehlinger, J., Lutz, R.R.: A Product-Line Requirements Approach to Safe Reuse in Multi-Agent Systems. In: SELMAS '05, ACM Press (2005)
10. Nunes, I. et al.: Developing and evolving a multi-agent system product line: An exploratory study (to appear). In: AOSE 2008. LNCS. Springer-Verlag (2009)
11. Ciancarini, P. et al.: A case study in coordination: Conference management on the internet (1998) http:www.cs.unibo.it/cianca/wwwpages/case.ps.gz.
12. Nunes, I.: Towards a multi-agent product line development methodology (2008) http://www.inf.puc-rio.br/~ioliveira/maspl/.
13. Object Management Group (OMG): Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0 (2008)
14. Kang, K. et al.: Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie-Mellon University (1990)
15. da Silva, V.T., Choren, R., de Lucena, C.J.P.: MAS-ML: a multiagent system modelling language. IJAOSE **2**(4) (2008) 382–421
16. Rao, A., Georgeff, M.: BDI-agents: from theory to practice. In: ICMAS-95. (1995)
17. Nunes, I., Kulesza, U., Nunes, C., Cirilo, E., Lucena, C.: A domain analysis approach for multi-agent systems product lines (to appear). In: ICEIS 2009. (2009)
18. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring bdi agents in functional clusters. In: ATAL '99. (2000) 277–289
19. Nunes, C. et al.: On the modularity assessment of aspect-oriented multi-agent systems product lines: a quantitative study. In: SBCARS 2008. (2008) 122–135
20. Nunes, I., Kulesza, U., Nunes, C., Cirilo, E., Lucena, C.: Extending web-based applications to incorporate autonomous behavior. In: WebMedia 2008. (2008)