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.

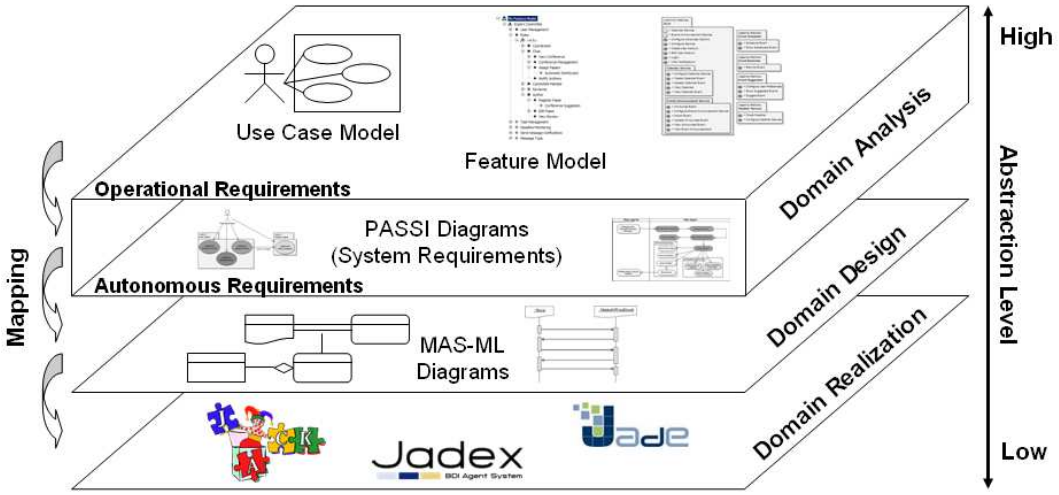


Figure 3.1: Domain Engineering Process Overview.

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.

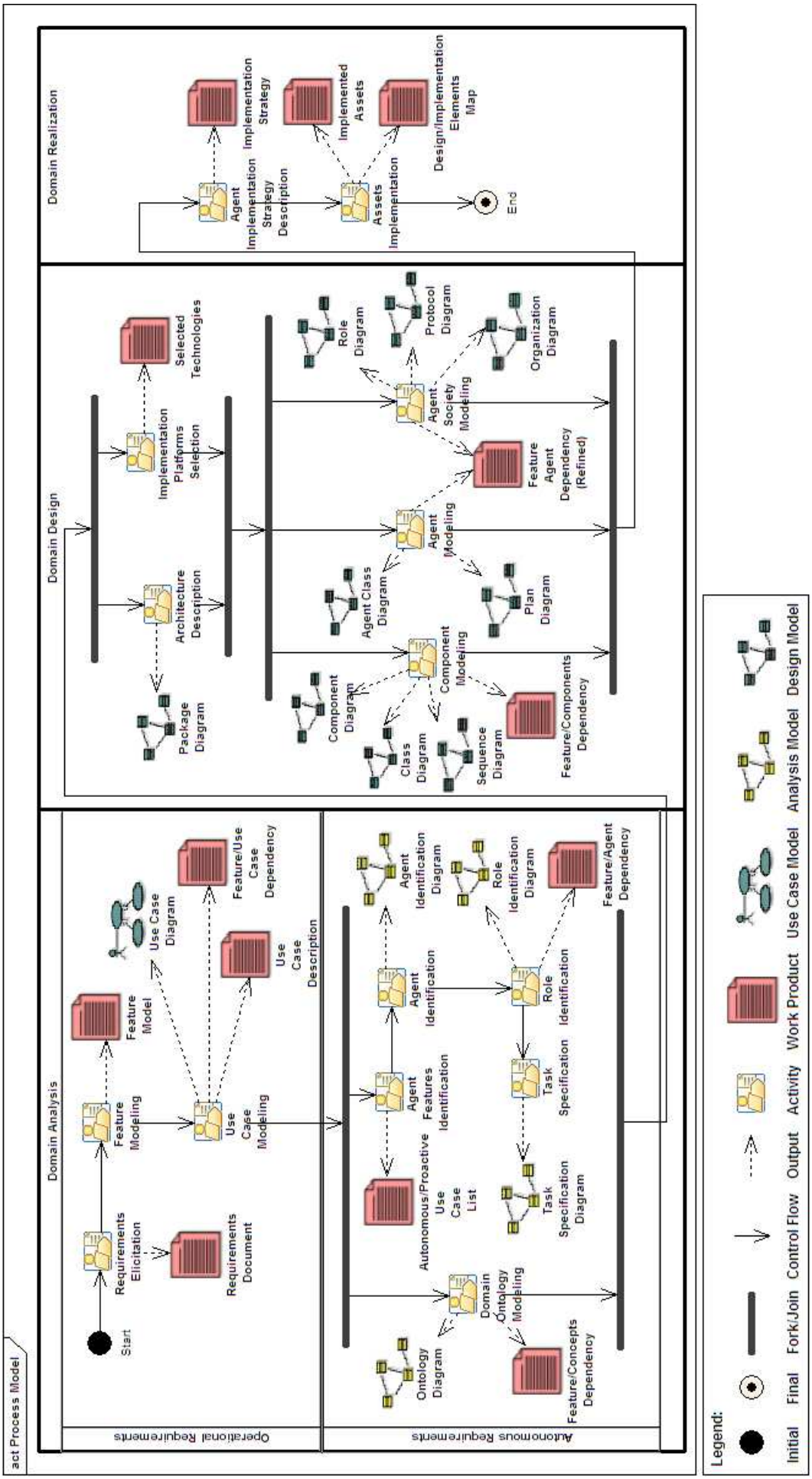


Figure 3.2: The Domain Engineering Process.

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.

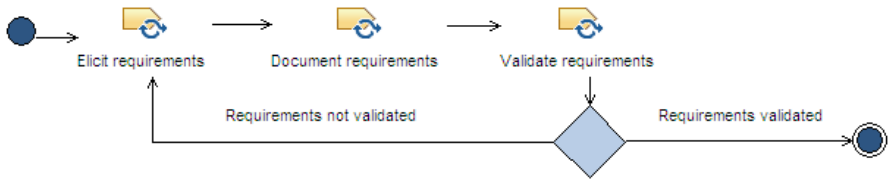


Figure 3.3: Requirements Elicitation activity diagram.

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.

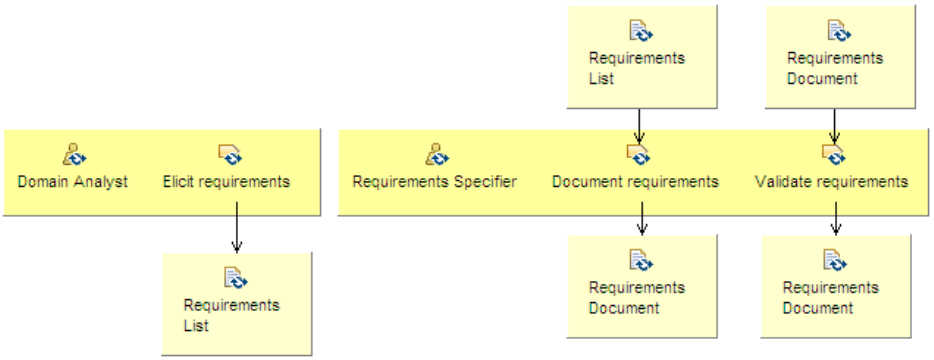


Figure 3.4: Requirements Elicitation detailed activity diagram.

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.

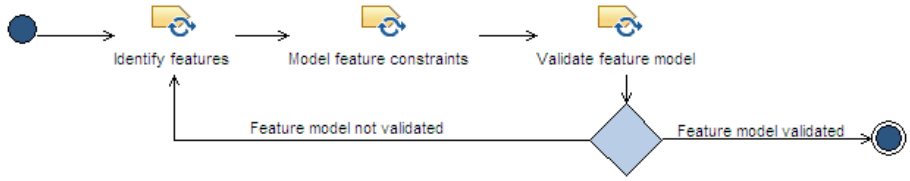


Figure 3.5: Feature Modeling activity diagram.

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-

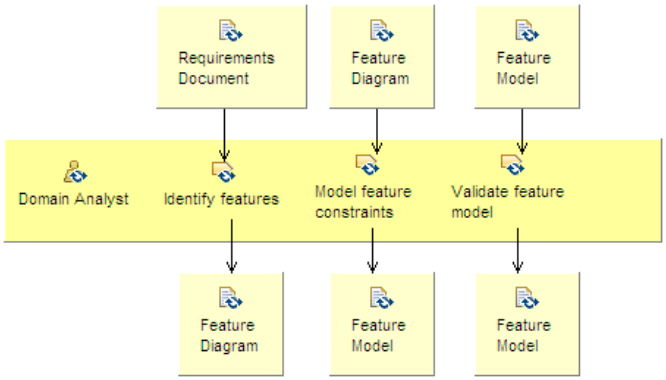


Figure 3.6: Feature Modeling detailed activity diagram.

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.

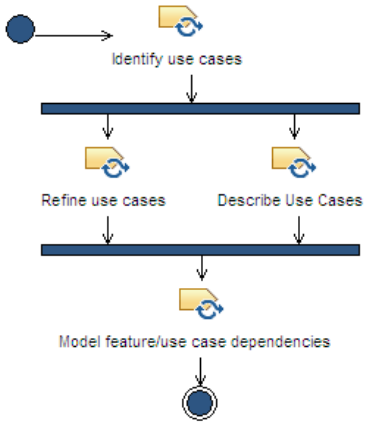


Figure 3.7: Use Case Modeling activity diagram.

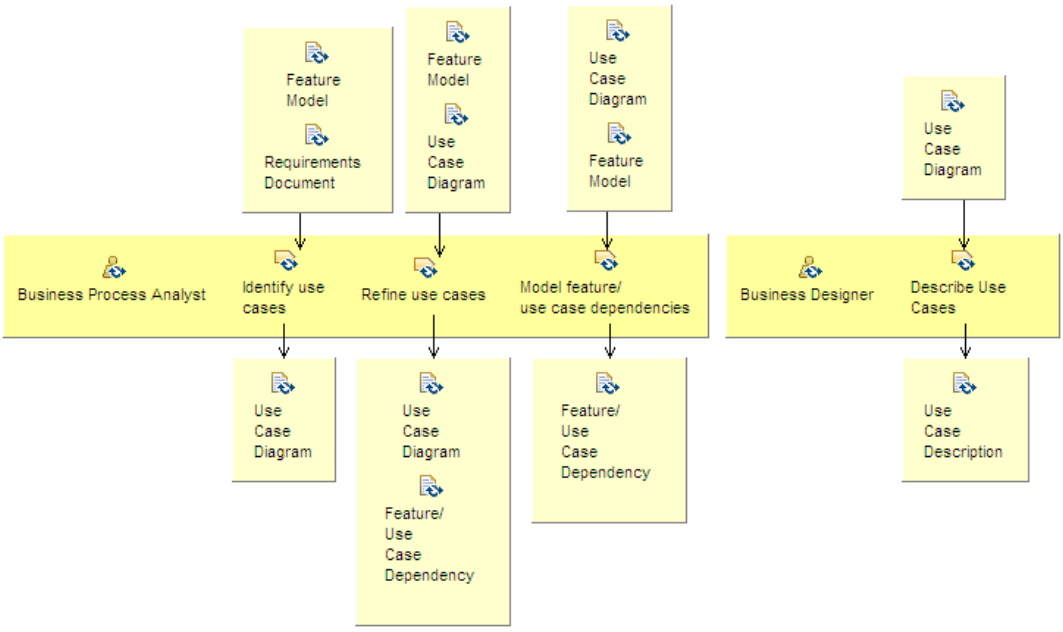


Figure 3.8: Use Case Modeling detailed activity diagram.

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.

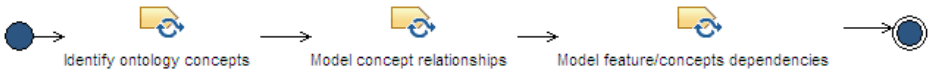


Figure 3.9: Domain Ontology Modeling activity diagram.

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

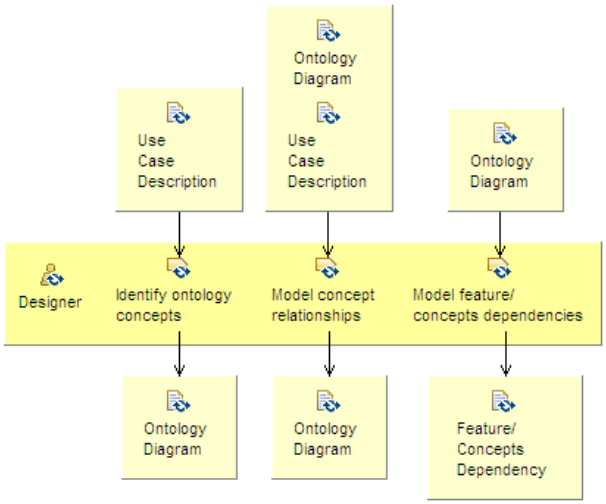


Figure 3.10: Domain Ontology Modeling detailed activity diagram.

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.

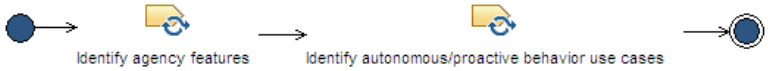


Figure 3.11: Agent Features Identification activity diagram.

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

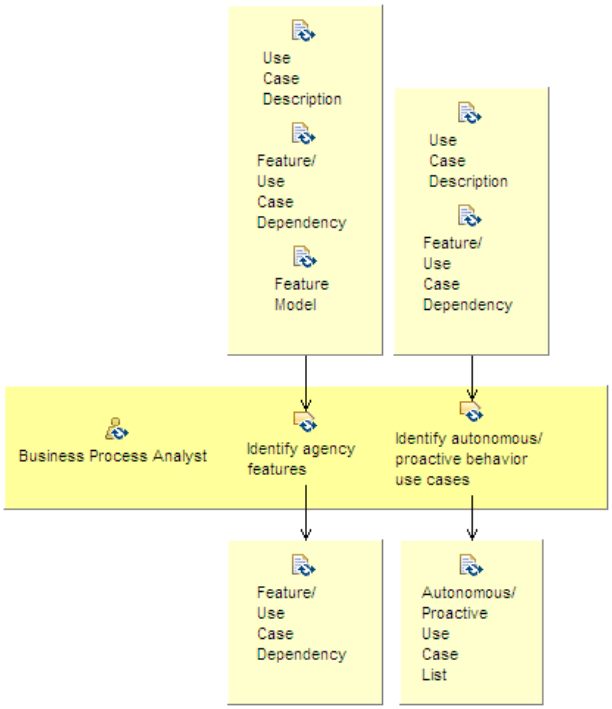


Figure 3.12: Agent Features Identification detailed activity diagram.

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.

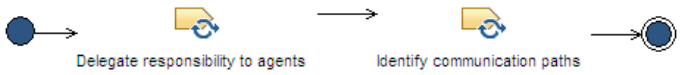


Figure 3.13: Agent Identification activity diagram.

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

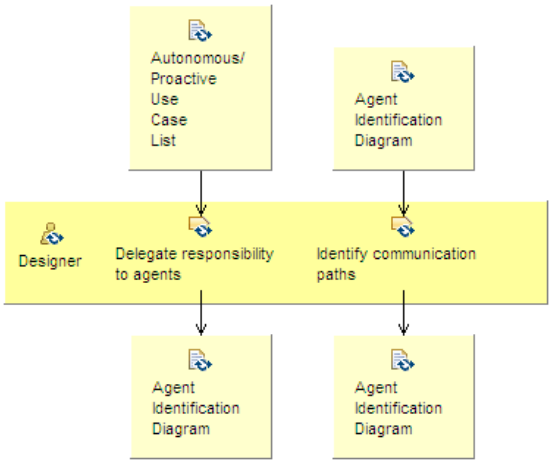


Figure 3.14: Agent Identification detailed activity diagram.

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.

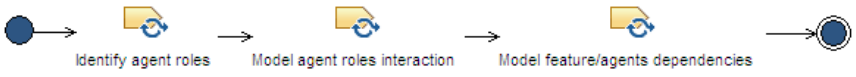


Figure 3.15: Role Identification activity diagram.

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

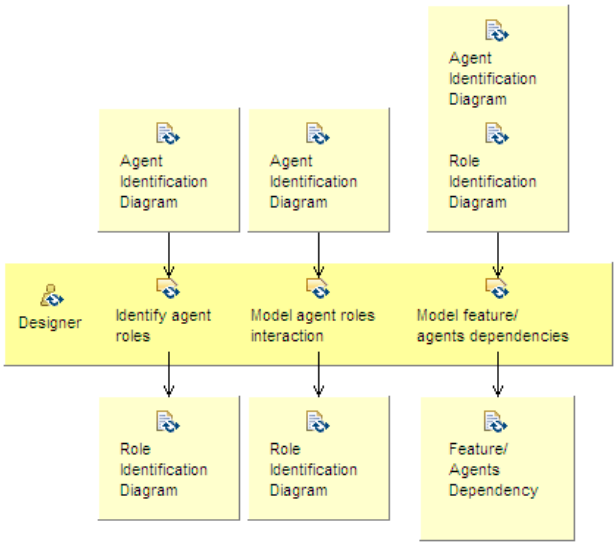


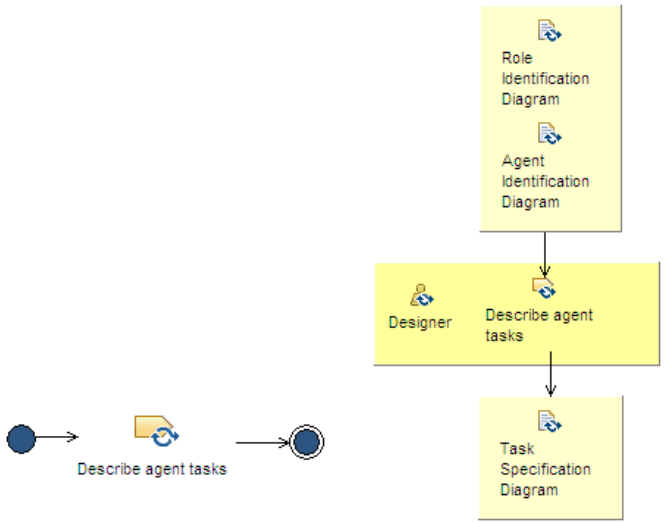
Figure 3.16: Role Identification detailed activity diagram.

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.

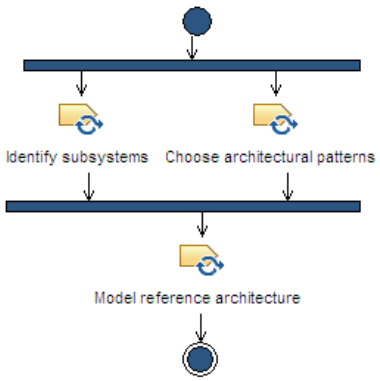


Figure 3.18: Architecture Description activity diagram.

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.

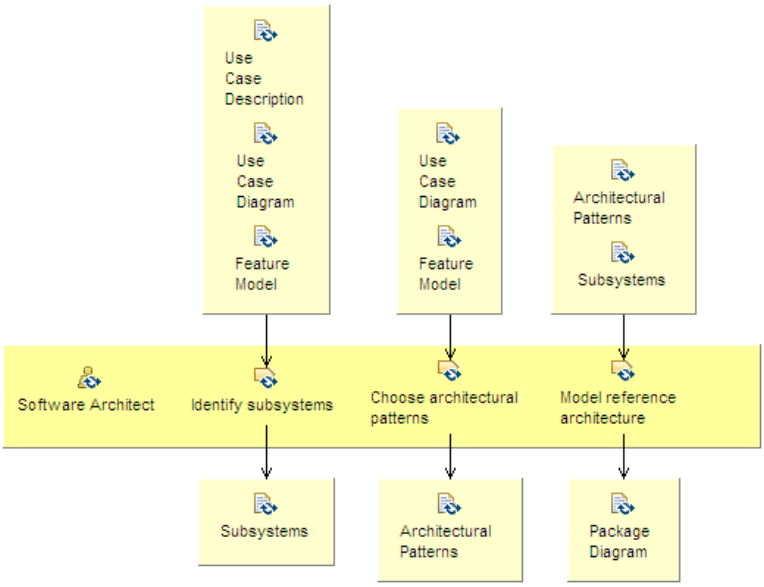


Figure 3.19: Architecture Description detailed activity diagram.

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).

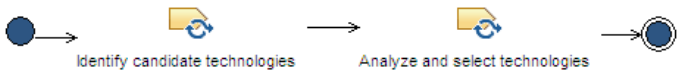


Figure 3.20: Implementation Platforms Selection activity diagram.

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.

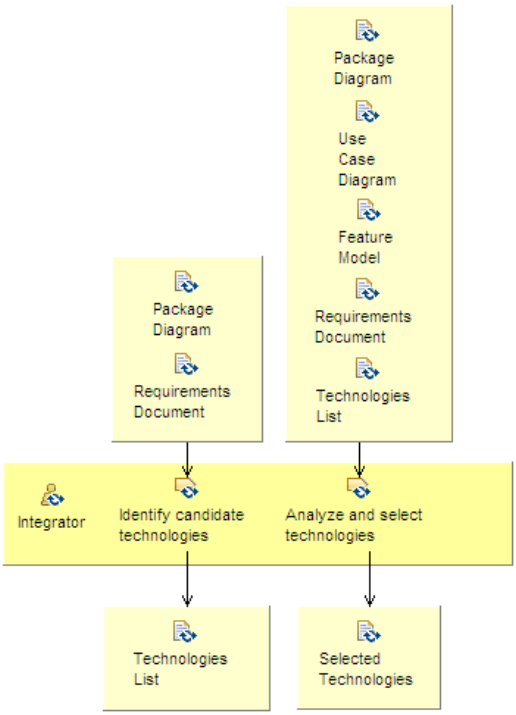


Figure 3.21: Implementation Platforms Selection detailed activity diagram.

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

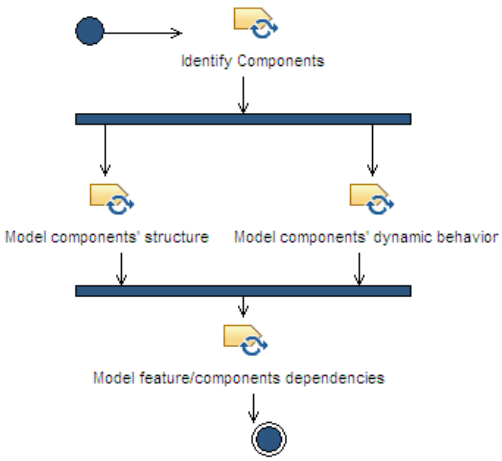


Figure 3.22: Component Modeling activity diagram.

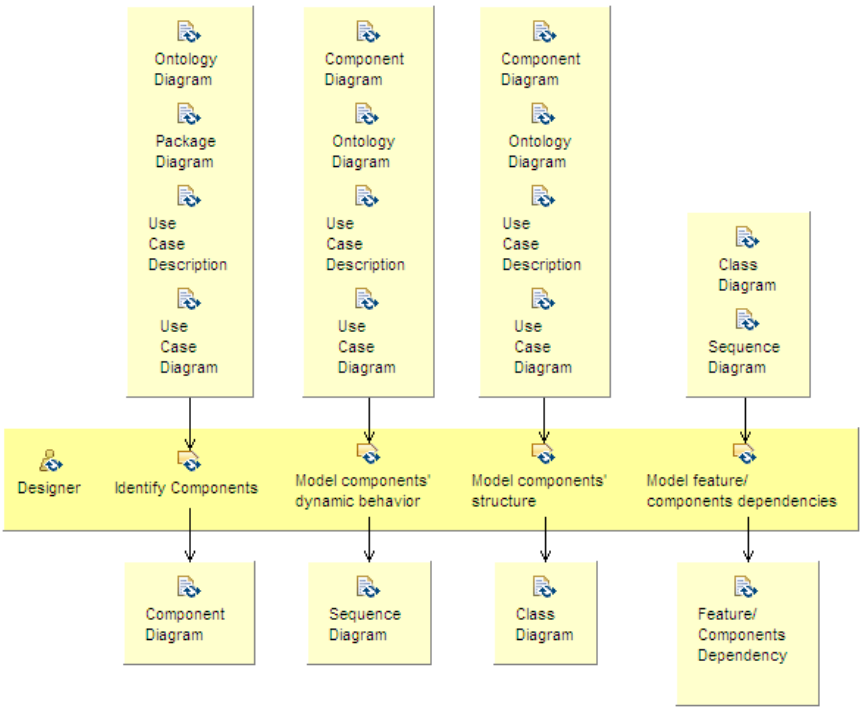


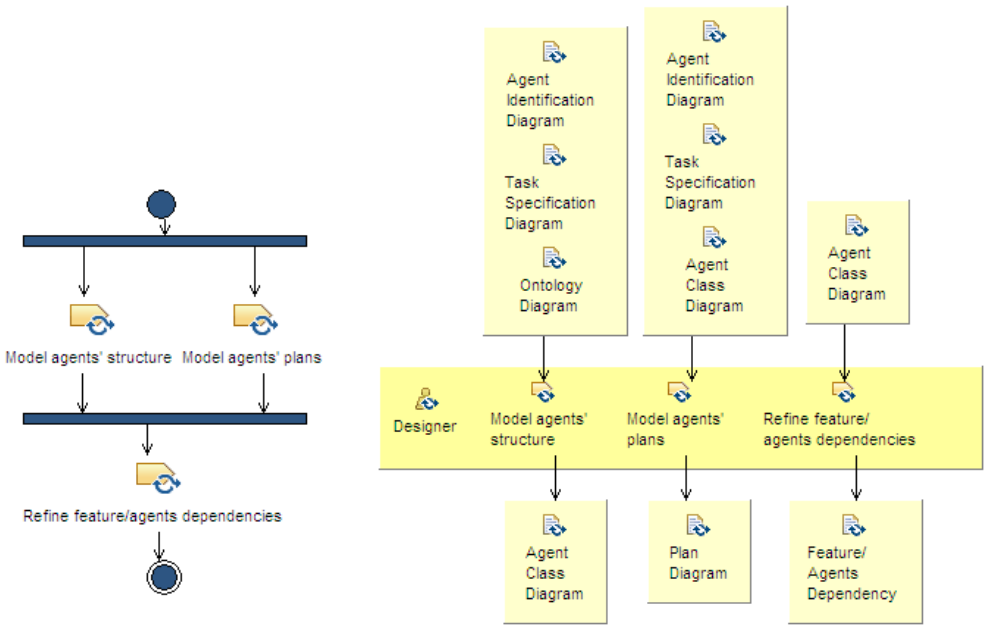
Figure 3.23: Component Modeling detailed activity diagram.

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 ≪*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 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 ≪*capability*≫ 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



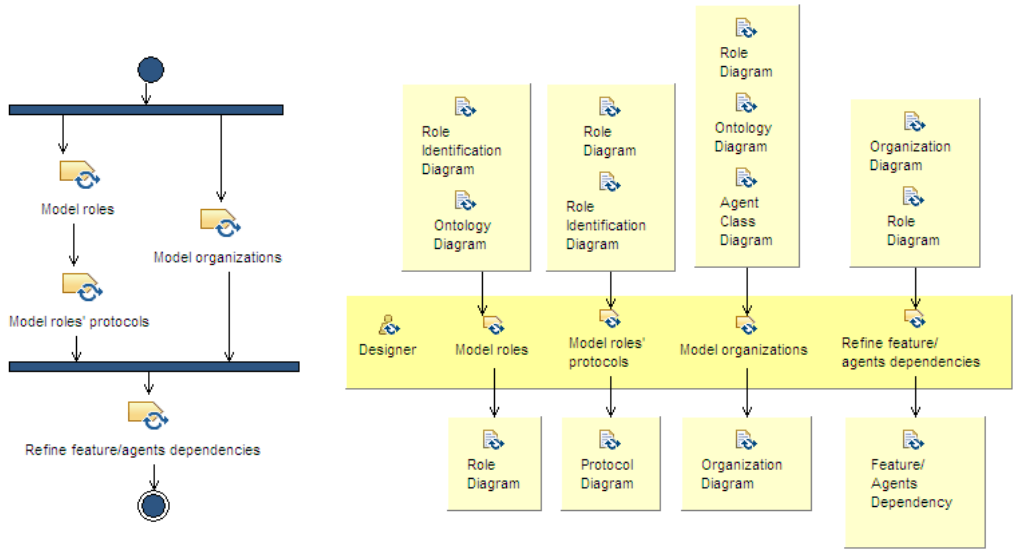
3.24(a): Activity diagram. 3.24(b): Detailed activity diagram.

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.

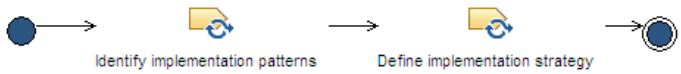


Figure 3.26: Agent Implementation Strategy Description activity diagram.

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

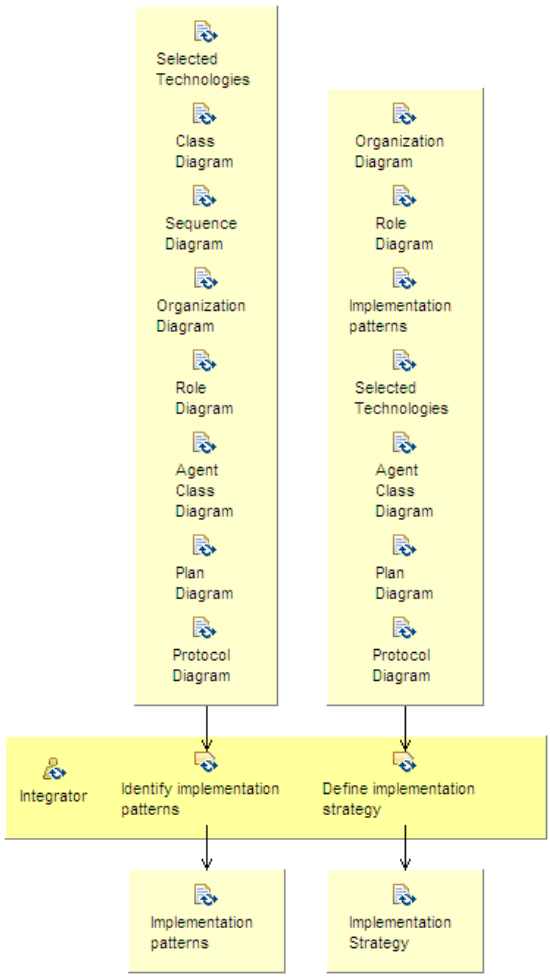


Figure 3.27: Agent Implementation Strategy Description detailed activity diagram.

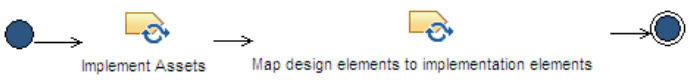


Figure 3.28: Assets Implementation activity diagram.

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.

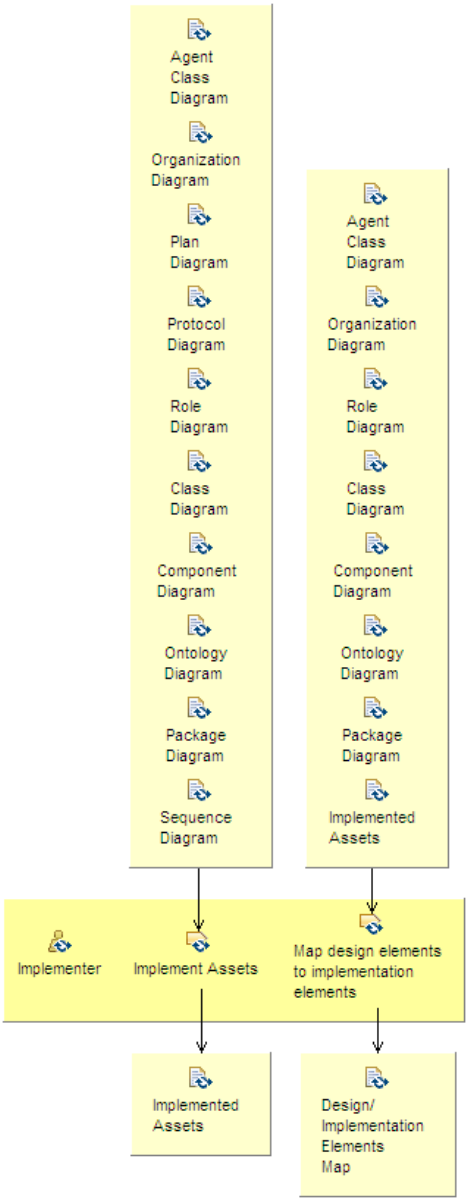


Figure 3.29: Assets Implementation detailed activity diagram.