4
Case Studies

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

4.1
ExpertCommittee Case Study

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

So, first we give an overview of the EC evolution (Section 4.1.1), detailing which functionalities composed each one of the system versions. Next, we show how we transformed the several versions of the EC system into a MAS-PL using our approach (Section 4.1.2). Our process allowed to analyze the commonalities and variabilities among the system versions, and to document and model (agent) features. As a result, we had to make some refactoring in the existing code in order to modularize the features. Besides helping on the construction of the MAS-PL, the use of our approach generates a documentation that: (i) provides a better understanding of the EC and its features; (ii) facilitates a posterior evolution of this MAS-PL to incorporate new (agent) features.

4.1.1 ExpertCommittee Overview

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

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

Table 4.1 summarizes the different versions of the EC system. Each new version was implemented based on the previous one. The first version was built without any autonomous behavior, in other words, without software agents. The second version of the EC system added a feature to the system that is related to autonomous behavior, it monitors the system and automatically sends some important message notifications. Thus, in this second version, we have used the agent abstraction and AOSE techniques to allow the introduction of this new agent feature in the system. Software agents were used to model and implement this autonomous behavior, including an agent representing each user of the system playing different roles. The third version provided new functionalities (non-agent features) to the system related to the Reviewer role; however the user agent had to be modified by the inclusion of this new role for supporting messages notifications. The following versions also included agent features and, finally, the last version changed a feature previously included (message notifications), resulting in an alternative feature.

Table 4.1: The EC versions.

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

4.1.2
Transforming the EC System Versions into a MAS-PL

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

Domain Analysis

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

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

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

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

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

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

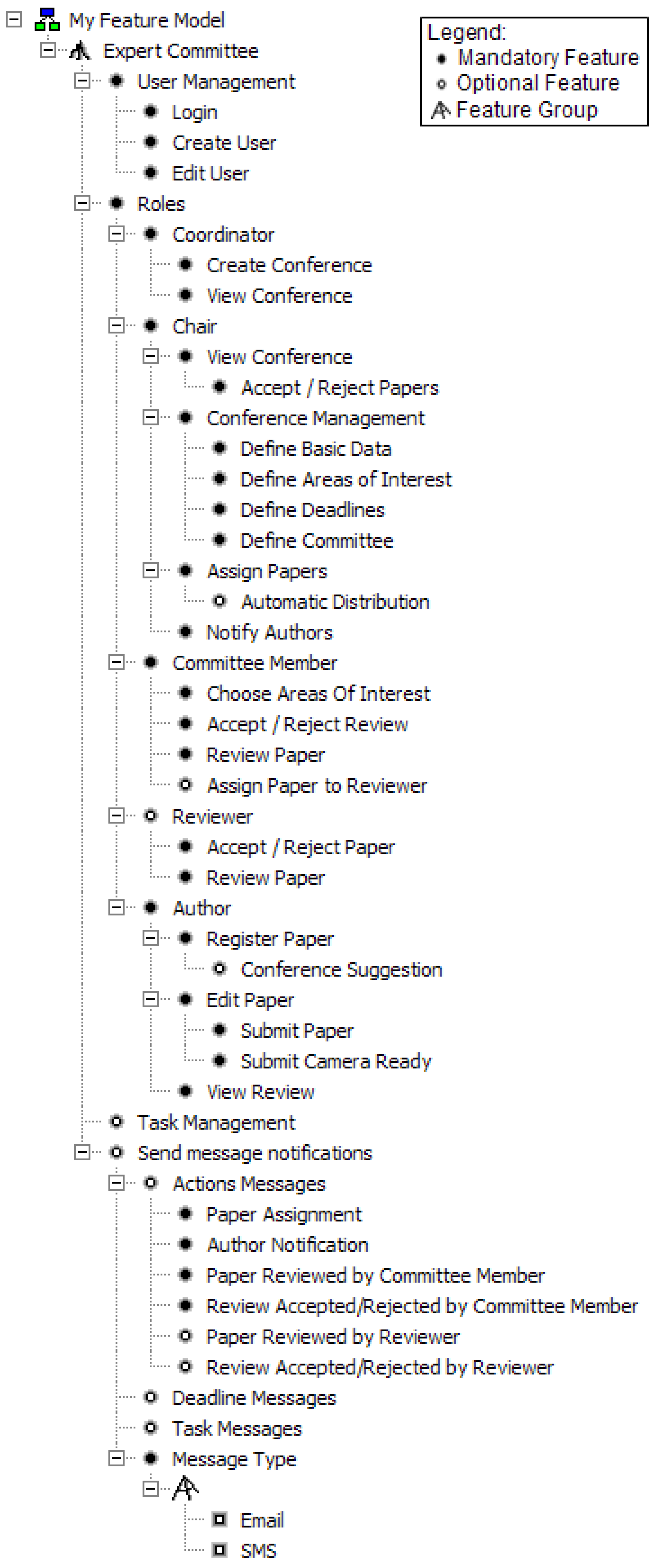


Figure 4.1: EC Feature Diagram.

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

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

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

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

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

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

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

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

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

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

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

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

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

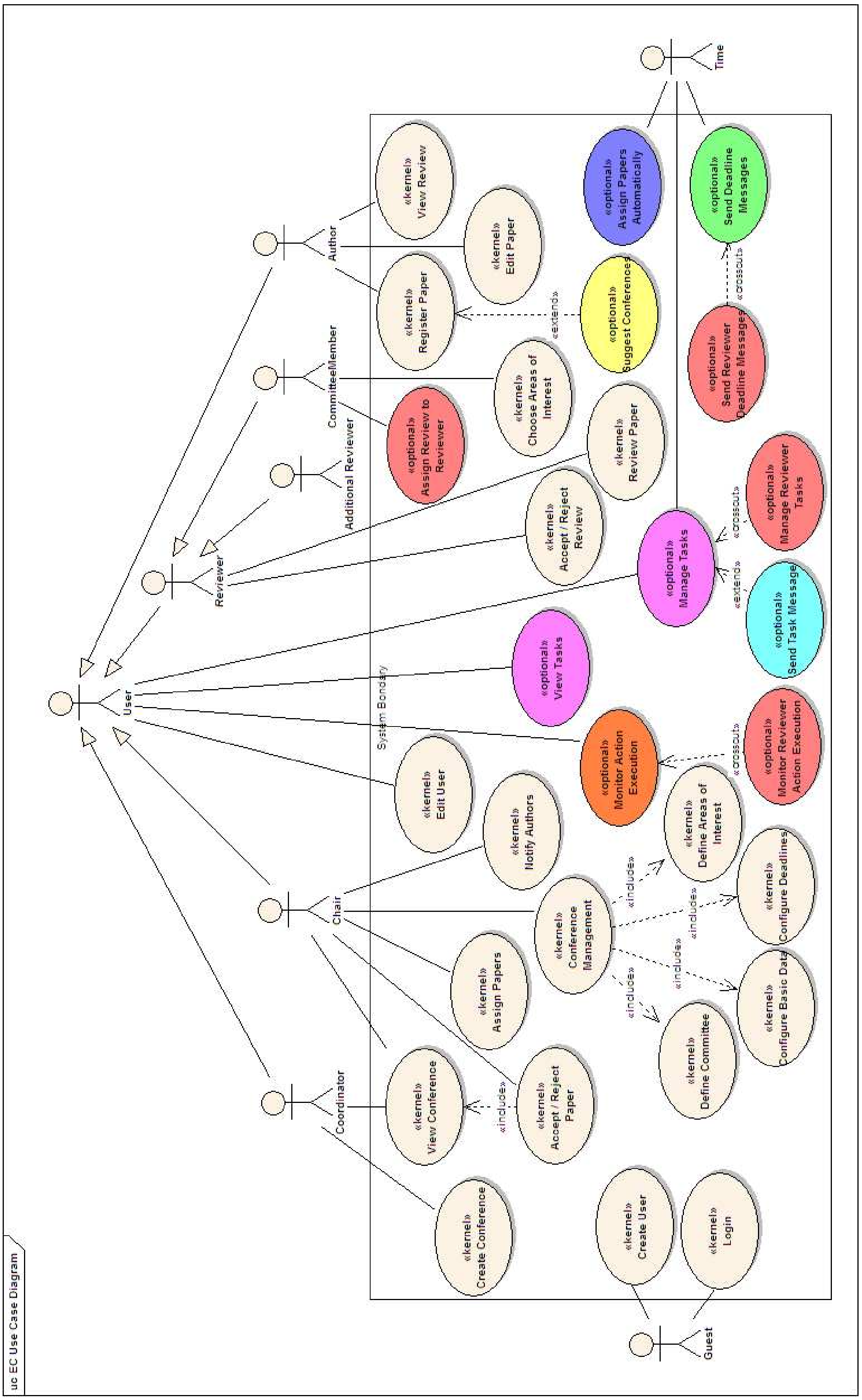

uc EC Use Case Diagram
System Bondary
User
Reviewer
Author
CommitteeMember
Additional Reviewer
Chair
Coordinator
Guest
Time
«kernel» View Review
«kernel» Edit Paper
«kernel» Register Paper
«optional» Assign Papers Automatically
«optional» Send Deadline Messages
«optional» Suggest Conferences
«extend»
«crosscut»
«kernel» Choose Areas of Interest
«optional» Send Reviewer Deadline Messages
«optional» Assign Review to Reviewer
«kernel» Review Paper
«kernel» Accept / Reject Review
«optional» Manage Reviewer Tasks
«optional» Manage Tasks
«optional» Send Task Message
«optional» View Tasks
«optional» Monitor Action Execution
«optional» Monitor Reviewer Action Execution
«kernel» Edit User
«kernel» Notify Authors
«kernel» Define Areas of Interest
«kernel» Conference Management
«include»
«kernel» Configure Deadlines
«kernel» Assign Papers
«kernel» Configure Basic Data
«kernel» Define Committee
«kernel» View Conference
«kernel» Accept / Reject Paper
«kernel» Create Conference
«kernel» Create User
«kernel» Login

Figure 4.2: EC Use Case Diagram.

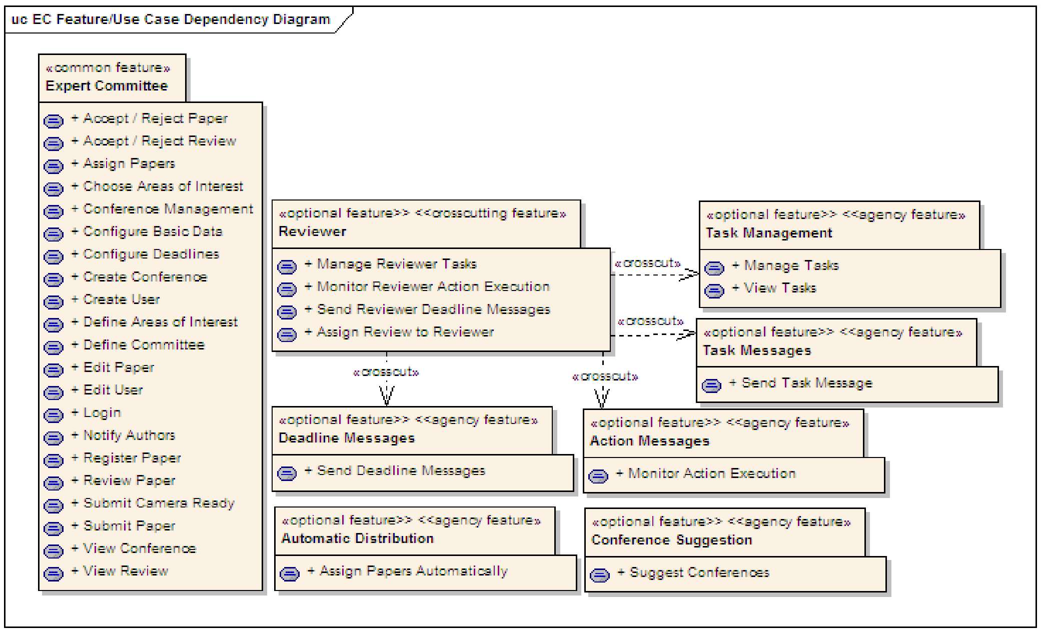

uc EC Feature/Use Case Dependency Diagram
«common feature» Expert Committee
+ Accept / Reject Paper
+ Accept / Reject Review
+ Assign Papers
+ Choose Areas of Interest
+ Conference Management
+ Configure Basic Data
+ Configure Deadlines
+ Create Conference
+ Create User
+ Define Areas of Interest
+ Define Committee
+ Edit Paper
+ Edit User
+ Login
+ Notify Authors
+ Register Paper
+ Review Paper
+ Submit Camera Ready
+ Submit Paper
+ View Conference
+ View Review
«optional feature» <<crosscutting feature» Reviewer
+ Manage Reviewer Tasks
+ Monitor Reviewer Action Execution
+ Send Reviewer Deadline Messages
+ Assign Review to Reviewer
«crosscut»
«optional feature» <<agency feature» Task Management
+ Manage Tasks
+ View Tasks
«crosscut»
«optional feature» <<agency feature» Task Messages
+ Send Task Message
«crosscut»
«crosscut»
«optional feature» <<agency feature» Deadline Messages
+ Send Deadline Messages
«optional feature» <<agency feature» Action Messages
+ Monitor Action Execution
«optional feature» <<agency feature» Automatic Distribution
+ Assign Papers Automatically
«optional feature» <<agency feature» Conference Suggestion
+ Suggest Conferences

Figure 4.3: EC Feature/Use Case Dependency Diagram.

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

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

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

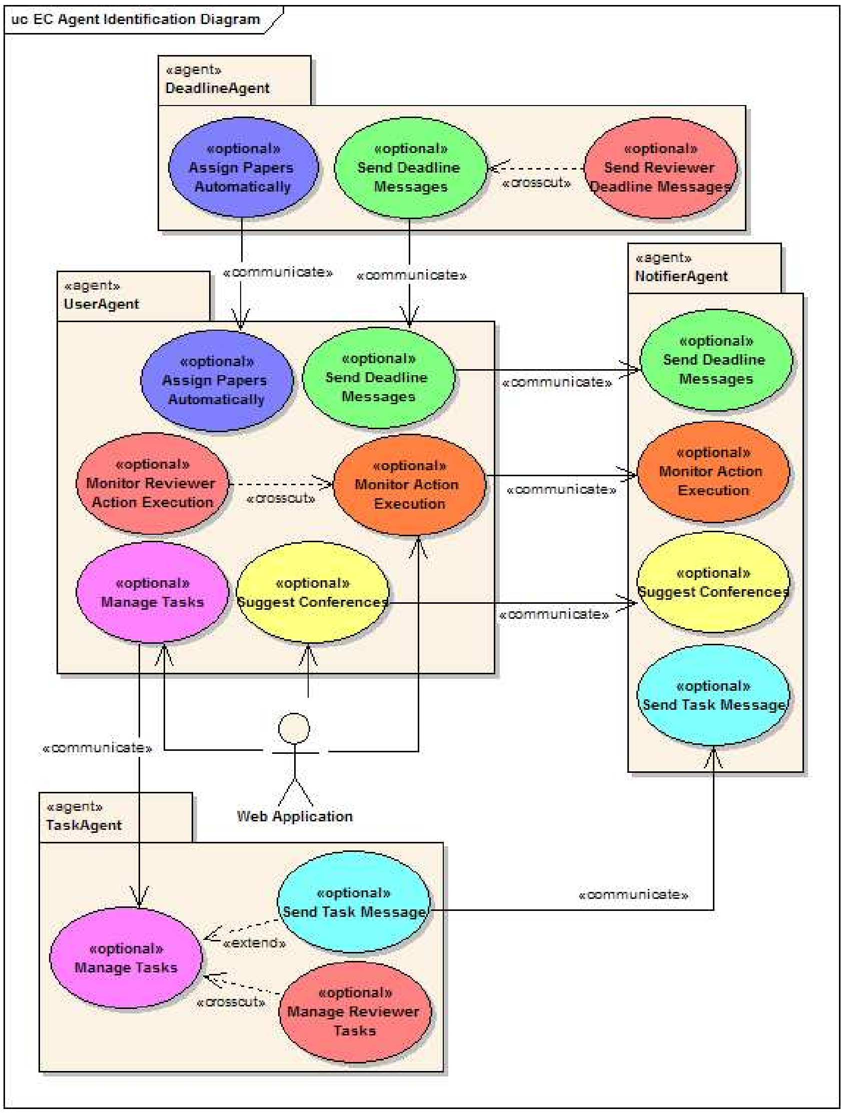

uc EC Agent Identification Diagram
«agent» DeadlineAgent
«optional» Assign Papers Automatically
«optional» Send Deadline Messages
«crosscut»
«optional» Send Reviewer Deadline Messages
«communicate»
«communicate»
«agent» UserAgent
«optional» Assign Papers Automatically
«optional» Send Deadline Messages
«optional» Monitor Reviewer Action Execution
«crosscut»
«optional» Monitor Action Execution
«optional» Manage Tasks
«optional» Suggest Conferences
«agent» NotifierAgent
«optional» Send Deadline Messages
«optional» Monitor Action Execution
«optional» Suggest Conferences
«optional» Send Task Message
«communicate»
«communicate»
«communicate»
«communicate»
Web Application
«agent» TaskAgent
«optional» Send Task Message
«extend»
«optional» Manage Tasks
«crosscut»
«optional» Manage Reviewer Tasks
«communicate»

Figure 4.4: EC Agent Identification Diagram.

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

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

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

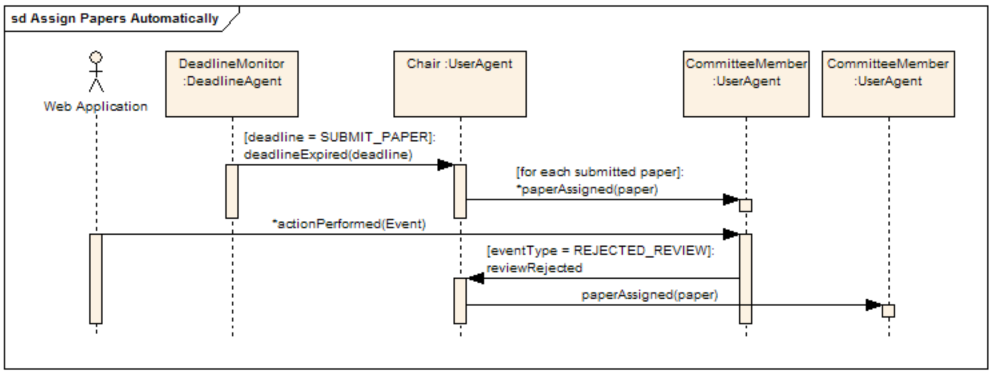

sd Assign Papers Automatically
Web Application
DeadlineMonitor :DeadlineAgent
Chair :UserAgent
CommitteeMember :UserAgent
CommitteeMember :UserAgent
[deadline = SUBMIT_PAPER]: deadlineExpired(deadline)
[for each submitted paper]: *paperAssigned(paper)
*actionPerformed(Event)
[eventType = REJECTED_REVIEW]: reviewRejected
paperAssigned(paper)

Figure 4.5: Role Identification Diagram - Assign Papers Automatically.

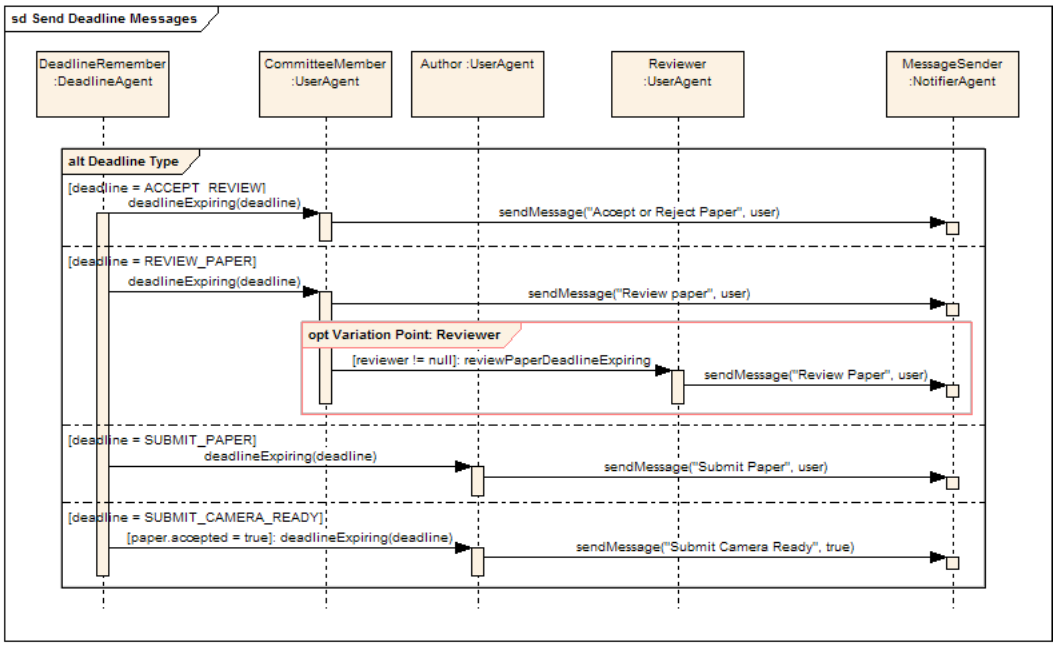
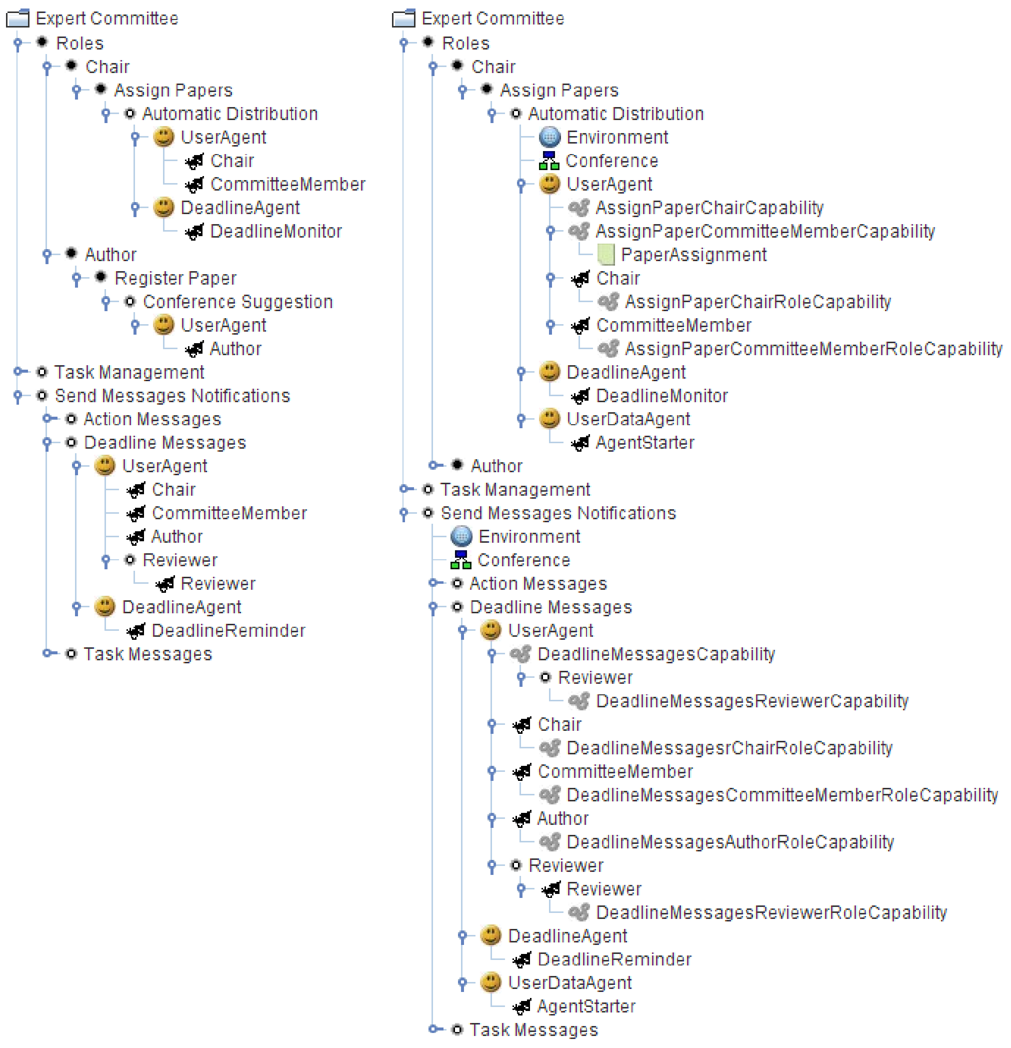

sd Send Deadline Messages
DeadlineRemember :DeadlineAgent
CommitteeMember :UserAgent
Author :UserAgent
Reviewer :UserAgent
MessageSender :NotifierAgent
alt Deadline Type
[deadline = ACCEPT_REVIEW]
deadlineExpiring(deadline)
sendMessage("Accept or Reject Paper", user)
[deadline = REVIEW_PAPER]
deadlineExpiring(deadline)
sendMessage("Review paper", user)
opt Variation Point: Reviewer
[reviewer != null]: reviewPaperDeadlineExpiring
sendMessage("Review Paper", user)
[deadline = SUBMIT_PAPER]
deadlineExpiring(deadline)
sendMessage("Submit Paper", user)
[deadline = SUBMIT_CAMERA_READY]
[paper.accepted = true]: deadlineExpiring(deadline)
sendMessage("Submit Camera Ready", true)

Figure 4.6: Role Identification Diagram - Send Deadline Messages.

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

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



Expert Committee
- Roles
  - Chair
    - Assign Papers
      - Automatic Distribution
        - UserAgent
          - Chair
          - CommitteeMember
        - DeadlineAgent
          - DeadlineMonitor
  - Author
    - Register Paper
      - Conference Suggestion
        - UserAgent
          - Author
- Task Management
- Send Messages Notifications
  - Action Messages
  - Deadline Messages
    - UserAgent
      - Chair
      - CommitteeMember
      - Author
      - Reviewer
        - Reviewer
    - DeadlineAgent
      - DeadlineReminder
  - Task Messages

4.7(a): Analysis Level

Expert Committee
- Roles
  - Chair
    - Assign Papers
      - Automatic Distribution
        - Environment
        - Conference
        - UserAgent
          - AssignPaperChairCapability
          - AssignPaperCommitteeMemberCapability
            - PaperAssignment
          - Chair
            - AssignPaperChairRoleCapability
          - CommitteeMember
            - AssignPaperCommitteeMemberRoleCapability
        - DeadlineAgent
          - DeadlineMonitor
        - UserDataAgent
          - AgentStarter
  - Author
- Task Management
- Send Messages Notifications
  - Environment
  - Conference
  - Action Messages
  - Deadline Messages
    - UserAgent
      - DeadlineMessagesCapability
        - Reviewer
          - DeadlineMessagesReviewerCapability
      - Chair
        - DeadlineMessagesrChairRoleCapability
      - CommitteeMember
        - DeadlineMessagesCommitteeMemberRoleCapability
      - Author
        - DeadlineMessagesAuthorRoleCapability
      - Reviewer
        - Reviewer
          - DeadlineMessagesReviewerRoleCapability
    - DeadlineAgent
      - DeadlineReminder
    - UserDataAgent
      - AgentStarter
  - Task Messages

4.7(b): Design Level

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

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

Domain Design

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

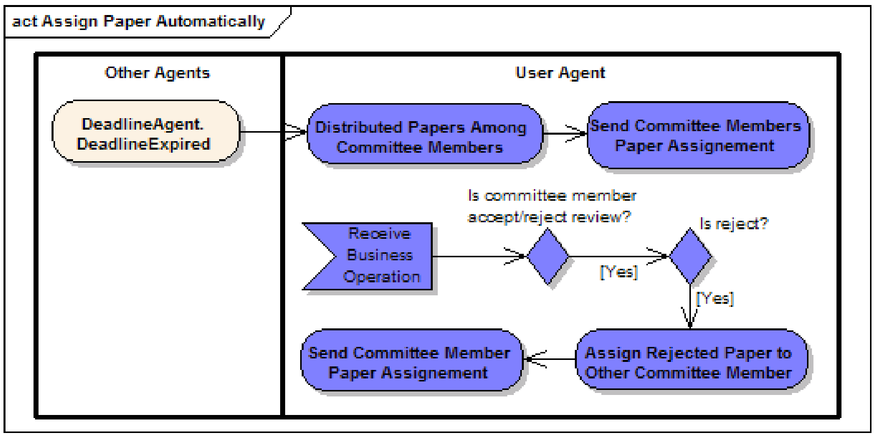

act Assign Paper Automatically
Other Agents
User Agent
DeadlineAgent. DeadlineExpired
Distributed Papers Among Committee Members
Send Committee Members Paper Assignement
Is committee member accept/reject review?
Is reject?
Receive Business Operation
[Yes]
[Yes]
Send Committee Member Paper Assignement
Assign Rejected Paper to Other Committee Member

Figure 4.8: Task Specification Diagram - Assign Paper Automatically.

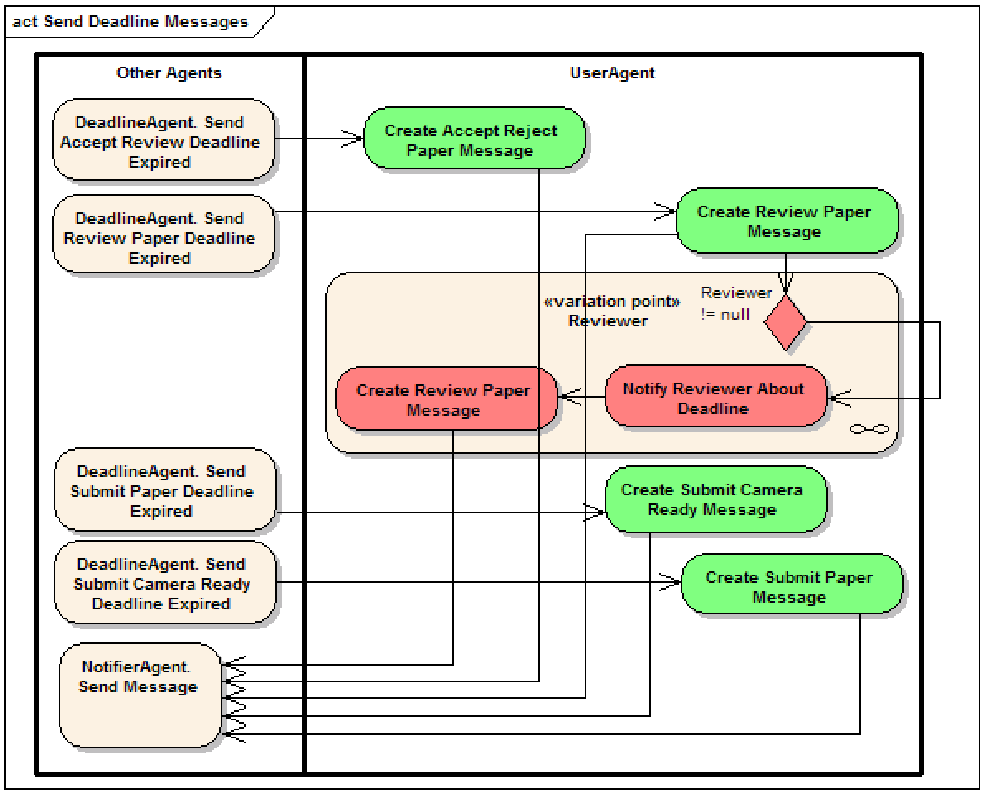

act Send Deadline Messages
Other Agents
UserAgent
DeadlineAgent. Send Accept Review Deadline Expired
Create Accept Reject Paper Message
DeadlineAgent. Send Review Paper Deadline Expired
Create Review Paper Message
«variation point» Reviewer
Reviewer != null
Create Review Paper Message
Notify Reviewer About Deadline
DeadlineAgent. Send Submit Paper Deadline Expired
Create Submit Camera Ready Message
DeadlineAgent. Send Submit Camera Ready Deadline Expired
Create Submit Paper Message
NotifierAgent. Send Message

Figure 4.9: Task Specification Diagram - Send Deadline Messages.

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

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

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

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

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

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

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

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

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

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

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

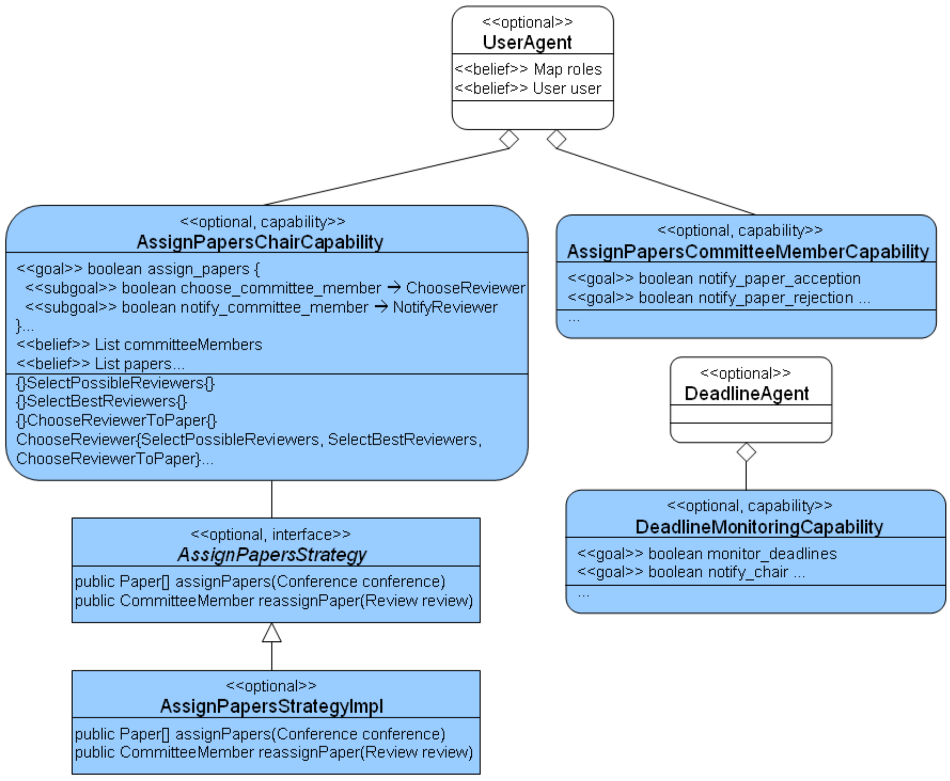
`TaskAgent`: this agent is responsible for managing the user tasks. It receives requests for creating, removing and setting the execution date of tasks. The requests are made by the `UserAgent`s;

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

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

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

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



<<optional>>
UserAgent
<<belief>> Map roles
<<belief>> User user
<<optional, capability>>
AssignPapersChairCapability
<<goal>> boolean assign_papers {
<<subgoal>> boolean choose_committee_member → ChooseReviewer
<<subgoal>> boolean notify_committee_member → NotifyReviewer
}...
<<belief>> List committeeMembers
<<belief>> List papers...
{}SelectPossibleReviewers{}
{}SelectBestReviewers{}
{}ChooseReviewerToPaper{}
ChooseReviewer{SelectPossibleReviewers, SelectBestReviewers, ChooseReviewerToPaper}...
<<optional, capability>>
AssignPapersCommitteeMemberCapability
<<goal>> boolean notify_paper_acception
<<goal>> boolean notify_paper_rejection ...
...
<<optional>>
DeadlineAgent
<<optional, capability>>
DeadlineMonitoringCapability
<<goal>> boolean monitor_deadlines
<<goal>> boolean notify_chair ...
...
<<optional, interface>>
AssignPapersStrategy
public Paper[] assignPapers(Conference conference)
public CommitteeMember reassignPaper(Review review)
<<optional>>
AssignPapersStrategyImpl
public Paper[] assignPapers(Conference conference)
public CommitteeMember reassignPaper(Review review)

Figure 4.10: EC Class Diagram.

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

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

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

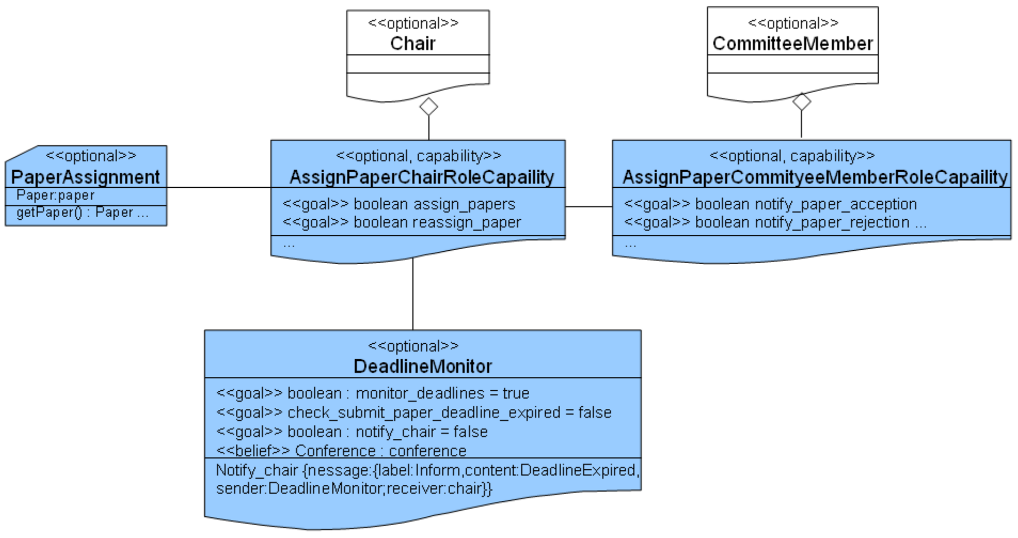

<<optional>>
Chair
<<optional>>
CommitteeMember
<<optional>>
PaperAssignment
Paper:paper
getPaper() : Paper ...
<<optional, capability>>
AssignPaperChairRoleCapaility
<<goal>> boolean assign_papers
<<goal>> boolean reassign_paper
...
<<optional, capability>>
AssignPaperCommityeeMemberRoleCapaility
<<goal>> boolean notify_paper_acception
<<goal>> boolean notify_paper_rejection ...
...
<<optional>>
DeadlineMonitor
<<goal>> boolean : monitor_deadlines = true
<<goal>> check_submit_paper_deadline_expired = false
<<goal>> boolean : notify_chair = false
<<belief>> Conference : conference
Notify_chair {nessage:{label:Inform,content:DeadlineExpired,
sender:DeadlineMonitor;receiver:chair}}

Figure 4.11: EC Role Diagram.

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

Domain Realization

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

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

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

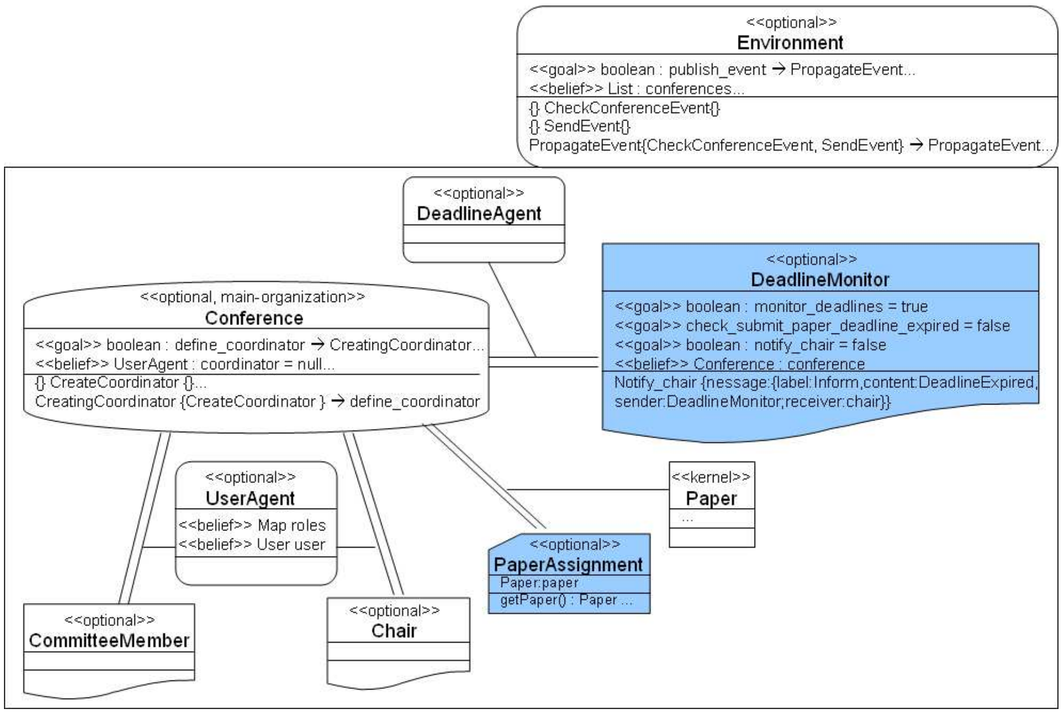

<<optional>>
Environment
<<goal>> boolean : publish_event → PropagateEvent...
<<belief>> List : conferences...
{} CheckConferenceEvent{}
{} SendEvent{}
PropagateEvent{CheckConferenceEvent, SendEvent} → PropagateEvent...
<<optional>>
DeadlineAgent
<<optional, main-organization>>
Conference
<<goal>> boolean : define_coordinator → CreatingCoordinator...
<<belief>> UserAgent : coordinator = null...
{} CreateCoordinator {}...
CreatingCoordinator {CreateCoordinator } → define_coordinator
<<optional>>
DeadlineMonitor
<<goal>> boolean : monitor_deadlines = true
<<goal>> check_submit_paper_deadline_expired = false
<<goal>> boolean : notify_chair = false
<<belief>> Conference : conference
Notify_chair {message:{label:Inform,content:DeadlineExpired, sender:DeadlineMonitor;receiver:chair}}
<<optional>>
UserAgent
<<belief>> Map roles
<<belief>> User user
<<kernel>>
Paper
...
<<optional>>
PaperAssignment
Paper:paper
getPaper() : Paper ...
<<optional>>
CommitteeMember
<<optional>>
Chair

Figure 4.12: EC Organization Diagram.

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

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

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

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

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

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

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

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

Listing 4.1: XML Configuration File.

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

The EC MAS-PL was implemented using JADE framework, so agents, roles and plans were developed by extending the classes provided by the framework. Figure 4.13 shows the EC implementation elements, such as classes, interfaces, mapped to the design elements (partial view). It presents some of the techniques detailed in the previous Section: (i) the `Environment` was implemented as an agent with a JADE behavior that publishes events to other agents; (ii) the `UserAgent` was implemented with the `UserAgent` interface and the `UserAgentCore` and `UserAgentRole` classes, which are according to the Role Pattern; and (iii) some design elements were implemented by code fragments that are delimitated by tags that allow the conditional compilation.

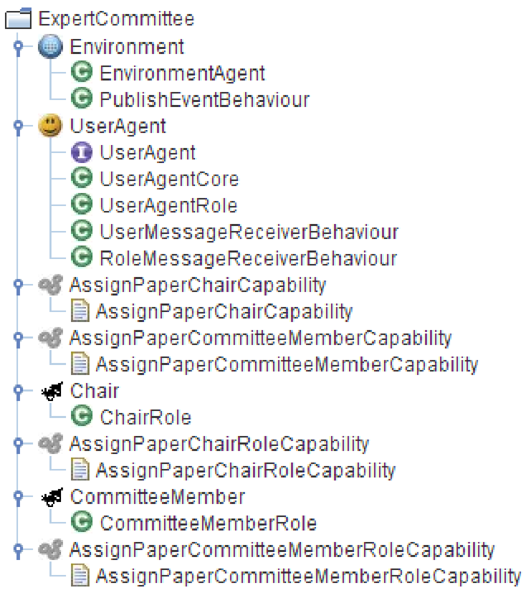

```
ExpertCommittee
├─ Environment
│   ├─ EnvironmentAgent
│   └─ PublishEventBehaviour
├─ UserAgent
│   ├─ UserAgent
│   ├─ UserAgentCore
│   ├─ UserAgentRole
│   ├─ UserMessageReceiverBehaviour
│   └─ RoleMessageReceiverBehaviour
├─ AssignPaperChairCapability
│   └─ AssignPaperChairCapability
├─ AssignPaperCommitteeMemberCapability
│   └─ AssignPaperCommitteeMemberCapability
├─ Chair
│   └─ ChairRole
├─ AssignPaperChairRoleCapability
│   └─ AssignPaperChairRoleCapability
├─ CommitteeMember
│   └─ CommitteeMemberRole
└─ AssignPaperCommitteeMemberRoleCapability
    └─ AssignPaperCommitteeMemberRoleCapability
```

Figure 4.13: EC Design/Implementation Elements Mapping.

4.2
OnLine Intelligent Services (OLIS) Case Study

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

In this Section, we start by introducing the OLIS MAS-PL detailing the features that compose it (Section 4.2.1). Second, we present the modeling of this MAS-PL using our development process (Section 4.2.2). The activities of the process were performed each time that a new feature was added to the product line, resulting at the end in the artifacts presented here.

4.2.1 The OLIS MAS-PL Overview

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

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

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

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

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

Events Suggestion. When a new event is announced, the system automatically recommends the event after checking if it is interesting to the users based on their preferences. The system also checks if the weather is going to be appropriate according to the place type where the event is going to take place; and

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

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

4.2.2 Developing OLIS with our Process

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

Domain Analysis

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

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

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

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

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

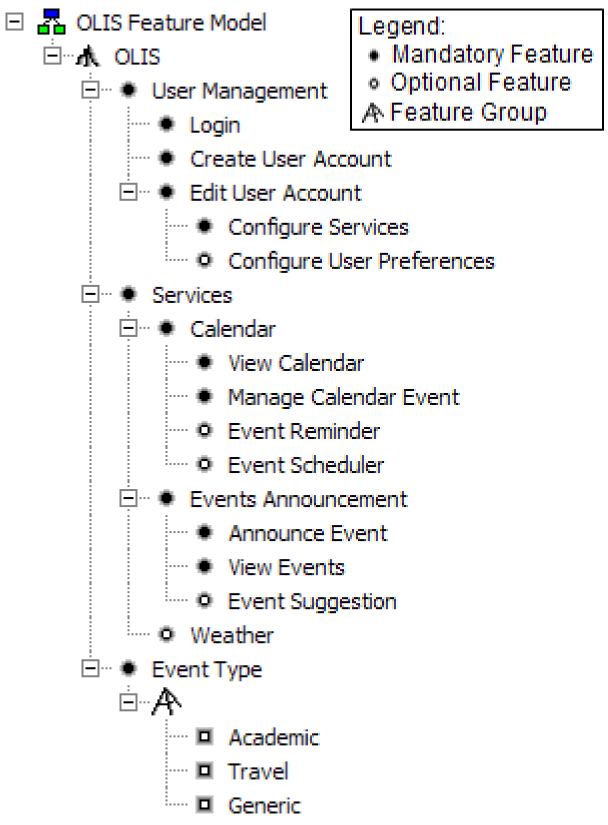


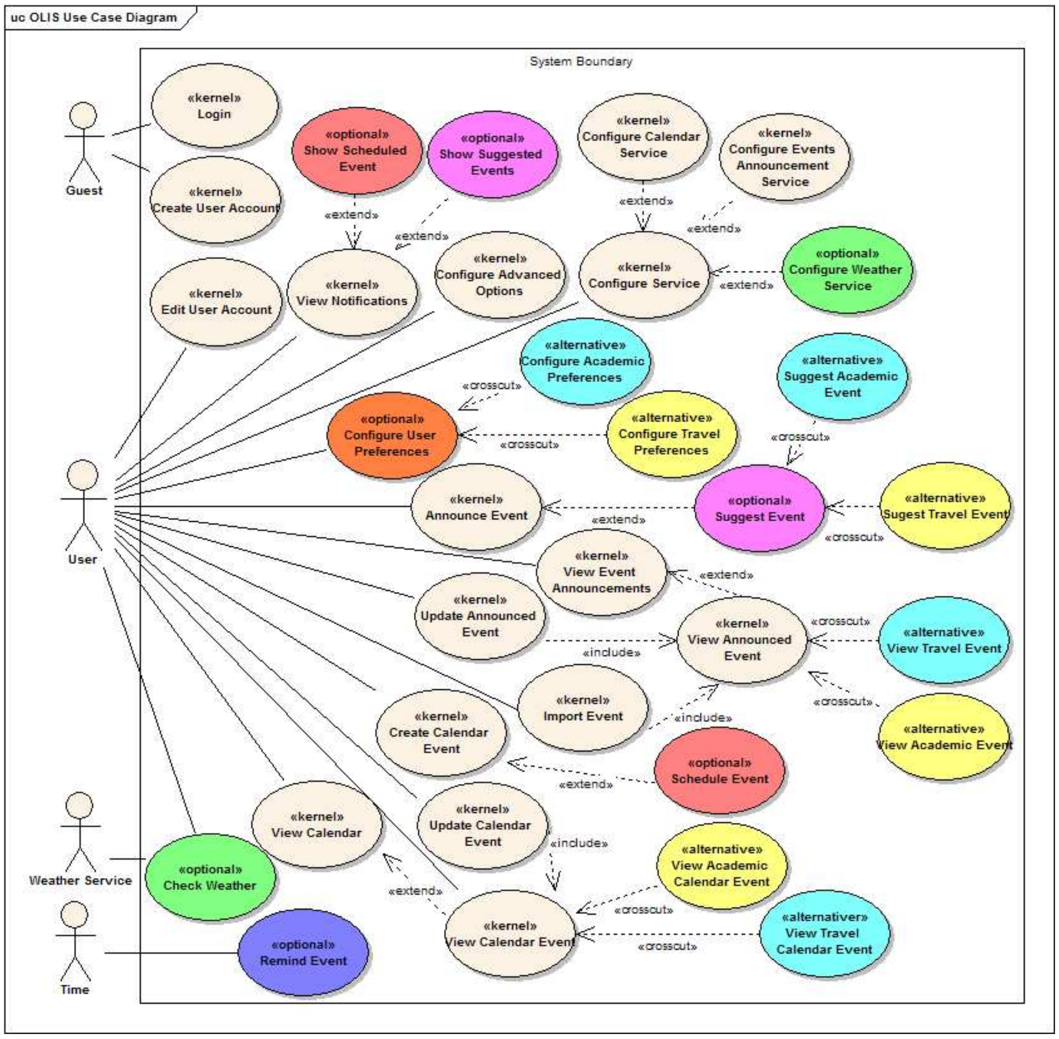
Figure 4.14: EC Feature Diagram.

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

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

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

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



uc OLIS Use Case Diagram
System Boundary
Guest
User
Weather Service
Time
«kernel» Login
«kernel» Create User Account
«kernel» Edit User Account
«optional» Show Scheduled Event
«optional» Show Suggested Events
«kernel» View Notifications
«kernel» Configure Advanced Options
«kernel» Configure Calendar Service
«kernel» Configure Events Announcement Service
«kernel» Configure Service
«optional» Configure Weather Service
«alternative» Configure Academic Preferences
«optional» Configure User Preferences
«alternative» Configure Travel Preferences
«alternative» Suggest Academic Event
«optional» Suggest Event
«alternative» Sugest Travel Event
«kernel» Announce Event
«kernel» View Event Announcements
«kernel» Update Announced Event
«kernel» View Announced Event
«alternative» View Travel Event
«alternative» View Academic Event
«kernel» Import Event
«kernel» Create Calendar Event
«optional» Schedule Event
«kernel» View Calendar
«kernel» Update Calendar Event
«alternative» View Academic Calendar Event
«kernel» View Calendar Event
«alternativer» View Travel Calendar Event
«optional» Check Weather
«optional» Remind Event
«extend»
«crosscut»
«include»

Figure 4.15: OLIS Use Case Diagram.

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

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

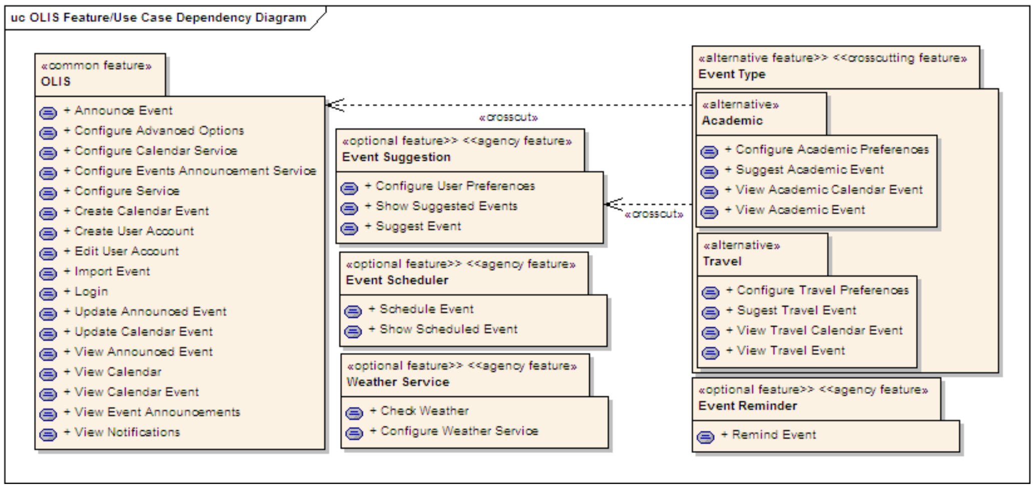


Figure 4.16: OLIS Feature/Use Case Dependency model.

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

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

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

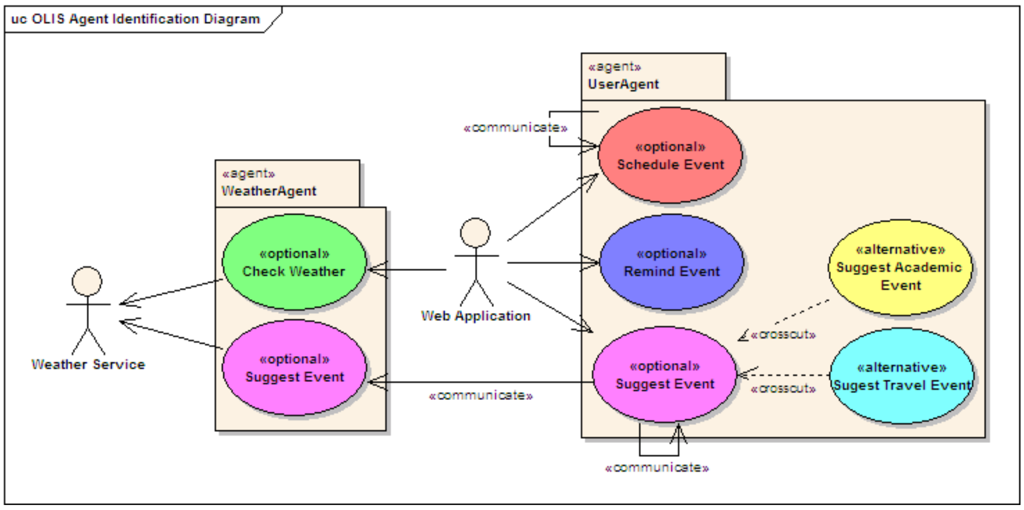


Figure 4.17: OLIS Agent Identification Diagram.

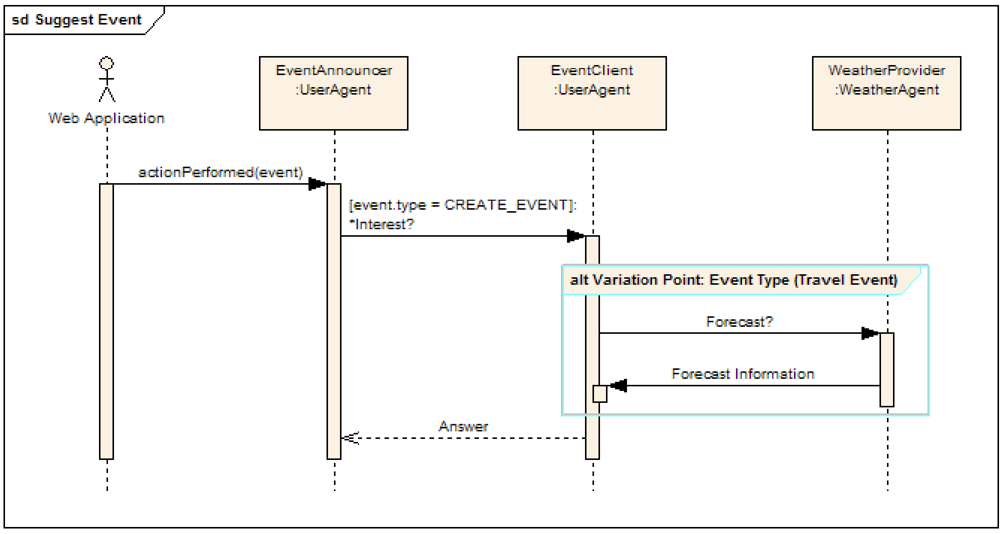
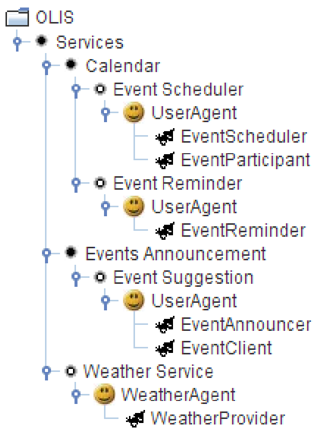

sd Suggest Event
Web Application
EventAnnouncer :UserAgent
EventClient :UserAgent
WeatherProvider :WeatherAgent
actionPerformed(event)
[event.type = CREATE_EVENT]: *Interest?
alt Variation Point: Event Type (Travel Event)
Forecast?
Forecast Information
Answer

Figure 4.18: Role Identification Diagram - Event Suggestion.

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

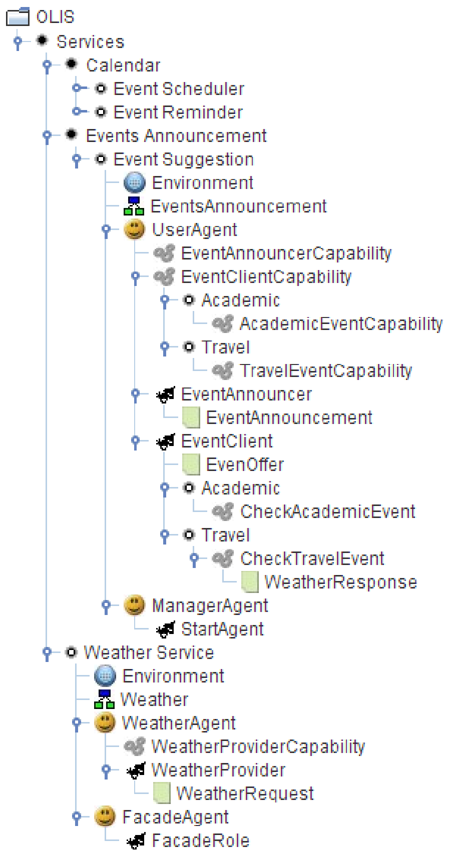
Even though there is a communication between the `UserAgent` and the `WeatherAgent` in the *Event Suggestion* feature when the event type is *Travel*, there is no direct dependency between the *Event Suggestion* and the `WeatherAgent`. However, there is an indirect dependency because there is a constraint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

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



```
OLIS
  Services
    Calendar
      Event Scheduler
        UserAgent
          EventScheduler
          EventParticipant
      Event Reminder
        UserAgent
          EventReminder
    Events Announcement
      Event Suggestion
        UserAgent
          EventAnnouncer
          EventClient
    Weather Service
      WeatherAgent
        WeatherProvider
```

4.19(a): Analysis Level



```
OLIS
  Services
    Calendar
      Event Scheduler
      Event Reminder
    Events Announcement
      Event Suggestion
        Environment
        EventsAnnouncement
        UserAgent
          EventAnnouncerCapability
          EventClientCapability
            Academic
              AcademicEventCapability
            Travel
              TravelEventCapability
          EventAnnouncer
            EventAnnouncement
          EventClient
            EvenOffer
            Academic
              CheckAcademicEvent
            Travel
              CheckTravelEvent
                WeatherResponse
        ManagerAgent
          StartAgent
    Weather Service
      Environment
      Weather
      WeatherAgent
        WeatherProviderCapability
        WeatherProvider
          WeatherRequest
      FacadeAgent
        FacadeRole
```

4.19(b): Design Level

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

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

Domain Design

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

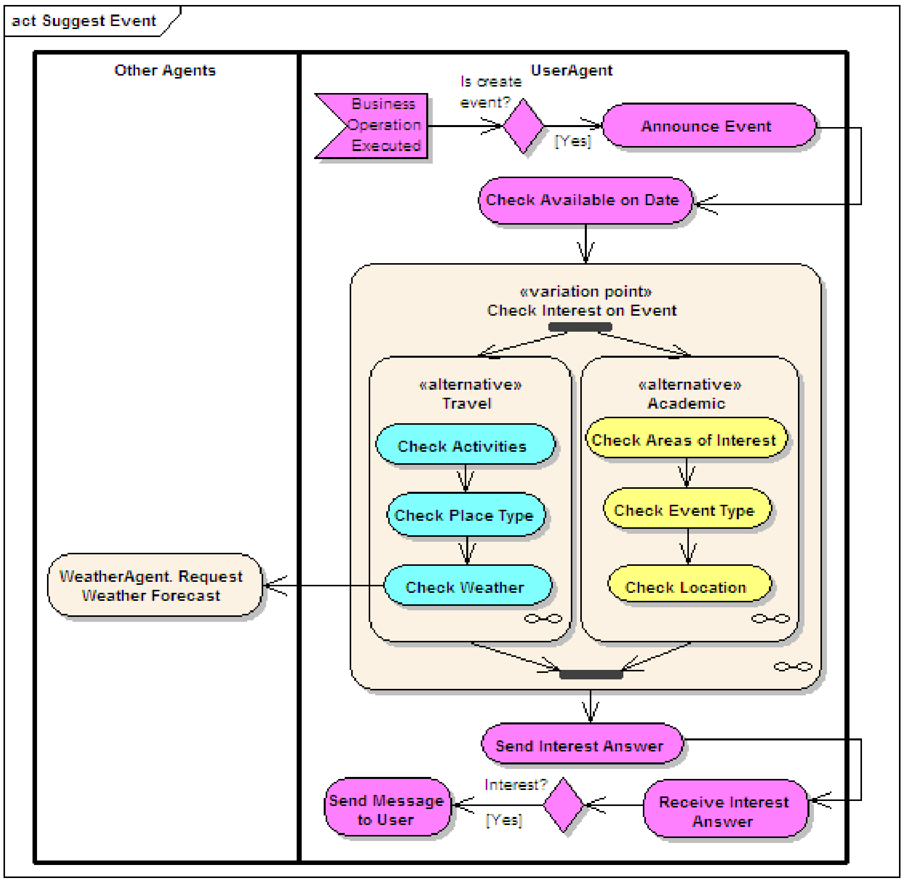


Figure 4.20: Task Specification Diagram - Event Suggestion.

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

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

This provides an easy configuration of the GUI layer.

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

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

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

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

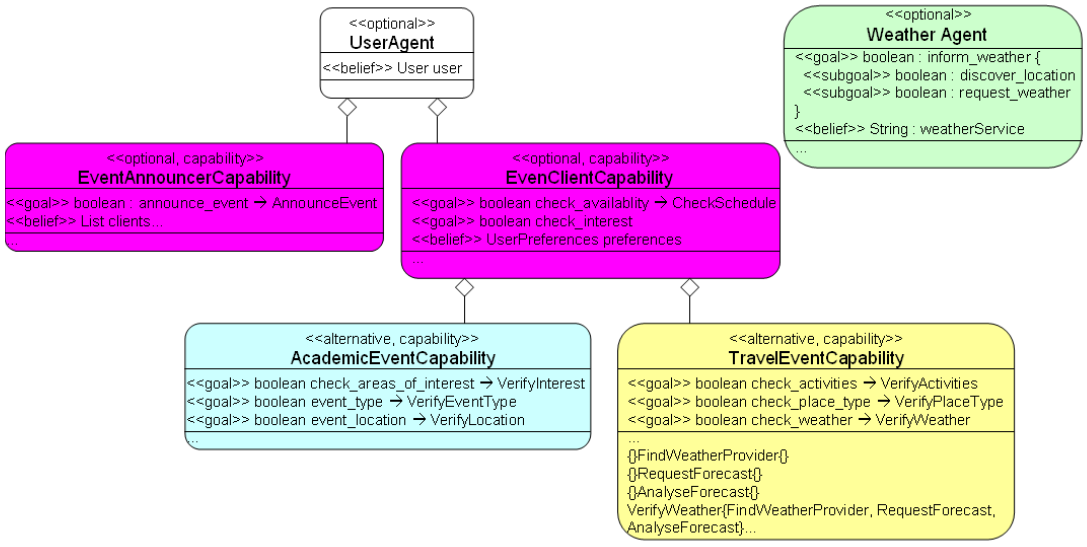
`WeatherAgent:` this agent provides the weather information. It looks for the current weather conditions and the forecast of a specific location;

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

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

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

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



<<optional>>
UserAgent
<<belief>> User user
<<optional>>
Weather Agent
<<goal>> boolean : inform_weather {
<<subgoal>> boolean : discover_location
<<subgoal>> boolean : request_weather
}
<<belief>> String : weatherService
...
<<optional, capability>>
EventAnnouncerCapability
<<goal>> boolean : announce_event → AnnounceEvent
<<belief>> List clients...
...
<<optional, capability>>
EvenClientCapability
<<goal>> boolean check_availablity → CheckSchedule
<<goal>> boolean check_interest
<<belief>> UserPreferences preferences
...
<<alternative, capability>>
AcademicEventCapability
<<goal>> boolean check_areas_of_interest → VerifyInterest
<<goal>> boolean event_type → VerifyEventType
<<goal>> boolean event_location → VerifyLocation
...
<<alternative, capability>>
TravelEventCapability
<<goal>> boolean check_activities → VerifyActivities
<<goal>> boolean check_place_type → VerifyPlaceType
<<goal>> boolean check_weather → VerifyWeather
...
{}FindWeatherProvider{}
{}RequestForecast{}
{}AnalyseForecast{}
VerifyWeather{FindWeatherProvider, RequestForecast, AnalyseForecast}...

Figure 4.21: OLIS Class Diagram.

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

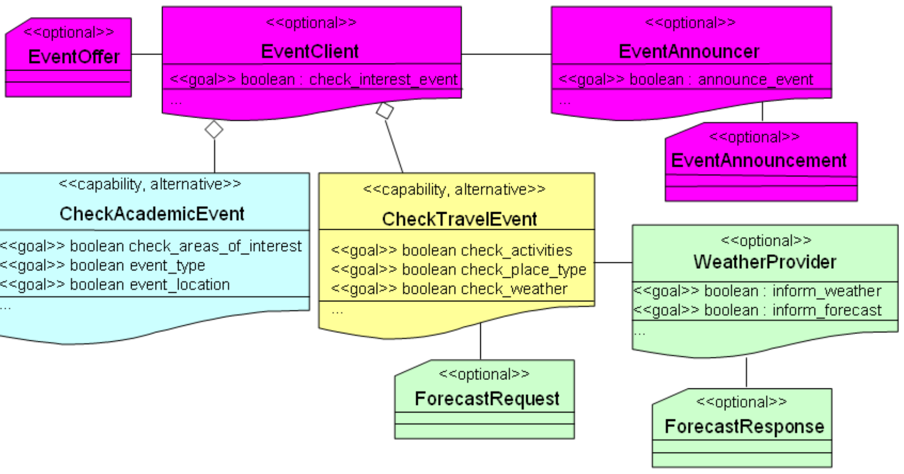

<<optional>>
EventOffer
<<optional>>
EventClient
<<goal>> boolean : check_interest_event
<<optional>>
EventAnnouncer
<<goal>> boolean : announce_event
<<optional>>
EventAnnouncement
<<capability, alternative>>
CheckAcademicEvent
<<goal>> boolean check_areas_of_interest
<<goal>> boolean event_type
<<goal>> boolean event_location
<<capability, alternative>>
CheckTravelEvent
<<goal>> boolean check_activities
<<goal>> boolean check_place_type
<<goal>> boolean check_weather
<<optional>>
WeatherProvider
<<goal>> boolean : inform_weather
<<goal>> boolean : inform_forecast
<<optional>>
ForecastRequest
<<optional>>
ForecastResponse

Figure 4.22: OLIS Role Diagram.

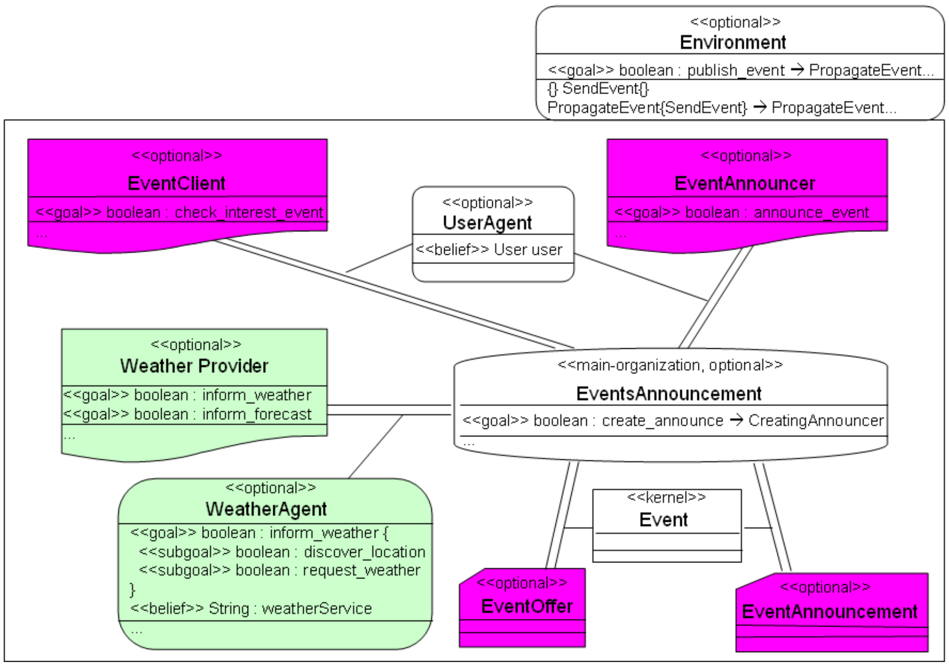
– were each one modeled in a separate sequence diagram. Even though this feature is crosscut by the *Event Type* feature, the agents' variable parts were modularized into capabilities. For instance, when a `UserAgent` has a goal of verifying interest in some event, this goal is achieved in a certain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

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

Domain Realization

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

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



<<optional>>
Environment
<<goal>> boolean : publish_event → PropagateEvent...
{} SendEvent{}
PropagateEvent{SendEvent} → PropagateEvent...
<<optional>>
EventClient
<<goal>> boolean : check_interest_event
...
<<optional>>
UserAgent
<<belief>> User user
<<optional>>
EventAnnouncer
<<goal>> boolean : announce_event
...
<<optional>>
Weather Provider
<<goal>> boolean : inform_weather
<<goal>> boolean : inform_forecast
...
<<main-organization, optional>>
EventsAnnouncement
<<goal>> boolean : create_announce → CreatingAnnouncer
...
<<optional>>
WeatherAgent
<<goal>> boolean : inform_weather {
<<subgoal>> boolean : discover_location
<<subgoal>> boolean : request_weather
}
<<belief>> String : weatherService
...
<<kernel>>
Event
<<optional>>
EventOffer
<<optional>>
EventAnnouncement

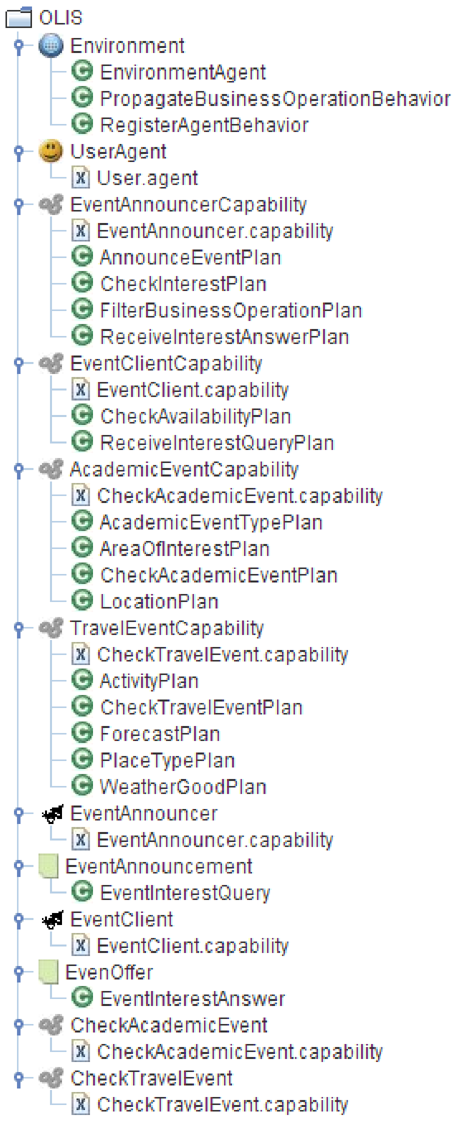
Figure 4.23: OLIS Organization Diagram.

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

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

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

The last task of the Assets Implementation activity is to map the implementation elements to the design elements. Figure 4.24 illustrates a partial view of this mapping for the OLIS. As the agents of this MAS-PL were implemented using JADE and Jadex frameworks, some agents have only classes associated to them and others have XML files too. The environment was implemented with JADE, so three classes (`EnvironmentAgent`, `PropagateBusinessOperationBehavior` and `RegisterAgentBehavior`) are associated with it. The other agents in Figure 4.24 were implemented with Jadex, so they have an XML file that implements agents and some Java classes that implement plans. Similarly, capabilities have an XML file and classes associated with them. Jadex does not provide a way to model roles, so the capability XML implements this concept.



```
OLIS
  Environment
    EnvironmentAgent
    PropagateBusinessOperationBehavior
    RegisterAgentBehavior
  UserAgent
    User.agent
  EventAnnouncerCapability
    EventAnnouncer.capability
    AnnounceEventPlan
    CheckInterestPlan
    FilterBusinessOperationPlan
    ReceiveInterestAnswerPlan
  EventClientCapability
    EventClient.capability
    CheckAvailabilityPlan
    ReceiveInterestQueryPlan
  AcademicEventCapability
    CheckAcademicEvent.capability
    AcademicEventTypePlan
    AreaOfInterestPlan
    CheckAcademicEventPlan
    LocationPlan
  TravelEventCapability
    CheckTravelEvent.capability
    ActivityPlan
    CheckTravelEventPlan
    ForecastPlan
    PlaceTypePlan
    WeatherGoodPlan
  EventAnnouncer
    EventAnnouncer.capability
  EventAnnouncement
    EventInterestQuery
  EventClient
    EventClient.capability
  EvenOffer
    EventInterestAnswer
  CheckAcademicEvent
    CheckAcademicEvent.capability
  CheckTravelEvent
    CheckTravelEvent.capability
```

Figure 4.24: OLIS Design/Implementation Elements Mapping.

4.3 Final Remarks

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

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