PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 10/07

An Aspect-Oriented Software Architecture for Code Mobility

Cidiane Aracaty Lobato
Alessandro Fabricio Garcia
Alexander Romanovsky
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 10/07 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena June, 2007

An Aspect-Oriented Software Architecture for Code Mobility*

Cidiane Aracaty Lobato, Alessandro Fabricio Garcia[1],
Alexander Romanovsky[2], Carlos José Pereira de Lucena

[1]Computing Department, Lancaster University, Lancaster, UK
[2]Computing Science School, University of Newcastle, Newcastle upon Tyne, UK

{cidiane, lucena}@inf.puc-rio.br, garciaa@comp.lancs.ac.uk,
alexander.romanovsky@newcastle.ac.uk

Abstract. Mobile agents have come forward as a technique for tackling the complexity of open distributed applications. However, the pervasive nature of code mobility implies that it cannot be modularized using only object-oriented (OO) concepts. In fact, developers frequently evidence the presence of mobility tangling and scattering in their modules. Despite these problems, they usually rely on OO application programming interfaces (APIs) offered by the mobility platforms. Such API-oriented designs suffer a number of architectural restrictions and there is a pressing need for empowering developers with an architecture supporting a flexible incorporation of code mobility in the agent applications. This work presents an aspect-oriented software architecture, called ArchM, ensuring a clean modularization, a more straightforward introduction, and an improved variability of code mobility in mobile agent systems. It addresses OO APIs' restrictions and is independent on specific platforms and applications. An ArchM implementation provides solutions to fine-grained problems related to mobility tangling and scattering in the implementation level. The usefulness and usability of ArchM has been assessed within the context of two case studies, and through its composition with two mobility platforms.

Keywords: Mobile Agents; Aspect-Oriented Software Development; Reuse.

Resumo. Agentes móveis são utilizados como uma técnica para o tratamento da complexidade de aplicações distribuídas abertas. Contudo, por sua própria natureza, a mobilidade de código não pode ser modularizada usando apenas conceitos da Orientação a Objetos (OO). De fato, desenvolvedores freqüentemente evidenciam o espalhamento e entrelaçamento da mobilidade de código e, apesar disso, têm se utilizado apenas de interfaces de programação de aplicações (APIs) OO das plataformas de mobilidade para a construção de aplicações. As APIs OO impõem várias restrições arquiteturais aos projetos e isto torna necessária uma arquitetura que permita a incorporação flexível de mobilidade de código nas aplicações. Este trabalho apresenta uma arquitetura de software orientada a aspectos, chamada ArchM, que assegura clara modularização, introdução transparente e aumento de variabilidade da mobilidade de código nos sistemas. ArchM trata das restrições impostas pelas APIs OO e é independente de plataformas e aplicações específicas. A utilidade e a usabilidade de ArchM é medida através de dois estudos de caso e através de sua composição com duas plataformas de mobilidade.

Palavras-chave: Agentes Móveis; Desenvolvimento Orientado a Aspectos; Reuso.

---

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge for publications:

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

Table of Contents

1 Introduction

Using mobile agents, open distributed applications can be developed to run over the Internet in a much more flexible way than before [21, 39]. As a result, code mobility capabilities can be exploited as a means to reduce the complexity of such contemporary applications. However, with mobile agent systems growing in size and complexity, developers are still facing the challenge of achieving enhanced system modularization in the presence of code mobility. The central problem is the invasive and widely-scoped nature of mobility concerns [22-26, 8, 34, 35, 48, 56], hindering the system modularity, composability, and evolvability. It is widely recognized nowadays that these modularity problems occur mainly because the mobility issues cannot be explicitly captured using only object-oriented (OO) abstractions and mechanisms [22-26, 49, 56]. This is primarily due to the fact that the implementation of many of these mobility-specific concerns naturally tends to crosscut other system concerns, such as the basic agent functionalities and coordination activities [22-26, 49, 55, 56]. Despite these modularity breakdowns caused by code mobility, the developers have mostly relied on OO application programming interfaces (APIs) from mobility platforms and on the Java programming language. Another side effect caused by such inefficient modularization is that the introduction of the mobility property into stationary agents is intrusive and error prone.

Modular design support for mobile agents has been studied from different perspectives, including design patterns (e.g. [3]) and mobility frameworks supporting the structuring of code mobility concerns in software agents, such as Aglets [43], JADE [7] and RoleEP/EpsilonJ [56]. Although these frameworks provide OO APIs that offer a number of mobility abstractions and services, they also inherently bring a number of design breakdowns. First, they impose architectural restrictions on the agent design, which are responsible for the tangling and scattering of mobility-specific code over the system. Second, in order to introduce the mobility capabilities into systems, developers must usually modify the agent design to: declare that application agent classes extend specific API classes from mobility platforms, implement the API abstract methods, declare and possibly specify the implementation of the API interfaces, and explicitly invoke the API mobility methods on the system classes which are not created to address mobility concerns. Hence, such implementation strategies result in a high coupling between the underlying models of mobility platforms and the design of mobile agent systems. Third, the direct usage of such APIs does not allow the reuse and explicit handling of the scattered mobility code as variability points in systems implementation. In other words, the variability points cannot be exposed in separate modules where extension or exclusion can be applied. Finally, the lack of proper modularization also makes the composition of the mobility framework with infrastructures addressing other typical concerns in mobile agent system development, such as collaboration and learning, more difficult.

In this context, this work presents an aspect-oriented (AO) software architecture, the ArchM ("Architecting Mobility"), and an example of its implementation, a framework what we refer as AspectM ("Aspectizing Mobility"), to solve the problem of code mobility modularization in mobile agent system development. In the ArchM architecture, architectural aspects [24, 29] are used as unifying abstractions capturing the mobility issues, which are hard to modularize with object-oriented abstractions. More specifically, architectural aspects [24, 29] are used to decouple the mobility concerns from the basic functionalities and other system concerns, including

interaction, adaptation, learning, and autonomy [24]. In the case of the AspectM framework, which has been implemented in AspectJ [38] and is based on the ArchM architecture, the mobility aspects promote this decoupling by defining pointcuts that pick out join points related to the instantiation, migration, initialization, and destruction of mobile agents, and also defining the advice that are associated with these pointcuts. These advice are implemented following a general pattern called the mobility protocol: events are picked out, conditions are checked and the appropriate mobility-specific methods are invoked. For example, the migration advice is responsible for checking the need for the agent roaming and for calling the mobility actions. In summary, the ArchM mobility protocol is defined in such a way that it prevents the explicit invocations of the mobility services by the mobile agent systems and it is independent on specific platforms [3, 7, 43, 56] or applications [5, 22-26].

The usefulness and usability of the ArchM architecture has been assessed in the context of two case studies. The Expert Committee [24] is an open multi-agent system that supports the management of paper submission and reviews for conferences. The MobiGrid [5] is a framework for mobile agent support within a grid environment [30]. The assessment of these case studies is based on architectural metrics rooted at fundamental modularity principles [52], such as separation of concerns, narrow interfaces, low architectural coupling, high component cohesion and composition simplicity. Thanks to these metrics, we have assessed to what extent the ArchM promotes in fact superior modularity in the presence of code mobility. After the ArchM evaluation, we have observed that it allows: (1) a clean separation between the mobility-specific concerns and other agent concerns; (2) a more straightforward introduction of code mobility into software agents; (3) an improved variability of the mobility concerns, such as a flexible choice of the mobility platform.

This work is organized as follows. **Section 2** presents the essential concerns in the development of mobile agent systems. **Section 3** provides a systematic analysis of the modularity problems caused by the crosscutting nature of code mobility concerns in terms of a case study. **Section 4** recalls the fundamental concepts and definitions of aspect-oriented software development. **Section 5** offers an overview of the ArchM architecture, while **Section 6** presents the detailed issues of an ArchM implementation, the AspectM framework. The case studies used to evaluate the ArchM architecture are described in **Section 7** and the evaluation results are detailed in **Section 8**. **Section 9** overviews the related work, comparing the ArchM/AspectM with existing proposals, such as the RoleEP/EpsilonJ [56] framework. Finally, **Section 10** presents the concluding remarks and the future work.

2 Mobile Multi-Agent Systems

2.1 Basic Concepts

Typical mobile agent systems (MAS) consist of a mobility platform and the mobile agents [33, 58] instantiated on this platform. The ***platform*** defines the mechanisms that support the mobile agent execution, and, in general, provides a framework for MAS programming. A ***mobile agent*** consists of code and data. ***Code*** is the program that implements the agent behavior, which is often derived from the framework provided by the platform in use. ***Data*** are the values of the internal attributes modified during agent execution, which can be resulted from an agent internal computation and/or

derived from the platform runtime. A mobile agent can move in a distributed system, from one host to another, carrying its data and code.

Figure 1 presents the physical architecture associated with mobile agent execution: servers, execution contexts, and the underlying network infrastructure. In MAS development, some agents are designed to be mobile, but others are designed to be stationary. A mobile agent only moves to the hosts where a program called agent server is installed. The ***agent server*** enables the agent migration, provides the context to agent execution, and allows communication between agents. The agent migration is supported by the negotiation of the local server with other servers. When the agent migration is required, the agent execution is stopped, the agent is transferred to a remote server and, upon the mobile code arrival, the agent execution is resumed at the remote location. In each location, the agent server provides a ***context***, that is, a complete environment designed for the concurrent execution of mobile agents. Thanks to contexts, agents also can communicate with each other through message exchanges.

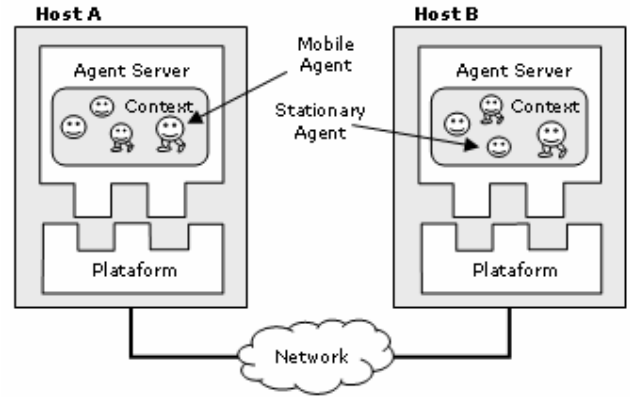

Figure 1. Generic Structure of Mobile Agent Systems

2.2 The Mobile Agent Lifecycle

The mobile agent life is modeled in different stages, which we have termed ***lifecycle model***. The model stages are instantiation, initialization, migration, and destruction of mobile agents. **Figure 2** shows the mobile agent lifecycle model. Different protocols can be used to implement each lifecycle stage. The ***instantiation*** of the mobile agent is made only once when it is created. Every agent receives a unique id and an initial state. ***Initialization*** is performed each time the agent arrives at a new host. ***Destruction*** means that the agent terminates all its activities and frees all the resources it was using. ***Migration*** represents a transfer of an agent from one host to another. The instantiation, initialization, migration and destruction define what we will refer as the ***agent mobility protocol***. An application developer defines the circumstance when a mobile agent must be created; we will refer to this circumstance as the ***instantiation point***. On the other hand, the developers also define the instants where the migration action must be carried out; we will refer to them as ***migration points***.

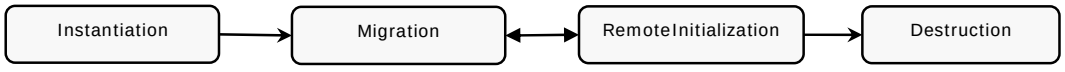

Figure 2. The Mobile Agent Lifecycle

Procedures executed by an agent immediately before the departure to remote environments are called ***departure procedures***. Sending departure messages and blocking processes are some examples of departure procedures. Procedures executed immediately after arrival in remote environments are called ***arrival procedures***. Sending arrival messages and starting processes are example of arrival procedures. In general, departure and arrival procedures data are related to execution contexts. Such data are called ***itinerary*** throughout this paper. Itinerary includes identifiers for reachable hosts and visited hosts, and is implemented either statically or dynamically depending on the restrictions imposed by servers. Itinerary maintenance is particularly important because an agent must be context-aware; that is, an agent has to access the definition of the objects that are locally reachable, the agent neighbors, the agent masters, etc.

2.3 Mobility Strategies and Platforms

MAS may follow strong or weak mobility [21], depending on whether the agent state can or cannot be migrated. On one hand, there are MAS that support the migration of the execution state (***strong mobility***), and on the other hand, those that only support program code and instance data are moved (***weak mobility***). This work focuses on weak mobility, since most mobility frameworks support this form of mobility [21, 39]. In general, the goal of these frameworks is to provide an infrastructure and associated libraries for the MAS development, beyond the functionalities independent on specific MAS applications, such as message transport, encoding and parsing or agent lifecycle. For the application-dependent functionalities, the mobility frameworks allow the specification of agent types, instantiation and migration points, and other mobility protocol issues. To achieve that, framework users may use the MAS fundamental abstractions. For example, Aglets [43] and JADE [7] are Java-based mobility platforms for MAS development, including abstractions such as ***agents***, ***agent ids***, and ***agent contexts***. However, Aglets also introduces an ***agent proxy*** abstraction to deal with context and messaging issues; on the other hand, JADE supports MAS development in compliance with FIPA specifications [20].

Despite these differences, Aglets and JADE APIs implement agents as Java threads running in contexts. In JADE, the basic class for agent instantiation is the `JadeAgent` class; in Aglets, the `Aglet` class is used. In Aglets, a context is implemented through the `AgletContext` interface; in JADE, the context concept corresponds to an `AgentContainer` instance. The `AgletContext` as well as the `AgentContainer` instances consist of a complete environment designed for the concurrent execution of mobile agents. Even though the "Aglet context" and "JADE container" are not equivalent, there are a number of similarities with respect to the issues we have dealt in this work. For example, JADE and Aglets API methods correspond: (1) `onCreation()` and `setup()`, (2) `dispose()` and `doDelete()`, (3) `dispatch()` and `doMove()`; these methods implement the same functionalities in both platforms.

3 Modularization Of Code Mobility

The literature has pointed out that code mobility is often a widely-scoped property that crosscuts the modules implementing other system concerns, such as the basic agent functionalities and coordination activities [22-26, 55, 56]. Although OO frameworks (**Section 2**) are essential to the development of mobile agents, they impose architectural restrictions on MAS design. These architectural restrictions basically can be viewed from three perspectives: they lead to (1) a poor modularization, which decreases MAS

reusability and evolvability, (2) an inefficient introduction of the code mobility into stationary agents, and (3) a difficult composition of MAS with other infrastructures than the mobility platforms. To understand the architectural restrictions above, this section illustrates more fine-grained design problems relative to code mobility in terms of the solution using the JADE framework [7]. We analyze two different approaches for implementation of mobility issues in MAS: direct use of OO APIs from mobility platforms, and the RoleEP/EpsilonJ [56] framework. The advantages and disadvantages of each approach are pointed out through a case study, which will be also used throughout this paper to show the applicability of our proposal for separation of crosscutting mobility concerns.

3.1 A Case Study

Expert Committee (EC) is an open multi-agent system that supports the management of paper submission and reviews for conferences, a classical example of an application based on mobile agents [16, 59]. The EC encompasses two agent types: information and user agents. Each agent type provides different services. For simplicity purposes, this section focuses on the description of the user agents. The basic functionality of the user agents is to infer and keep information about the corresponding users related to their research interests and their participation in conferences. In addition to their basic functionality, user agents can collaborate with each other; the collaboration concern is represented by the roles played by the agents. Each role represents collaborative activities in specific contexts. Different roles are attributed to each EC agent, but the main ones are paper author, reviewer, PC member, and chair. Since these roles need to communicate with each other in the reviewing process, user agents play them in order to cooperate with each other. Classes are used to represent the basic functionalities of the agent types and the different roles.

Each role is associated with a set of plans, which are used to implement more sophisticated collaborative activities; plans are represented by separate classes. The chair role has plans for distributing review proposals; the reviewer and PC member roles have plans for evaluating the chair proposals. The chair, PC members and reviewers negotiate with each other for performing reviews. There are other plans to address user workloads and invitations to new reviewers. **Figure 3** presents some classes representing EC agent types (`ResearcherAgent`, `InformationAgent`), roles (`Reviewer`, `Chair`), and plans (`DistributionPlan`). The chair role is associated with a plan for distributing review proposals.

EC agents need to move in some circumstances, including when they are playing a specific role. For example, when a user agent is playing the chair role, it needs to consult the reviewer profiles in order to optimize the paper distribution in terms of the research interests of each reviewer. If a reviewer profile is not available, it collaborates with an information agent and requests this agent to search for information of the specified reviewer. The information agent controls the local database and is able to query for the profile. However, if the information is not available in the database, the chair role needs to move and try to find the missing profile in remote environments. The agent assigns the searching task to the information agents dispersed over the Internet by roaming through hosts.

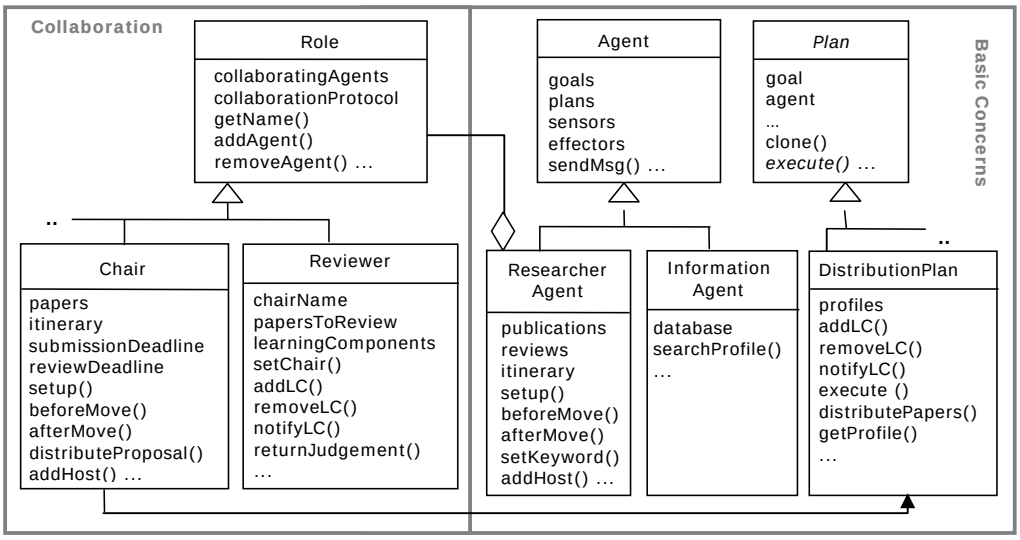

Figure 3. An Object-Oriented Design for the EC Agents and Roles

3.2 Using OO APIs from Mobility Platforms

The use of OO APIs provided by existing mobility platforms requires the invasive implementation of mobility issues. In order to make their agents mobile, developers usually have to change their agents' design to extend mobility-specific classes, implement abstract methods of those mobility classes, implement mobility-specific interfaces (e.g. serialization interface), and invoke explicitly mobility methods in the classes implementing other agent concerns. **Figure 4** presents the OO design of the EC mobile agents and their roles using the JADE mobility framework. For simplification, it only shows some important classes, the others essentially follow the same pattern; we also omit the classes related to learning, adaptation, and autonomy. The main purpose of each set of classes, surrounded by a gray rectangle, is to modularize a specific agent concern, namely interaction, collaboration, mobility, and basic concerns. However, note that the mobility concerns crosscut classes implementing other agent concerns; it has a huge impact on the basic agent structure, and on the collaboration and interaction designs. Although part of the mobility concern is localized in the mobility classes, such as `JADEAgent` and `Itinerary`, mobility-specific code replicates and spreads across several class hierarchies of an agent.

Figure 4 shows each crosscutting-related problem with a number surrounded by a circle. There are classes that represent the agent types and roles, extending the abstract `JADEAgent` class to incorporate the mobility capabilities. However, the use of inheritance results in code replication as well as in both code tangling and scattering; the basic functionality and collaborative activities are amalgamated to mobility methods (problem ①). The agent and role classes also need to hold an explicit reference to mobility elements (e.g. `itinerary`) as attributes (problem ②). These classes have additional methods to manage these elements (problem ③). In addition, several methods contain mobility code in order to define the agent migration points, w.r.t. the decisions on when the agent should move to a remote environment (problem ④), and when going back to the home location (problem ⑤). As a result, this code is replicated in various methods of plan, role, and agent type classes.

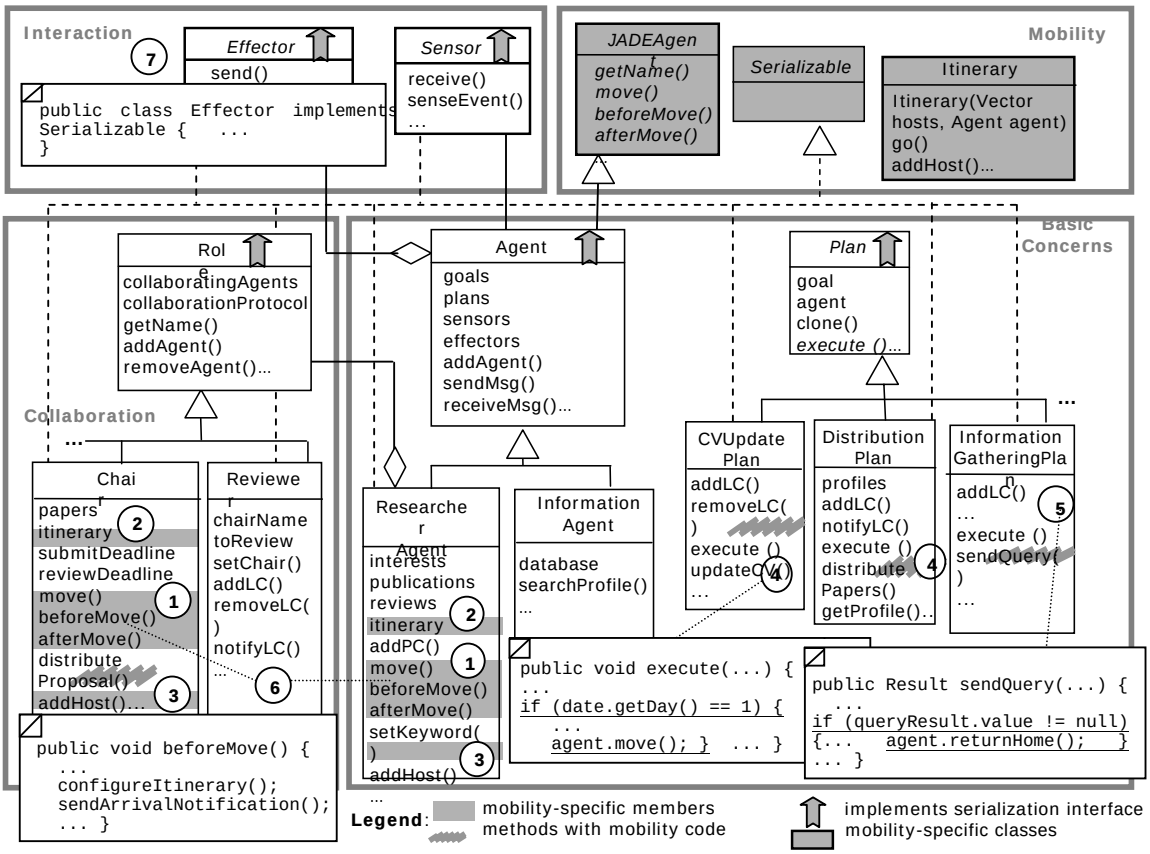

Figure 4. JADE-Based Design for the Expert Committee System

Moreover, there can be a spread of usual preconditions and postconditions when an agent moves to another host (problem ⑥). Such conditions either can be dependent or independent from the application agents. Finally, several classes have to implement the `Serializable` interface for allowing the objects, which are part of the agent, to be moved across hosts (problem ⑦). All these problems decrease the system reusability and evolvability, since adding or removing the mobility code from classes requires invasive changes in those classes. Note that we cannot find a more modular solution even if we try refactoring of the OO solution presented in **Figure 4** or use another mobility framework. This problem occurs because mobility is a crosscutting concern independent on the OO decomposition used [22-26, 56].

With respect to problem ⑦, the `Serializable` interface is just a representative example of the following scenario: OO APIs from platforms usually provide a number of interfaces with methods that are implemented by systems in order to make plans and actions automatically be executed at specific moments of the mobile agent lifecycle; for instance, plans and actions can be automatically executed immediately after the agent instantiation or immediately before the agent migration. In this way, mobility-specific code spreads across several plan methods, roles and agent types.

Beyond the architectural restrictions on the agent design, the pervasiveness of existing mobility frameworks also introduces conceptual problems. The `Agent` class has to extend the `JADEAgent` class even when some agent types are stationary. For example, it is not possible to directly define the `UserAgent` class (**Figure 4**) as a `JADEAgent` subclass since the former already extends the `Agent` class. The same problem happens when specifying a specific role as mobile (e.g. the `Chair` class in

Figure 4). Moreover, the use of OO frameworks can lead to potential conflicts. For example, the `JADEAgent` class defines an abstract method `getName()`; the `Agent` class also has a method `getName()` with a different purpose. The introduction of mobility in this system causes implementation clashes, and requires the renaming of this method and changes in the respective callers.

3.3 RoleEP/EpsilonJ

RoleEP [56] is an approach that addresses the problem of constructing MAS with a mechanism for separating the mobility and collaboration concerns. To do that, RoleEP proposes specific abstractions, such as agents, roles, objects and environments. An ***environment*** is composed of attributes, methods, and roles. A ***role***, which can move between hosts that exist in an environment, is composed of attributes, methods and binding-interfaces. Role attributes and methods are only available in an environment to which the role belongs. A ***binding-interface***, which is similar to an abstract method interface, is used when an object binds itself to a role. Common data and functions that are used in roles are described by environment attributes and methods.

An ***object***, which cannot move between hosts, is composed of attributes and methods. Although an object cannot move between hosts, it can move by binding itself to a role that has mobility functions. An object becomes an agent by binding itself to a role that belongs to an environment, and can collaborate with other agents within the environment. The notion of the ***binding-operation*** binds binding-interfaces of roles to concrete methods of objects. The binding-interface defines the interface in order to receive messages from other roles existing in the same environment. The binding-operation is permitted only when an object has methods corresponding to the binding-interface. Binding-operations are implemented by creating delegational relations between roles and objects dynamically. That is, if a role receives a message corresponding to its binding-interface from other roles or itself, the role delegates the message to an object bound to the role.

Regarding this point, it is important to note that RoleEP approach is based on its own specific concepts and imposes a number of restrictions in the application design. In other words, using this approach, MAS cannot be constructed only with objects; it is necessary to use RoleEP-specific concepts, such as environment and role. In fact, RoleEP concepts are realized by the MAS instantiated from EpsilonJ framework [56]. In ***EpsilonJ***, an environment class is defined as a subclass of the `Environment` class, and a role class is defined as a subclass of the `Role` class. The `Role` class is implemented as a subclass of the `Aglets` class that has mobility functions. A class of an EpsilonJ's object is defined as a subclass of the `EpsilonObj` class that presents functions for binding-operations. **Figure 5** illustrates the partial result of the EC development process using mobility services through the EpsilonJ framework.

As we can observe, developers need to extend a number of EpsilonJ classes in order to instantiate their MAS. As we have demonstrated in **Section 3.2**, a framework instantiation process using only OO abstractions and mechanisms hinders a satisfactory modularization of mobility issues. Moreover, for MAS development using EpsilonJ, we have implemented drastic changes on the application's original design. For instance, for `Chair` role class access the Aglets mobility services (and not the JADE ones as before) we make a number of changes in the class inheritance trees of the EC system (**Figure 5**). In **Section 6**, we show that in order to instantiate an application from AspectM, it is not necessary to extend any classes from framework, while we also maintain the mobility platform flexibility.

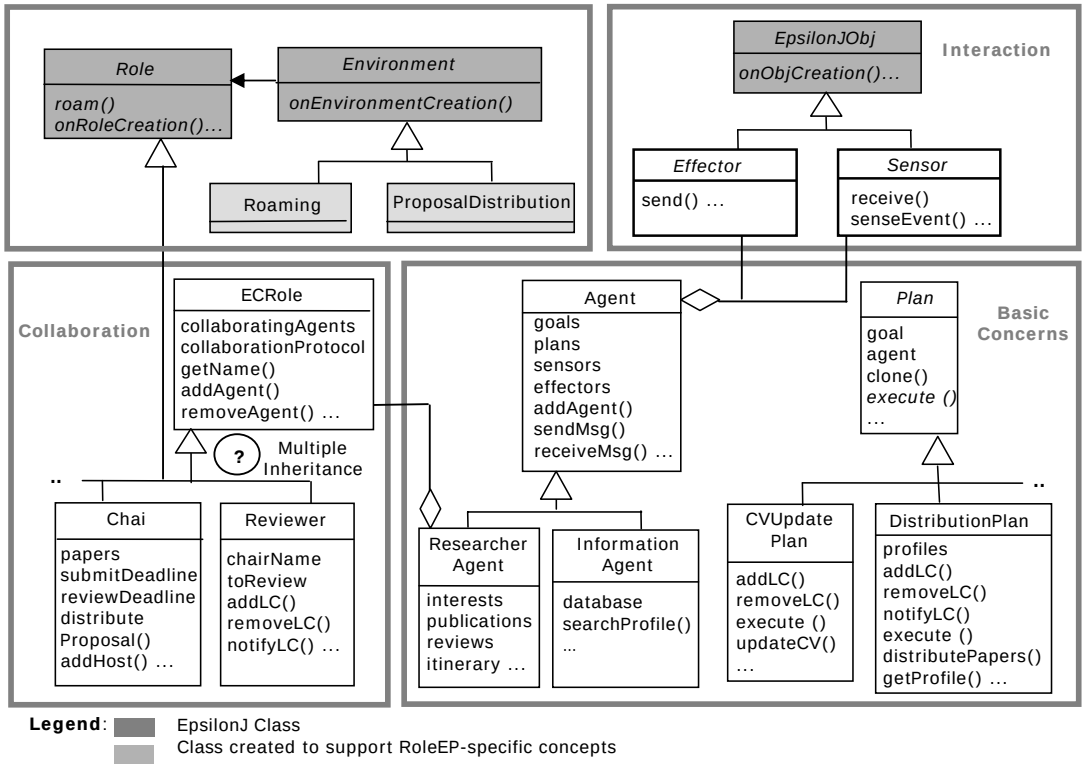

Figure 5. RoleEP-Based Design for Expert Committee System

4 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) is an emerging area with a goal of promoting advanced separation of concerns throughout the software development lifecycle. This section introduces the AOSD concepts and modeling notations. **Section 4.1** presents AOP definitions and **Section 4.2** shows the AO modeling notations for architectural and detailed design.

4.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [37] is an emerging programming paradigm with the goal of improving separation of crosscutting concerns at the implementation level through new abstractions and composition mechanisms. ***AspectJ*** [38] is the most widely used AO programming language and, therefore, the most representative of a family of AOP languages, such as JBoss AOP, Spring Framework, etc. Most case studies found in the literature explore AOP in AspectJ [38] in the context of different crosscutting concerns, such as exception handling [19], persistence and distribution [50, 54], and design patterns [28]. However, AOP has not been fully explored an improved end-to-end modularization of code mobility since an early version of the design. The main abstractions supported by AspectJ are: (1) aspects, (2) join points, (3) pointcuts, (4) advice, and (5) inter-type declarations. ***Aspect*** is the abstraction to support improved modularity of crosscutting concerns. An aspect can crosscut one or more classes, changing their structure or dynamics. ***Join points*** are well-defined points in the dynamic execution of a system which are used to specify how classes and aspects are related. A collection of join points can be specified through a ***pointcut***.

Advice is a special method-like construct attached to pointcuts, which defines a crosscutting feature to affect the dynamic behavior of classes. An advice can run before, after or around whenever a join point is reached. An aspect can also contain internal attributes, methods and inter-type declarations. The ***inter-type declarations*** specify new members (attributes or methods) for classes to which the aspect is attached, or change the inheritance relationship between classes. Unlike advice, which operates primarily dynamically, inter-type declarations operate statically, at compile-time. Inter-type declarations are static crosscutting features since they affect the static structure of components. Aspects can be defined as abstract and extended by concrete aspects; both methods and pointcuts can be qualified as abstract.

4.2 Aspect-Oriented Architecture and Detailed Design

Aspects can be represented not only at the implementation, but also at the architectural and detailed design level [4, 6, 13, 29, 30]. In fact, aspects have always a broadly-scoped impact at the design decomposition and encompass driving architectural concerns [30]. ***Software architecture*** is a high-level description of the system organization in terms of architectural components, their interrelationships and responsibilities [49]. Each ***component*** conforms to and provides the realization of a set of ***interfaces***, which make available services implemented by the component.

The notion of an aspect-oriented software architecture introduces the concept of an aspectual component (or architectural aspect) [24, 29]. An ***aspectual component*** modularizes a crosscutting concern at the architectural level. Each of the aspectual components is related to more than one architectural component, representing their crosscutting nature. The relationships are associated with the component interfaces, which are classified in conventional (normal) or crosscutting interfaces. A ***conventional interface*** only provides services to other components. ***Crosscutting interfaces*** provide services to the system, but also specify when and how an architectural aspect affects other architectural components.

Figure 6 illustrates an aspect-oriented architecture for the aspect called `FaultHandler`. The architecture modeling is based on the AOGA notation [12, 24, 40-41], which extends the aSideML language [11]. These languages are used throughout this paper: the aSideML language extends UML with semantics and visual notations for representing aspects at the detailed design level. The AOGA notation suppresses all information about aspect internal elements and adds notation to represent architectural aspects. Aspects are represented as diamonds, while a crosscutting interface is displayed as a small grey circle with its name placed next to the circle.

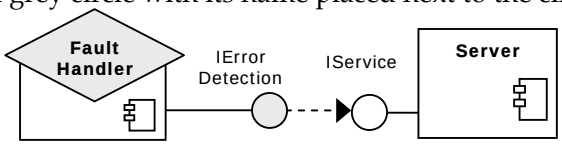

Figure 6. An Aspect-Oriented Architecture for the FaultHandler Aspect

Figure 7 illustrates the detailed design of the `FaultHandler` aspect using the aSideML language [11]. In this language, an aspect is composed of internal structure and crosscutting interfaces. The ***internal structure*** declares the internal attributes and methods. A ***crosscutting interface*** specifies when and how the aspect affects one or more classes. Each crosscutting interface is composed of inter-type declarations, pointcuts and advice, and is represented using a rectangle symbol with compartments. The first compartment represents inter-type declarations, and the second compartment

represents pointcuts and their attached advice. The notation uses a dashed arrow to represent the ***crosscutting relationship***. Note that, at the detailed design level, an architectural component is realized as a set of aspects and auxiliary classes; the conventional components are refined and implemented as a set of classes.

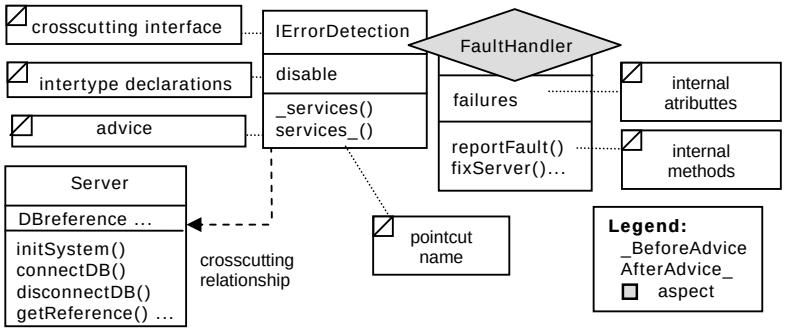

Figure 7. An Aspect-Oriented Design for the FaultHandler Aspect

The internal structure of the `FaultHandler` aspect consists of two methods and one attribute. They were moved from the `Server` class to the aspect since they are part of the error-handling concern. The `IErrorDetection` crosscutting interface (**Figure 7**) declares how the `FaultHandler` aspect crosscuts the `Server` class. This interface introduces the disabled attribute on `Server`, and two pieces of advice. There is one before advice and one after advice, both of them associated with the same pointcut named `services`. Note that the `FaultHandler` aspect modularizes the error-handling concern, and the `Server` class contains no exception-handling code. **Figure 8** presents interaction diagrams that illustrate when the `FaultHandler` aspect dynamically affects the `Server` class. The join points are the catching of exceptions `FaultException` and calls to the `getComplaints()` method.

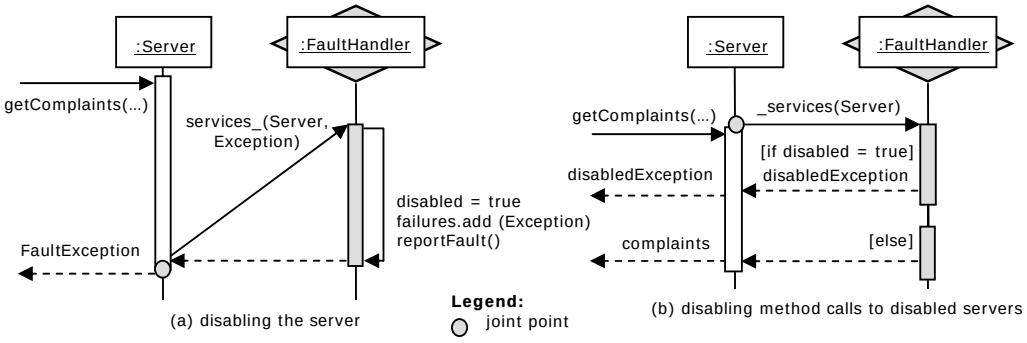

Figure 8. The Dynamics of the Server Class and the FaultHandler Aspect

5 The ArchM Software Architecture

This section presents ArchM, an aspect-oriented software architecture (**Section 4.2**) for addressing the mobility tangling and scattering (**Section 3.2**) often found in MAS design and implementation artifacts. We use the AOGA notation (**Section 4.2**) in **Figure 9** for presenting the ArchM architecture.

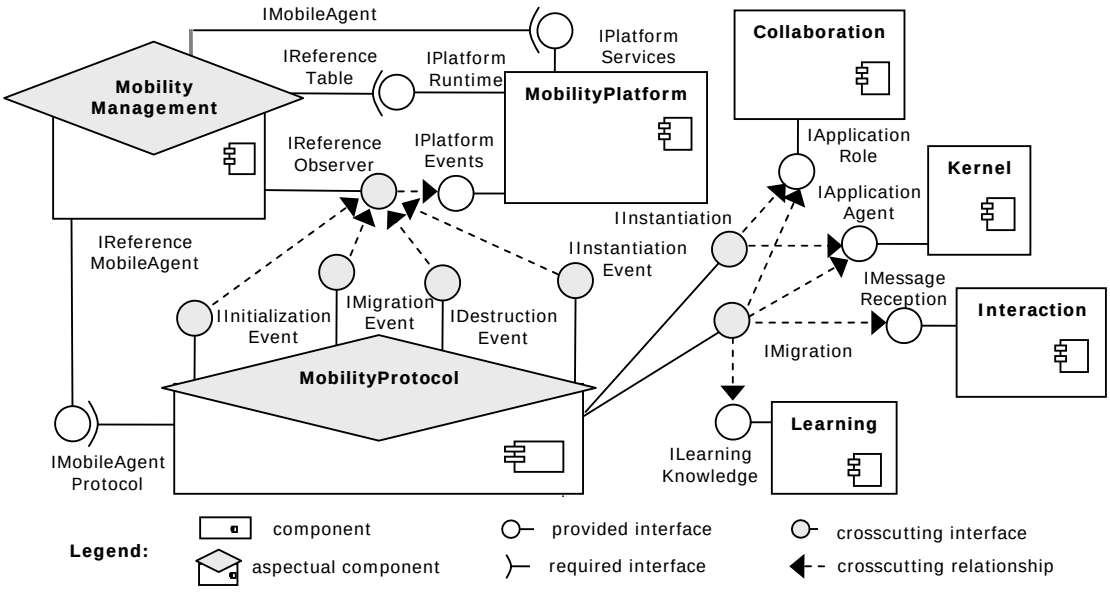

Figure 9. The ArchM Architecture

In **Figure 9**, the separation of the mobility concerns and the integration between mobile agent applications and distinct mobility platforms respectively resulted in the conception of two architectural aspects (**Section 4.2**): `MobilityProtocol` and `MobilityManagement`. **Section 5.3** explains how these components (**Section 5.1**) and interfaces (**Section 5.2**) interact with each other in order to solve the architectural restrictions in MAS. Solving these restrictions, ArchM ensures: (1) a clean modularization of the code mobility, (2) a more straightforward introduction of code mobility into stationary agents, and (3) an improved variability of the mobility concerns, such as a flexible choice of the platform.

5.1 Components

The ArchM architecture is composed of five kinds of components:

1. the `Kernel` component, which modularizes the basic concerns of an agent-based application;
2. the `MobilityProtocol` component, which modularizes the mobility protocol execution (**Section 2.2**), i.e. the instantiation, migration, remote initialization, and destruction of agents in applications;
3. the `MobilityManagement` component, which provides a flexible integration between MAS and distinct mobility platforms (**Section 2.3**);
4. other components, which represent additional agent concerns, such as collaboration and learning;
5. the `MobilityPlatform`, which represents a specific mobility platform being used, such as JADE [7] and Aglets [43] (**Section 2.3**).

The component `Kernel` modularizes the basic functionality associated with an agent type. It is also responsible for modularizing the elements of the agent's intrinsic knowledge. Alternatively, it can represent an existing object, which needs to be transformed into an agent. The `Kernel` component realizes the interface that makes services available for the agent's clients, the `IServices` interface (**Figure 9**).

The MobilityProtocol and MobilityManagement components separate together the mobility concern from Kernel and other components. In this way, the MobilityProtocol and MobilityManagement components isolate mobility concern from the agent basic functionality and intrinsic knowledge (Kernel component), as well from the other agent concerns (other aspectual and non-aspectual components). Particularly, the MobilityProtocol component isolates agent mobility protocols; that is, the MobilityProtocol component isolates crosscutting concerns referring to instantiation, initialization, migration and destruction protocols (**Section 2.2**). On the other hand, the MobilityManagement component connects MAS with the MobilityPlatform component, which modularizes the platform services. It allows a flexible integration of MAS with different platforms.

Other architectural components are used to improve the separation of specific crosscutting concerns of MAS, including interaction [24], adaptation [24], learning [24, 27], autonomy [24, 53], coordination [2, 36, 55, 56], context-awareness [47], error handling [15, 18-19], distribution, persistence and concurrency [50, 54] and design patterns [9, 28]. In other words, the additional components modularize agent properties, such as autonomy and adaptation, in such a way the agent properties are isolated from the agent kernel and from each other. In **Figure 9**, components that modularize additional agent concerns are represented by interaction, collaboration and learning components. These components also implement a number of normal interfaces; **Figure 9** omits them for simplification purposes.

The MobilityPlatform component represents a specific mobility platform being used. The goal of a mobility platform is to provide an infrastructure and associated libraries for the MAS development (**Section 2.3**). In general, mobility frameworks provided by platforms are object-oriented and allow the specification of a number of mobility protocol issues. However, the direct use of OO mobility frameworks APIs requires the invasive implementation of mobility issues (**Section 3.2**). In the ArchM architecture, the MobilityPlatform component is completely separated from the other agent concerns.

5.2 Interfaces

The MobilityManagement component realizes four interfaces (**Figure 9**). The IMobileAgent normal interface determines recurrent mobile agent services on platforms. This interface allows to access to mobile agent services provided by mobility platforms through the IReferenceMobileAgent interface. The crosscutting nature of the MobilityManagement component is specified through the IReferenceObserver interface (**Figure 9**). This interface crosscuts join points as calls and/or executions of mobility platform services. For example, the IReferenceObserver interface provides access to service execution of a mobile agent lifecycle, such as the destruction of a mobile agent in its platform. The MobilityManagement component also realizes a conventional interface called IReferenceTable, which is used to abstract the context and messaging services provided by different platforms.

The MobilityProtocol component is related to more than one architectural component, representing its crosscutting nature (**Figure 9**). The IInstantiation crosscutting interface determines when and how a mobile agent is instantiated on a platform to represent a specific agent on an application. For this reason, the IInstantiation interface crosscuts agent types and roles localized in the Kernel and Collaboration components, respectively (**Figure 9**). Thanks to the

IInstantiation interface, we maintain a univocal association between an application agent or role instance and its respective mobile agent instance on the platform in use. The specification of the IInstantiation interface corresponds to the specification of an agent instantiation point in MAS (**Section 2.2**).

The IMigration defines the components that trigger the agent travel to remote environments and the agent return to the home host. This interface crosscuts the IApplicationAgent interface since the mobile agent may have to travel whenever elements of the agent kernel change or before/after the execution of a service (**Figure 9**). In addition, the IMigration interface also crosscuts other elements (**Figure 9**): (1) the Interaction component, because travels are also motivated by external events or by messages received from other agents; (2) the Collaboration component, since travels can be triggered due to some actions performed in the context of an agent role; and (3) other components (e. g. the LearningKnowledge component), depending on the agent type/role features. The IMigration interface corresponds to the specification of an agent migration point (**Section 2.2**).

Beyond the IInstantiation and IMigration interfaces, the MobilityProtocol component realizes other interfaces, which crosscut the IReferenceObserver interface of the MobilityManagement component. For example, the InstantiationEvent and MigrationEvent interfaces detect the creation and the migration of a mobile agent. In addition, the IInitializationEvent interface detects the agent arrival in a new host, and the IDestructionEvent interface detects the destruction of a mobile agent in the platform. The access to the mobile agent lifecycle makes the definition of a generic mobility protocol possible, that is, a mobility protocol (**Section 2.2**) without any references to a specific mobility platform. Its purpose is to decrease coupling between agent architectural elements and platform models.

5.3 Architectural Solutions

The ArchM architecture uses architectural aspects and crosscutting interfaces (**Section 4.2**) to make a clean modularization of the code mobility possible in MAS. The MobilityProtocol component implements a generic mobility protocol in order to prevent the explicit invocations of the mobility services by the other components. To achieve that, the IInstantiation crosscutting interface is used to determine when and how a mobile agent is instantiated to represent a specific agent in an application. In addition, the IMigration interface is used to affect well-defined mobility join points in order to determine when agents should move.

Other interfaces are used in the ArchM architecture in order to maintain a flexible integration between Kernel and the MobilityPlatform components; they make information relative to the mobile agent lifecycle available to the other components. The IReferenceObserver interface crosscuts join points as calls of mobility platform services. In turn, the InstantiationEvent, MigrationEvent, IInitializationEvent, and IDestructionEvent interfaces also crosscut the IReferenceObserver interface to allow the MobilityProtocol component to specify a generic mobility protocol.

Finally, the IMobileAgentProtocol and IReferenceMobileAgent interfaces play a central role in the ArchM architecture. The IMobileAgentProtocol is the interface that delegates to the IReferenceMobileAgent the mobility services invoked by the Kernel component; this delegation is independent on platform-specific issues. The IMobileAgent is the interface that is responsible for delegating to

a specific platform the invoked services; to achieve that, it uses the `IPlatformServices` interface provided by the `MobilityPlatform` component. To introduce code mobility in MAS, application users then must only perform the specification of agent instantiation and migration points.

6 The AspectM Detailed Design

In this section, we present our ArchM software architecture (**Section 5**) implementation, the AspectM framework. This framework provides solutions to more fine-grained problems related to the (1) modularization of code mobility (**Section 6.1.1**), (2) introduction of code mobility into agents (**Section 6.1.2**), and (3) integration of MAS with mobility platforms (**Section 6.1.3**). In other words, we present solutions to the mobility-specific tangling and scattering problems often found in the detailed design and implementation levels of MAS development. AspectM contains more than 30 abstract classes and aspects. However, we focus on the description of the AspectM main elements in order to show how the framework overcomes the problems described in **Section 3**, and how they are instantiated to a specific application (**Section 6.3**). The dynamics of these elements is also presented in **Section 6.2**.

6.1 The AspectM Framework Structure

6.1.1 Improved Modularization of Mobility Concerns

Figure 10 illustrates the detailed design of the AspectM `MobilityProtocol` component (**Section 5.1**). The modeling is based on the aSideML language (**Section 4.2**); we have enhanced the notation to explicitly distinguish the hot spots (variable parts), which are marked with a star, from the frozen parts of the framework. At the detailed design level, the `MobilityProtocol` component is implemented as the abstract `Mobility` aspect and its subaspects (`ChairMobility` aspect in **Figure 10**). The abstract `Event` aspect also appears in **Figure 10**, but it does not belong to the `MobilityProtocol` component; it is used to introduce the existing relationship between the `MobilityProtocol` and the `MobilityManagement` components. The `MobilityManagement` component will be presented in detail in **Section 6.1.3**. The Expert Committee classes are included in **Figure 10**, but similar to the `Event` aspect, these classes do not belong to the `MobilityProtocol` component. They are part of the `Kernel` (e. g. `ResearcherAgent` class) or additional components (e. g. `Role` class belongs to the `Collaboration` component).

The purpose of the `Mobility` aspects is to decouple the mobility concerns from the basic functionalities and other system concerns. The abstract `Mobility` aspect promotes this decouple by defining pointcuts that pick out join points related to the instantiation, migration, initialization, and destruction of mobile agents. It also contains the advice that are associated with these pointcuts. These advice are implemented following a general pattern: events are picked out, conditions are checked, and the appropriate mobility-specific methods are invoked. In other words, the `Mobility` advice are the AspectM frozen spots. For example, the migration advice is responsible for checking the need for the agent roaming and for calling the mobility actions in an abstract way. Thus, the `Mobility` advice jointly correspond to the agent mobility protocol (**Section 2**).

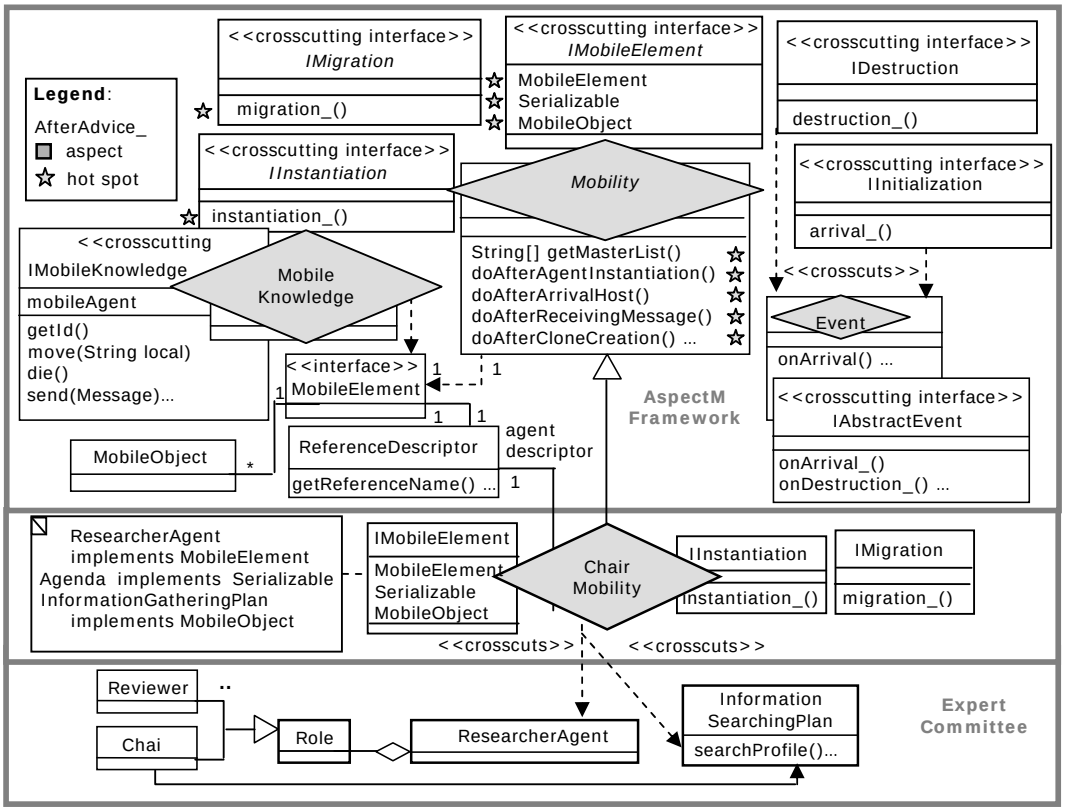

Figure 10. AspectM Solution for Expert Committee

Note that the `doAfterAgentInstantiation()` and `doAfterArrivalHost()` methods in the `ChairMobility` (**Figure 10**) aspect correspond to the return of the control flow back to the EC-specific procedures. Conversely, it does not occur in migration and destruction protocols, where control flow is not deviated back to the EC system. Note also that the instantiation and migration pointcuts are application dependent; they run after join points such as those from a `Plan` subclass (**Figure 10**); we need to specify in `Mobility` subaspects the elements affected by the `instantiation_()` and `migration_()` advice. The initialization and destruction pointcuts are directly detected from platforms.

In addition to the `Mobility` and `Event` aspects, AspectM implements the `MobileElement` interface, used by the `Mobility` aspects to delegate the mobility protocol actions to a specific platform. More specifically, this interface defines mobile agent-specific services abstracted from the distinct platforms, and is responsible for delegating to a specific mobility platform the services provided by its interface. These services, usually important to the MAS developers, include: (1) methods that grant access to ids (`getName()`, `getId()`, `getContextId()`, `getMessageId()`, etc.); (2) methods implementing the mobile agent lifecycle (`move()`, `clone()` and `die()`, etc.); and (3) methods for agent messaging (`send()`, `sendAsync()`, etc.). Thus, the `MobileElement` interface implements the methods that usually crosscut the OO design of software agents. An inter-type declaration is used to specify if an application class implements the `MobileElement` interface. Additional declarations are used to define which objects implement the `Serializable` interface and those that implement the `MobileObject` interface. These declarations represent the specification of the `IMobileElement` crosscutting interface in **Figure 10**.

Finally, note that the `Mobility` aspect holds a reference to the OO mobility platform and encapsulates the mobility protocol interactions with such a specific platform (see **Section 6.1.3**). **Figure 10** shows that all the mobility code is localized in the `Mobility` aspects. As a result, the agent and role classes are not intermingled with mobility code, therefore improving their modularity, reusability and changeability. In fact, the `Mobility` aspects modularize the crosscutting concerns presented in **Section 3**, thereby solving the problems ①, ②, ③, ④, ⑤, ⑥ and ⑦.

6.1.2 Transparent Introduction of Code Mobility

The AO design in **Figure 10** uses AOP to make an explicit separation of mobility concerns in MAS possible. The mobility tangling and scattering problems are solved in AspectM by combining the following design decisions. We use (1) the `Mobility` aspect for the generic code mobility, (2) a `Mobility` subaspect for each mobile element (role, agent, or plan) in order to prevent the explicit invocation of mobility-specific methods in MAS classes, and (3) the `Mobility` pointcuts to bridge the AspectM framework with MAS.

The use of such an aspect hierarchy inverts the way in which mobility concerns are typically implemented in MAS: OO abstractions and mechanisms, as inheritance and delegation, are replaced by AO abstractions and mechanisms. The latter ones are used to crosscut MAS join points in order to provide the mobility modularization. For example, in order to solve the problem related to explicit inheritance-based extensions involving MAS classes and elements of the OO frameworks (**Section 3**), we have used AO programming languages idioms [32] that allow the use of interfaces as if we were using abstract classes; mobility-specific methods can be called by an agent or role class through the direct use of interfaces while applications can maintain their own agent hierarchies.

Therefore, in order to introduce mobility-specific concerns into a stationary agent, the MAS designers only have to specify the following AspectM hot spots: (1) the elements to be defined as mobile, (2) the instantiation pointcut and methods, (3) the initialization methods, (4) the migration points, (5) the definition of which objects will be moved together with the mobile element, and (6) the definition of the serializable elements. For example, **Figure 10** shows that the `ChairMobility` aspect extends the `Mobility` aspect in order to specify the `Chair` mobility-specific behavior. Concrete implementations of the AspectM hot spots are defined for the context of the `Chair` role: (1) the `ResearcherAgent` type implements the `MobileElement` interface, which allows its instances to become mobile (the `ResearcherAgent` type must be declared as mobile because the `Chair` role depends on this agent type knowledge wherever it is transferred); (2) the `InformationSearchingPlan` type implements the `MobileObject` interface, so that the plan can be moved with its respective agent; (3) the `Agenda` type implements the `Serializable` interface, which allows an `Agenda` object to be moved together with its respective instance; (4) the `ChairItinerary` implements the `Itinerary` class; (5) the `getItineraryType()` and `getContextList()` methods; (6) the instantiation pointcut, which triggers the agent instantiation protocol (`agentInstantiation_()` advice); (7) the execution of the `InformationSearchingPlan searchProfile()` as the migration pointcut; and (8) the initialization EC procedures to be executed in the `doAfterArrivalHost()` method.

Even though it could be necessary to reengineer MAS classes in order to expose the appropriate mobility join points to be affected by the `Mobility` aspects, all we have

described confirms that the AspectM framework in general promotes a seamless introduction of mobility concerns into stationary agents. Introducing mobility concerns into agents corresponds to the user's task of making concrete a `Mobility` aspect for each stationary agent on his design. Otherwise, in order to turn a mobile agent into a stationary one, we just have to remove the concrete `Mobility` subaspect corresponding to this agent.

6.1.3 Integration with Distinct Platforms

Figure 11 presents the design elements used to support a flexible integration between MAS and distinct mobility platforms. These elements implement the `MobilityManagement` component of the ArchM architecture (**Section 5**). A general strategy of maintaining an agent reference table is applied, once mobility platforms present meaningful differences in their implementations for context and messaging services. Thanks to the use of such a table, it is possible to encapsulate contexts and message proxies through the `Mobility` aspects. In particular, we deal specifically with message proxies and formats through the `MessageParser` class, which executes parsing between a platform-specific format and the AspectM format (**Figure 11**). The AspectM format is independent on a particular mobility platform and contains `MessageParser`-specific hot spots: the `parser(Message)` and `parser(Object)` methods (**Figure 11**). In the following figure, we present other design elements containing AspectM hot spots that a mobile agent system developer must define in order to use platform-specific services in a flexible way.

The `ReferenceManager` class is a stationary agent that is a singleton and responsible for (1) the reference table instantiation and its update on each agent instantiation, initialization or destruction, and (2) response to common requests, such as getting the agent list in a specific context. These services correspond to AspectM frozen spots and are implemented through an interface with an abstract class behavior [32]. The AspectM `ReferenceManager`-specific hot spots are the platform-dependent methods, such the `getMessageId()` and `send(Message)` (**Figure 11**).

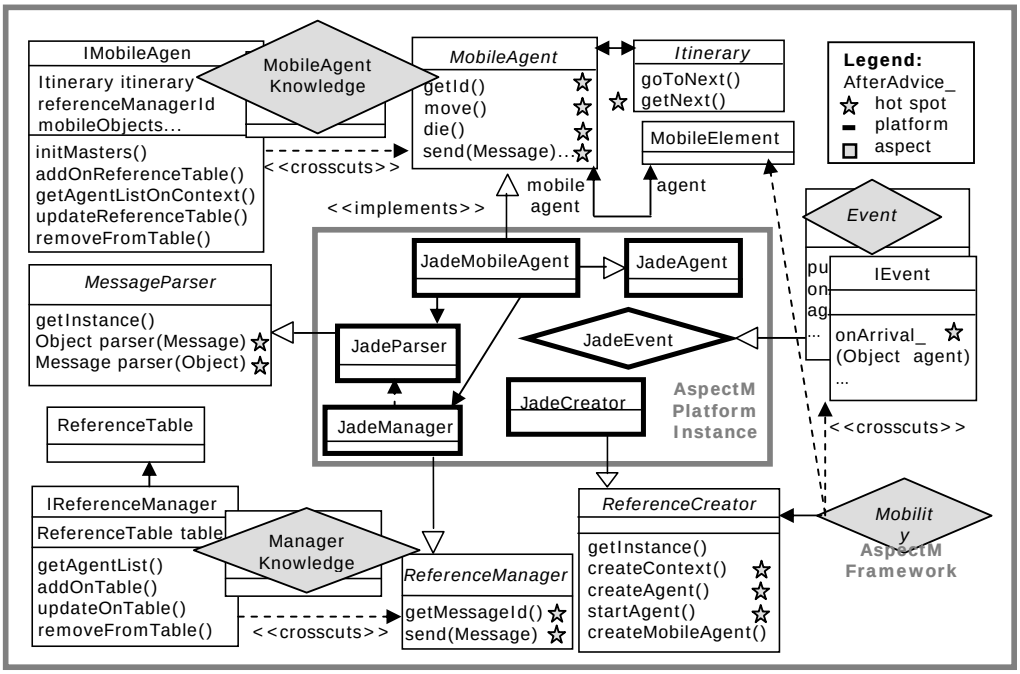

Figure 11. Integrating MAS with the JADE Mobility Platform

The `ReferenceCreator` class is implemented as a singleton for each context to be instantiated and is responsible for (1) the context creation for mobile agent execution at a specific host (`createContext()`), (2) the instantiation of mobile agents on this context (`createAgent()`), (3) the starting of platform services for instantiated agents (`startAgent()`), and (4) the template method for the agent instantiation (`createMobileAgent()`). The responsibilities from (1) to (3) are platform-dependent methods, and, thus, they are AspectM `ReferenceCreator`-specific hot spots. Conversely, the template method for the agent instantiation is an AspectM frozen spot (**Figure 11**).

The `MobileAgent` class defines mobile agent services abstracted from distinct platforms (AspectM `MobileAgent`-specific hot spots). These services include methods, such as: `getName()`, `getId()`, `move()`, `clone()`, `die()`, `send()`, `sendAsync()`, and so forth. In other words, the `MobileAgent` class is responsible for delegating to a specific mobility platform the agent services provided by its interface. This class is also responsible for the communication with the `ReferenceManager` class in order to reply to common requests, such as getting the agent list in a specific context. The communication between a `MobileAgent` instance and the `ReferenceManager` instance is an `MobileAgent`-specific frozen spot that is implemented through an interface with an abstract class behavior [32] (**Figure 11**).

The `Event` aspect allows detection of relevant platform-specific join points, such as the mobile agent initialization and destruction. For example, the `Mobility` aspect crosscuts the `Event` aspect when the `onArrival()` method is executed. This method is invoked on the `onArrival()` after advice. In turn, the `onArrival()` advice is executed immediately after the detection of the initialization pointcut related to a specific platform, which must be concrete in an `Event` subaspect. Note that the body of the `onArrival()` method may not have any code line, since its purpose is only to bridge the specific platform `onArrival()` method (the AspectM `Event`-specific hot spot) with the `Mobility` aspect initialization pointcut. This same detection strategy is used to access other agent lifecycle events, such as destruction, and cloning, and constitutes the AspectM `Event`-specific frozen spot (**Figure 11**). We can reach the same result with OO reflection, but AOP allows the detection of platform-specific events, such as arrival, destruction, and cloning, in a more natural way.

In addition, **Figure 11** shows that the `MobileElement` interface is the element that allows `MobileAgent` objects to represent the mobile elements defined in the `Mobility` aspects. The `MobileElement` class is the central element of the AO design, once it bridges MAS (e. g. EC) with the classes used to integrate MAS with mobility platforms (e. g. JADE). If an AO platform instance has been developed (e. g. Aglets or JADE), such instance can be largely reused in MAS. Finally, note that: (1) the `ReferenceCreator` and the `ReferenceManager` classes implement the `IReferenceMobileAgent` interface of the `MobilityManagement` component; (2) the `MobileAgent` class implements the `IMobileAgent` interface; (3) the `Event` aspect implements the `IReferenceObserver` interface; and (4) the `MessageParser` class is an internal `MobilityManagement` element.

6.2 The AspectM Framework Dynamics

Figure 2 called "The Mobile Agent Lifecycle" illustrates that the existence of a mobile agent can be represented through stages of a model that include recurrent procedures from instantiation, initialization, migration and destruction protocols (**Section 2**). In the AspectM framework, we have abstracted these recurring procedures from the

analysis of the agent mobility protocol (**Section 2**). These procedures are then AspectM frozen spots, which are illustrated in **Figure 12**.

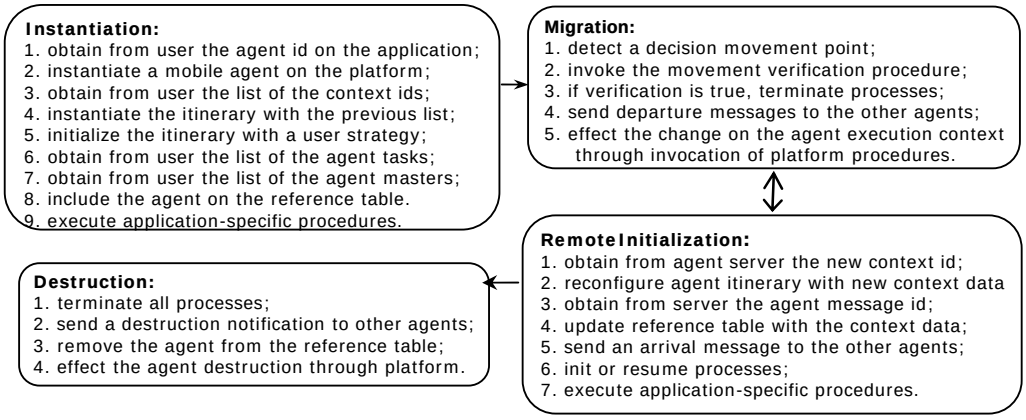

Figure 12. The AspectM Frozen Spots

The instantiation protocol in the context of an application agent and a platform mobile agent instantiated from the respective `UserAgent` and the `PlatformMobileAgent` classes is as follows (**Figure 13**):

1. obtains from user the agent id in the application (`id` argument in the `agentInstantiation` pointcut of the `UserAgentMobility` aspect);
2. obtains from user the agent reference name in the application (`agent` argument in the `agentInstantiation` pointcut of the `UserAgentMobility` aspect);
3. creates a mobile agent in the platform (a "platform mobile agent"), corresponding to the application agent (`createMobileAgent()` method);
4. configures the relationship between the application agent and the platform mobile agent (`setMobileAgent()` method);
5. obtains from user the agent itinerary type (`getItineraryType()` method), the list of itinerary context ids (`getContextList()` method), and other data necessary to the instantiation protocol;
6. creates the agent itinerary with the data obtained during the previous step (`createItinerary()` method);
7. obtains from platform mobile agent the original context id (`getLocalContextId()` method);
8. initializes the mobile agent itinerary from strategy defined by user and with the local context obtained from the previous step as the procedure argument (`initItinerary(home)` method);
9. obtains from the user the list of the objects that will be moved together with the agent (`getMobileAgentList()` method);
10. associates the mobile object list and the platform mobile agent (`initMobileObjects()` method);
11. obtains from user the list of the agent masters (`getMasters()` method);
12. associates the master list and the platform mobile agent (`initMasters()` method);

13. obtains a reference to the `ReferenceManager` agent (`configureManagerId()` method);
14. sends a message to the `ReferenceManager` agent in order to notify mobile agent instantiation (`addOnReferenceTable()` method);
15. executes application-specific procedures immediately after the agent instantiation (`doAfterAgentInstantiation()` method).

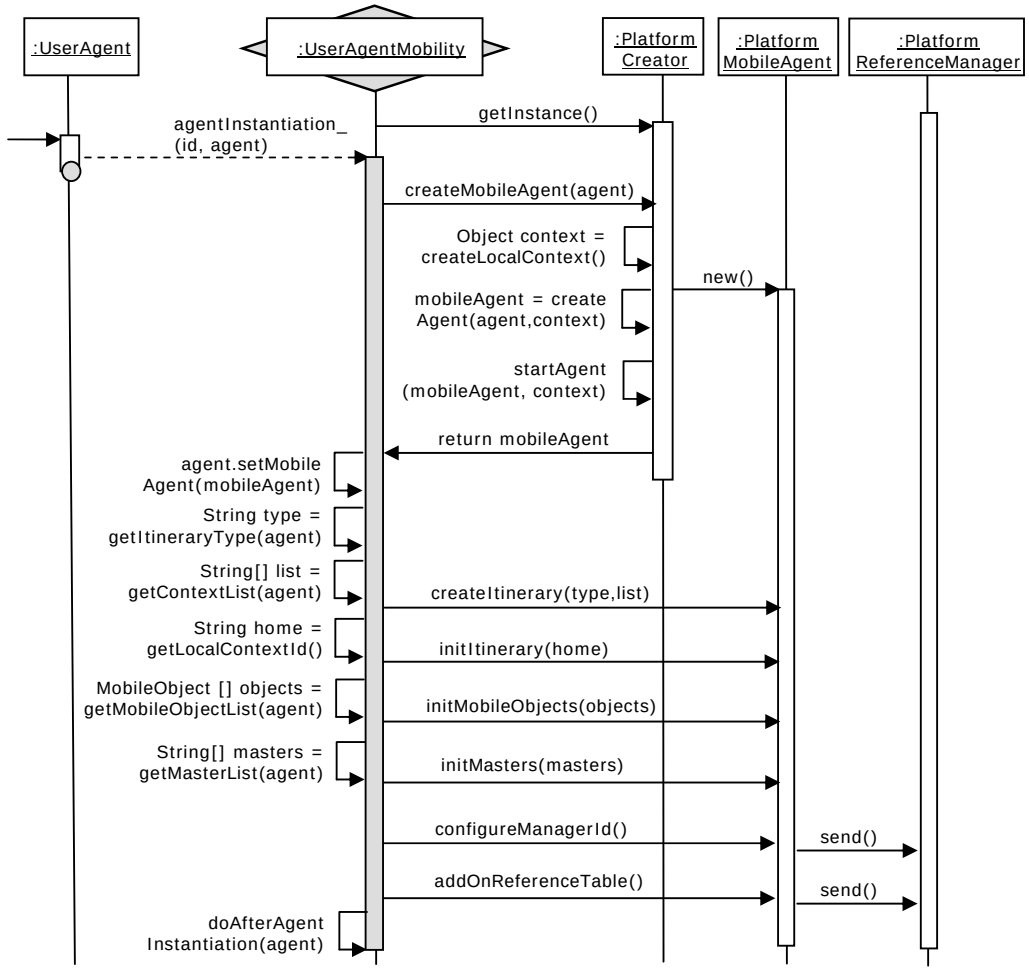

Figure 13. The AspectM Instantiation Protocol

In **Figure 14**, the agent migration protocol:

1. obtains from user the data related to the migration point (`object` argument);
2. from data obtained at previous step, invokes a procedure that verifies if migration must be effective (`checkDepartureNecessity()` method);
3. if verification returns a true value, executes departure procedures (`prepareToMove()` method);
4. changes the agent context through invocation of platform procedures (`move()` method).

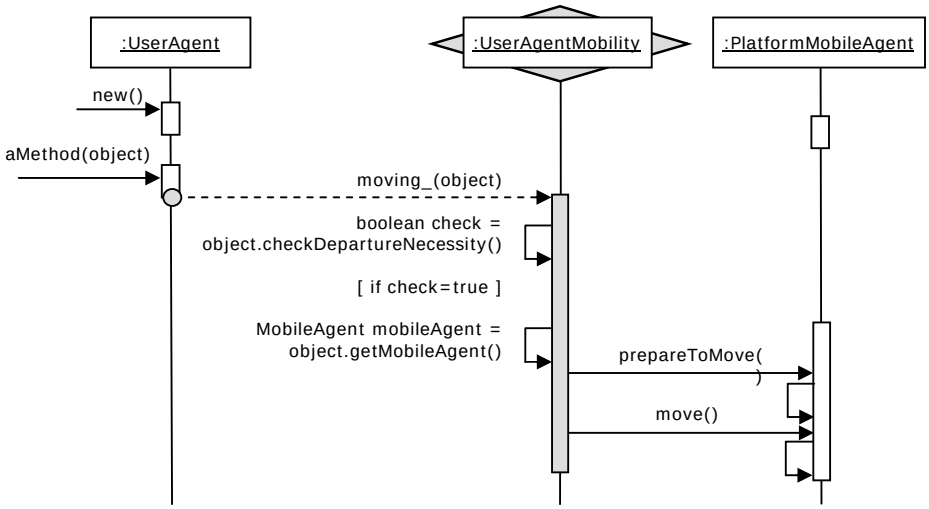

Figure 14. The AspectM Migration Protocol

A subtle detail in the `MobilityProtocol` component design is the `MobileObject` interface definition. This interface is used in two situations: (1) to define a common type for classes from which objects that compose the agent intrinsic knowledge are instantiated, such as in the case of the `getMobileObjectList()` return (**Figure 13**), or (2) to define a common interface for target objects in the migration protocol pointcut definition (**Figure 14**).

The agent initialization protocol is illustrated in **Figure 15**:

1. obtains from agent server the new local context id (`getLocalContextId()` method);
2. reconfigures agent itinerary with new context data (`configureItinerary()` method);
3. obtains from agent server the agent message id on the new context (`getMessageId()` method);
4. reconfigures agent attributes related to messaging with the context data (`configureMessageId()` method);
5. updates the application reference table with the context data, such as context and message ids, which provide an effective reference to the mobile agent for all other agents (`updateOnReferenceTable()`);
6. executes application-specific procedures (`doAfterArrivalHost()` method).

In **Figure 15**, the `updateOnReferenceTable()` method as well as the `addOnReferenceTable()` method (**Figure 13**) encapsulate a messaging between `UserAgent` and `ReferenceManager` agents. After `updateOnReferenceTable()` execution, new context and message ids are available to `ReferenceManager` agent.

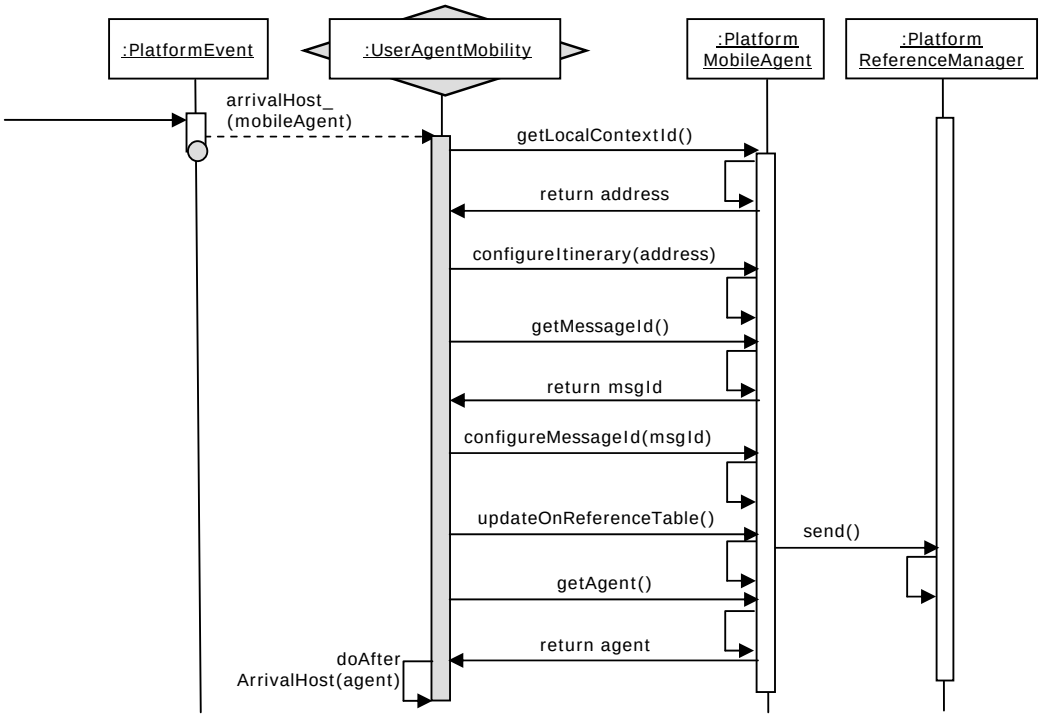

Figure 15. AspectM Initialization Protocol

In **Figure 16**, the agent destruction protocol:

1. executes application-specific procedures before agent destruction (`doBeforeAgentDestruction()` method);
2. removes from application reference table the reference to the mobile agent being destructed (`removeFromReferenceTable()` method);
3. affects the mobile agent destruction through invocation of platform procedures (`die()` method).

In **Figures 13 and 15**, note that the `doAfterAgentInstantiation()` and `doAfterArrivalHost()` methods in the `UserAgentMobility` aspect correspond to the return of the control flow to application-specific procedures. Note this does not occur in migration and destruction protocols (**Figures 14 and 16**) where control flow is not deviated back to user application.

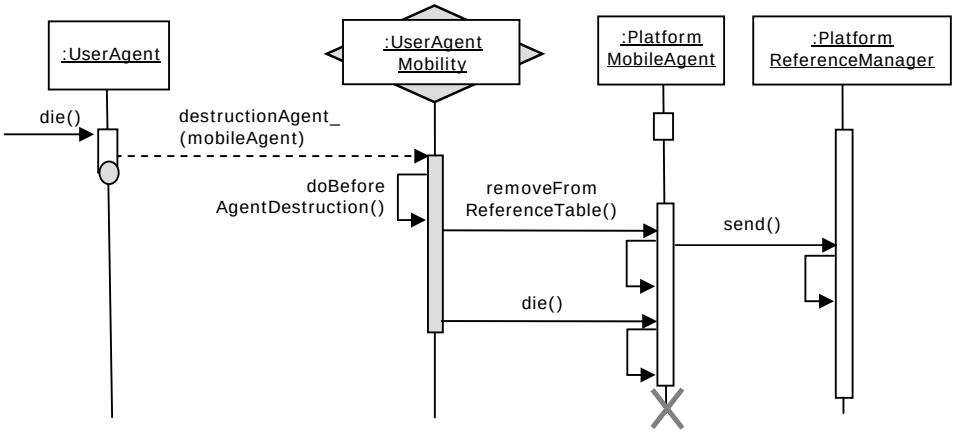

Figure 16. AspectM Destruction Protocol

6.3 Instantiation Process

One of the AspectM development purposes is to improve variability of mobility concerns in MAS, such as maintaining a flexible choice of mobility platforms in these systems. For example, in order to integrate a mobile agent system with the Aglets platform, an Aglets-specific adapter package must be developed from the AspectM framework. Once this package is independent from a particular mobile agent system, the classes that perform the integration between this system and the aglets will be reused in the instantiation process of any application that makes a choice of using services provided by the Aglets platform. Evidently, depending on the application-specific requirements, the Aglets package could be adapted in order to maintain the integration between MAS and different mobility platforms. AspectM users can execute the following steps to instantiate an application from AspectM framework:

1. for each `Mobility` subaspect, specify that an agent type or role has the mobility property, which is implemented by an intertype declaration `implements MobileElement`;
2. for each `Mobility` subaspect, specify the object types that will compose the agent (or role) mobile object list through intertype declarations `implements MobileObject`;
3. for each `Mobility` subaspect, specify the object types that may be moved together with the agent (or role) through intertype declarations `implements Serializable`;
4. if the itinerary notion is used, specify the application-specific itinerary classes extending the AspectM `Itinerary` interface (making concrete the `Itinerary` methods, such as the `getNext()`);
5. for each `Mobility` subaspect, make concrete the `getItineraryType()` and `getContextList()` methods (assuming that the itinerary classes are available);
6. for each `Mobility` subaspect, specify the instantiation protocol pointcut;
7. for each `Mobility` subaspect, specify the instantiation protocol procedures. In the case of application-specific methods, specify the calls to these procedures in the `doAfterAgentInstantiation()` method body;
8. for each `Mobility` subaspect, specify the migration protocol pointcut;
9. for each `Mobility` subaspect, specify migration protocol procedures, such as the `checkDepartureNecessity()` method, which verifies whether the migration should hold in the context of a migration-specific point;
10. for each `Mobility` subaspect, specify the initialization protocol procedures, such as application-specific procedures that will be executed on the `doAfterArrivalHost()` method.

For example, **Figure 17** presents the `UserAgentMobility` aspect, which extends the abstract `Mobility` aspect in order to specify the mobility-specific behavior to the `UserAgent` class. In other words, in the `UserAgentMobility` aspect we specify the AspectM hotspots in the context of `UserAgent` execution.

```
1:  public aspect UserAgentMobility extends Mobility {
2:  declare parents: UserAgent implements MobileElement;
3:  declare parents: UserMobileObject implements MobileObject;
4:  declare parents: UserObject implements Serializable;
5:    pointcut agentInstantiation(String referenceName,
6:    MobileElement agent): this(agent) && args(referenceName,*)
7:    && initialization(UserAgent+.new(String,*));
8:  String getItineraryType(MobileElement agent){...}
9: String[] getContextList(MobileElement agent){...}
10: MobileObject[] getMobileObjectList(MobileElement agent){...}
11: String[] getMasterList(MobileElement agent){...}
12: void doAfterAgentInstantiation(MobileElement agent){...}
13: void doAfterArrivalHost(MobileElement agent) {...}
14: pointcut agentMigration(MobileObject object):
15:    this(object) && execution(Hashtable
16:    UserMobileObject.execute (Vector));
17: ...
18: }
```

Figure 17. AspectM Instantiation for the UserAgent Class

In **Figure 17**, we specify the following AspectM hot spots for `UserAgentMobility`-specific context:

- ***Mobile elements' definition.*** The `UserAgent` class declares that implements the `MobileElement` interface (line 2). Thanks to this declaration, `MobileElement` attributes and methods are inherited by the `UserAgent` class;
- ***Mobile object list's definition.*** The `UserMobileObject` class declares that implements the `MobileObject` interface (line 3). Thanks to this declaration, `MobileObject` attributes and methods are inherited by the `UserMobileObject` class. The introduction of a `UserMobileObject` object in the `UserAgent` mobile object list is realized in the `getMobileObjectList()` method (line 11);
- ***Serializable elements' definition***, as the `UserObject` object (line 4);
- ***Instantiation pointcut definition*** (`agentInstantiation()` in lines 5-7). The `UserAgent` constructor is defined as the join point where the mobility code is introduced in the `UserAgent` class;
- ***Instantiation protocol procedures*** (lines 8-11), such as `getItineraryType()`, `getContextList()`, `getMobileObjectList()` and `getMasterList()`;
- ***Instantiation application-specific procedures*** (`doAfterAgentInstantiation()` in line 12). Application-specific procedures are executed immediately after the agent instantiation on the mobility platform;
- ***Initialization application-specific procedures*** (`doAfterArrivalHost()` in line 13). Application-specific procedures are executed immediately after the agent arrival at a new host;
- ***Migration pointcut definition*** (`agentMigration()` in lines 14-16). The `execute()` method execution in the `UserMobileObject` class is defined as a migration point.

Suppose now the package that bridges the AspectM framework to a specific platform is not available. The AspectM hot spots must be concrete in order to make the platform use possible. The instantiation process may apply the following sequence (an increasing order of complexity): (1) `MessageParser`; (2) `MobileAgent`; (3) `Event`; (4) `ReferenceCreator`; and (5) `ReferenceManager`. Remember the platform instantiation process is independent on the MAS.

7 The AspectM Case Studies

This section presents the case studies used for the AspectM evaluation: (1) the Expert Committee (**Section 3.1**), and (2) the MobiGrid [5]. Both systems were ideal for our experimental investigation for several reasons. First, the chosen systems have stringent modularity requirements due to the demand for producing reusable, adaptable and evolvable MAS architectures. Hence, all the system versions were developed with modularity principles as main driving design criteria, making sense the exploitation of AO software architectures. Second, the original architecture of each case study was developed in different contexts – the first system was developed in our own laboratory, while the second one has been developed out of our research environment [5]. Finally, they are realistic systems that involve emphasis on different MAS concerns, such as mobility, learning, autonomy, and their distinct compositions; they also encompasses the application of common mobility frameworks, such as JADE [7] and Aglets [43].

7.1 Expert Committee

In the EC system (**Section 3.1**), the mobility issues and the other agent concerns, such as the basic functionalities and the collaboration activities, must be separated how much it is possible. Particularly, for the ***chair role*** specification, it is necessary to specify the mobility protocol in such a way we could implement: (1) a seamless introduction of the mobility property into the chair role, and (2) a flexible integration between the EC and different mobility platforms. To do that, we may use the AspectM framework (**Section 6**). The next subsections describe the mobility introduction process into the EC.

7.1.1 The Chair Role

In **Figure 3**, `Role` subclasses are used to modularize several roles of the `ResearcherAgent` type (**Section 3.1**). A `Role` subclass is used to specify the extrinsic knowledge corresponding to a specific `ResearcherAgent` role. In other words, the agent intrinsic knowledge is defined in the class that represents the agent type (`ResearcherAgent` class); the extrinsic knowledge is defined in its respective `Role` subclass. For example, in the EC system, the `Chair` class, which is a `Role` subclass, defines additional attributes and methods for the `ResearcherAgent` agent type; in this way, a `ResearcherAgent` object can assume the chair role in a collaborative relationship. **Figure 18** presents the `Chair` role definition. We present only attributes and methods related to the chair-specific behavior; the methods that define how a chair role relates to a researcher agent type are not showed, once they are not relevant in this work.

In **Figure 18**, examples of attributes introduced by the `Chair` role into the `ResearcherAgent` agent type are: the `ResearcherAgent` instance associated to a `Chair` role instance (line 2), a plan of distribution of papers to reviewers (line 3), a list

of papers submitted (line 4), a list of paper reviewers (line 5), the limit dates to submit (line 6) and to review (line 7) papers, and so on. The `Chair` role also defines for `ResearcherAgent` agent type the methods for manipulation of attributes (lines 15-18). There are also attributes and methods related to the `Chair` mobility behavior; for example, the attribute that defines the `Chair` itinerary (line 8) and the methods for the itinerary manipulation (line 19).

```
1: public class Chair implements Serializable {
2: private ResearcherAgent agent;
3: private DistributionPlan distributionPlan;
4: private List papersList;
5: private List reviewersList;
6: private GregorianCalendar submissionDeadline;
7: private GregorianCalendar reviewDeadline;
8: private Itinerary itinerary; ...
9: public Chair(ResearcherAgent agent){
10:    this.agent = agent;
11:    distributionPlan = new DistributionPlan ();
12:    papersList = new Hashtable();
13:    submissionDeadline = new Calendar(); ...
14: }
15: public DistributionPlan getDistributionPlan(){
16:    return this.distributionPlan;
17: }
18: ...
19: public void addHost(String host){...}
20: }
```

Figure 18. The Chair Role Implementation in the Expert Committee

From the `Chair` implementation in **Figure 18**, it is reasonable to conclude that the `ResearcherAgent` agent type does not possess any attributes and methods corresponding to chair-specific beliefs and plans. In fact, the `Chair` class is developed in order to modularize the `Chair` role and the `ResearcherAgent` agent types. In consequence, it is reasonable also to conclude that the chair-specific mobility issues are also implemented in a modular way. However, not only the EC role classes, but also the EC agent types still extend classes and interfaces of APIs provided by the mobility platforms (**Figure 3**). The AspectM framework can be used to obtain a new definition for the chair mobility behavior (**Section 6**) without the architectural restrictions imposed by the mobility platforms' APIs on the MAS design. The EC mobility design using the AspectM framework is presented in the next section.

7.1.2 Mobility Issues of the Chair Role

In this section, we suppose the package that bridges the AspectM framework with JADE and Aglets platforms are available. **Section 6.3** presents the instantiation process to obtain this package from the AspectM. We have adopted the following steps in the AspectM instantiation for the EC classes:

1. for the `Chair` class, create the `ChairMobility` aspect extending the abstract `Mobility` aspect;
2. in the `ChairMobility` aspect, specify that the `ResearcherAgent` type implements the `MobileElement` interface ("`ResearcherAgent implements MobileElement`" in **Figure 11**). Thanks to this intertype declaration, the `ResearcherAgent` type becomes a mobile element in EC;
3. in the `ChairMobility` aspect, specify that the `InformationSearchingPlan` implements the `MobileObject` interface ("`InformationSearchingPlan implements MobileObject`" in **Figure 11**). Thanks to this intertype declaration, the `InformationSearchingPlan` type becomes a mobile object in the EC, which can be moved together with a `ResearcherAgent` instance, and can be used in the migration pointcut definition;
4. in the `ChairMobility` aspect, specify that the `Agenda` type implements the `Serializable` interface ("`Agenda implements Serializable`" in **Figure 11**). Thanks to this declaration, the `Agenda` type can be moved together with a `ResearcherAgent` instance;
5. since the itinerary notion is used, specify a `ChairItinerary` class that implements the AspectM `Itinerary` interface, which implements the `Itinerary` methods, such as the `getNext()`;
6. in the `ChairMobility` aspect, make concrete the `getItineraryType()` and the `getContextList()` methods;
7. in the `ChairMobility` aspect, specify the instantiation pointcut, which triggers the `ResearcherAgent` instantiation protocol (`agentInstantiation()`);
8. since there are not application-specific methods to be executed during `ResearcherAgent` instantiation protocol, not specify procedure calls in the `doAfterAgentInstantiation()` method;
9. in the `ChairMobility` aspect, specify that the `searchProfile()`execution of the `InformationSearchingPlan` type is the `ResearcherAgent` migration protocol pointcut;
10. in the `ChairMobility` aspect, specify the `searchProfileCheckDepartureNecessity()` method that verifies if `ResearcherAgent` must migrate in the context of the `searchProfile()`;
11. in the `ChairMobility` aspect, specify the initialization protocol procedures, such as application-specific procedures to be executed in the `doAfterArrivalHost()` method;
12. since there are not application-specific methods to be executed during `ResearcherAgent` destruction protocol, not specify procedure calls in the `doBeforeAgentDestruction()` method.

Figures 19 to 22 present the mobility protocol defined in the `ChairMobility` aspect. The `ChairMobility` aspect extends the `Mobility` aspect in order to specify the chair mobility-specific behavior. In the `ChairMobility` aspect, we specify the AspectM hot spots in the context of the `Chair` role.

Figure 19 presents the `ChairMobility` intertype declarations and the instantiation protocol. The instantiation protocol pointcut (lines 5-7) is defined as a `ResearcherAgent` instance initialization (line 7) even though the instantiation protocol is to be executed in a chair-specific context (`searchProfile()` is a `Chair` method, not a `ResearcherAgent` one). The pointcut refers to the `ResearcherAgent` type in order to guarantee that when a `Chair` role moves, the respective `ResearcherAgent` instance is also moved, once `Chair` behavior may depend on `ResearcherAgent`–specific knowledge.

Note also that the `getItineraryType()` method specifies the agent itinerary type as the `ChairItinerary` class (lines 9-11). The `getContextList()` method specifies hosts that compose the `Chair` itinerary (lines 12-17). The `getMobileObjectList()` method (lines 18-26) instantiates an `InformationSearchingPlan` plan, which is included in the agent mobile object list. There are no procedures to be called during the `ResearcherAgent` instantiation protocol. In fact, we have not specified method calls in the `doAfterAgentInstantiation()` method (line 8). The `ResearcherAgent` type does not specify the existence of any master agents (lines 27-29).

```
1: public aspect ChairMobility extends Mobility {
2: declare parents: ResearcherAgent implements MobileElement;
3: declare parents: InformationSearchingPlan implements MobileObject;
4: declare parents: Agenda implements Serializable;
5: pointcut agentInstantiation(String referenceName,  MobileElement agent):
6:    this(agent) && args(referenceName,*) &&
7:    initialization(ResearcherAgent+.new(String,*));
8: void doAfterAgentInstantiation(MobileElement agent){ }
9: String getItineraryType(MobileElement agent) {
10:    return "expertcommittee.chair.ChairItinerary";
11: }
12: String[] getContextList(MobileElement agent){
13:    String[] itinerary = new String[10];
14:    itinerary[0] = "Container-1";    ...
15:    itinerary[9] = "Container-10";
16:    return itinerary;
17: }
18: MobileObject[] getMobileObjectList(MobileElement agent){
19:    MobileObject[] mobileObjects = new Task[1];
20:    InformationSearchingGoal goal = new
21:    InformationSearchingGoal();
22:    InformationSearchingPlan plan = new
23:    InformationSearchingPlan(goal);
24:    mobileObjects[0] = plan;
25:    return mobileObjects;
26: }
27: String[] getMasterList(MobileElement agent) {
28:    return new String[0];
29: } } ...
```

Figure 19. The ChairMobility Instantiation Protocol

Figure 20 presents the `ChairMobility` migration protocol. The migration protocol pointcut is specified as the `searchProfile()` method execution (lines 2-3). This method is implemented in the `InformationSearchingPlan` class (line 3). The `searchProfileCheckDepartureNecessity()` method verifies if a `ResearcherAgent` instance needs to move in the context of the `searchProfile()` method (lines 4-7). The `informationNeedChecking()` pointcut and its advice (lines 8-14) are application-specific aspectual elements. In the `informationNeedChecking()` advice, any methods introduced in the `ResearcherAgent` instance by the "`ResearcherAgent implements MobileElement`" declaration can be invoked, including the `move()` (line 13).

```
1: public aspect ChairMobility extends Mobility { ...
2: protected pointcut agentMigration(MobileObject object):  this(object)
3:    && execution(Hashtable InformationSearchingPlan.searchProfile(Vector));
4: public boolean searchProfileCheckDepartureNecessity(Task task, Object
result){
5:    if (result == null) { return true; }
6:    else { return false; }
7: }
8: pointcut informationNeedChecking(Plan plan):
9:    this(plan) && execution(void DistributionPlan.executePlan(..));
10: before (Plan plan): informationNeedChecking(plan) {
11:    ...
12:    ResearcherAgent agent = plan.getAgent();
13:    agent.move("Container-5");    ...
14: } ...
15: }
```

Figure 20. The ChairMobility Migration Protocol

Figure 21 presents the `ChairMobility` initialization protocol. The application-specific procedures of the initialization protocol are invoked in the `doAfterArrivalHost()` method (lines 2-7). First, a test is specified in order to verify if the mobile agent is located in the original host (line 3). When the agent is not in the original host, it makes choice of tasks to execute (lines 4-6). Remember that in the `InformationSearchingPlan` execution (line 6), if the `searchProfile()` method is called, the agent migration protocol is triggered in the `Mobility` aspect (**Figure 20**, lines 2-3).

In **Figure 21**, other `Mobility` methods (lines 8-11) could be made concrete according to application-specific requirements. In EC, these methods are empty, once they are not being used in chair-specific migration scenarios. In the next case study (**Section 7.2**), we show examples where it is necessary to specify methods such as the `doAfterReceivingMessage()` and the `doBeforeCloneAgent()` ones.

```
1: public aspect ChairMobility extends Mobility { ...
2: protected void doAfterArrivalHost(MobileElement agent) {
3:    if (!agent.isAgentOut()) return;
4:    InformationSearchingPlan plan =(InformationSearchingPlan)
5:    agent.getMobileObjectOfType("InformationSearchingPlan");
6:    plan.executePlan((ResearcherAgent)agent, plan.getGoal());
7: }
8: void doAfterReceivingMessage(MobileElement agent, Message message) { }
9: protected void doBeforeCloneAgent(MobileElement agent) { }
10: protected void doAfterCloneAgent(MobileElement agent) { }
11: protected void doAfterCloneCreation(MobileElement agent) { }  ...
12: }
```

Figure 21. ChairMobility Initialization Protocol

Figure 22 presents the `ChairMobility` destruction protocol. Note that application-specific procedures to be executed immediately before agent destruction are not specified (line 2).

```
1: public aspect ChairMobility extends Mobility { ...
2: protected void doBeforeAgentDestruction(MobileElement agent) {} ...
3: }
```

Figure 22. ChairMobility Destruction Protocol

Figure 23 illustrates a `Chair` migration protocol scenario, which corresponds to the following sequence:

1. `SearchingInformationPlan` object instantiation. In other words, the creation of a chair searching information plan. The `SearchingInformationPlan` plan implements the `MobileObject` interface, and, for this reason, its execution context can be exposed in the `agentMigration()` pointcut;
2. the `ChairMobility` aspect detects the `searchProfile()` method execution in the context of the `SearchingInformationPlan` plan through the `agentMigration()` pointcut;
3. the `ChairMobility` aspect captures the `SearchingInformationPlan` context information. In this case, the relevant information is the return value of the `searchProfile()` method;
4. the `ChairMobility` aspect executes the `searchProfileCheckDepartureNecessity()` method in order to verify if the agent playing the chair role needs to migrate;
5. the `ChairMobility` aspect invokes the `prepareToMove()` method in order to execute departure procedures;
6. the `ChairMobility` aspect invokes the `move()` method in order to change the agent context through invocation of platform procedures.

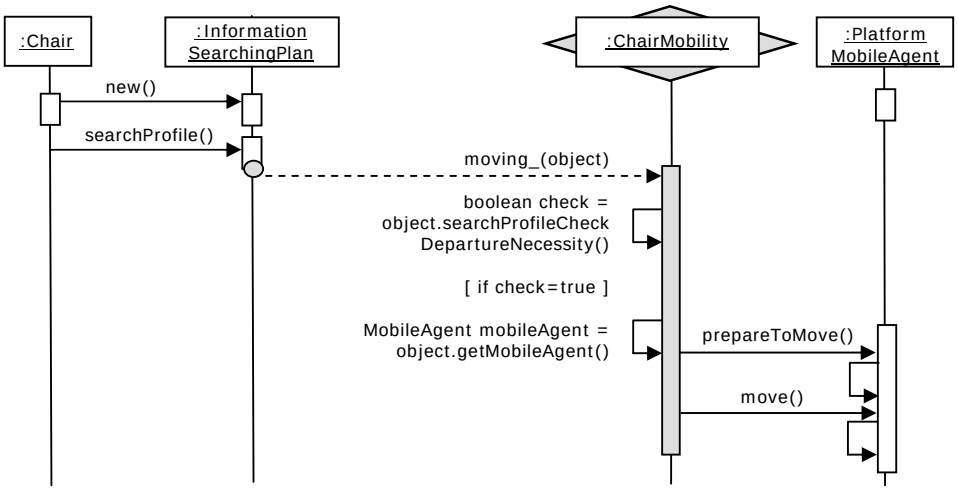

Figure 23. Chair Migration Protocol Scenario

Figure 24 presents the EC design using Aglets. Note that, even though the platform classes now correspond to the Aglets, the EC design remains the same. In other words, the EC design reaches a flexible integration with distinct mobility platforms (JADE and Aglets) using the AspectM framework. This occurs also with any other MAS using the AspectM framework and different platforms.

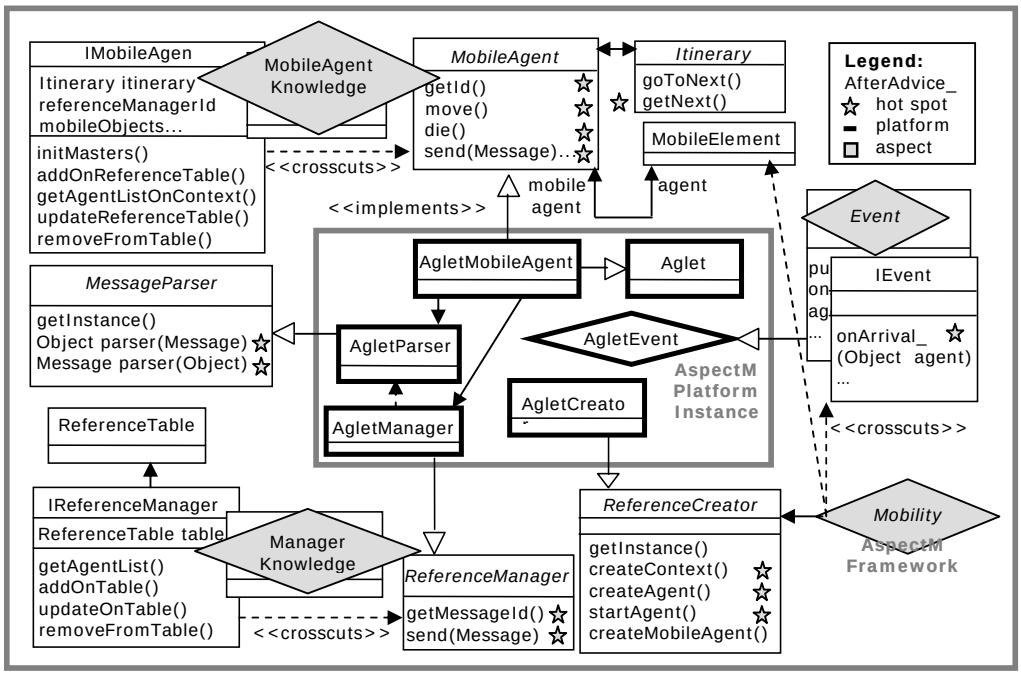

Figure 24. Expert Committee using AspectM and Aglets

7.2 MobiGrid

Our second case study was the ***MobiGrid framework*** [5], which is a mobile agent system within a grid environment project called ***InteGrade*** [31]. In this system, mobile agents are used to encapsulate and execute long processing tasks using the idle cycles of a network of personal workstations. The agents can migrate whenever the local machine is requested by its user since they are provided with automatic migration capabilities. The original MobiGrid architecture was defined based on the OO framework provided by the Aglets platform [43]. Due to the high coupling between the Aglets underlying model and the MobiGrid mobile agents, we decided to reengineer the MobiGrid architecture taking into account the following requirements: (1) to modularize the MobiGrid mobility-specific concerns, that is, to promote an explicit separation between the crosscutting mobility concerns and other non-crosscutting MobiGrid concerns; and (2) to enhance the MobiGrid variability in terms of a flexible choice of distinct mobility platforms to be used (e. g. Aglets [43], JADE [7], etc.). This section presents the MobiGrid original design (**Section 7.2.1**) and the MobiGrid reengineering outcome using the AspectM framework (**Section 7.2.2**). We also present the mobility-specific tangling and scattering problems (**Section 7.2.3**) solved by the MobiGrid reengineering steps using AspectM (**Section 7.2.4**).

7.2.1 The MobiGrid Original Design

Figure 25 presents the MobiGrid detailed design, which results from the composition between the MobiGrid original design and the OO framework provided by the Aglets platform [43]. For simplification purposes, we present only the most important design elements, privileging a partial design view. We do not show the physical view of the MobiGrid software architecture. **Figure 25** distinguishes the hot spots (variable parts), which are marked with a star, from the frozen parts of the frameworks.

The main concepts of the MobiGrid framework using Aglets are: (1) the task (`TaskAgent` class) and its state (`TaskState` class) that a programmer wants to submit to the MobiGrid framework, (2) the manager (`ManagerAgent` class) that the programmer must reference in order to register his tasks, (3) the server (`Server` class) that the MobiGrid maintains in order to use the Aglets runtime and its resources, (4) the event listeners (`TaskCloneListener`, `TaskMobilityListener`, and `ServerListener` interfaces) that a programmer must define in order to trigger specific procedures at special instants of a task or server execution, (5) the proxies (`AgletProxy` objects), which are used by the MobiGrid in order to perform message exchanges among tasks, managers, and servers, and (6) the ids (`AgletId` objects) that the MobiGrid uses in order to identify tasks, managers, and servers.

The `TaskAgent` and the `TaskState` classes allow the specification of a ***task*** and its ***state*** (the MobiGrid hot spots), both moved together with an Aglets mobile agent. To define a task and its state, a programmer must make these classes and their abstract methods concrete. In particular, the `TaskAgent define()` method must be made concrete in such a way it returns the `TaskState` object corresponding to the task that the programmer wants to submit to the MobiGrid framework. The programmer also must specify the task implementation in the `TaskState run()` method. The specification of these hot spots is realized in a MobiGrid client (`Client` class), which is the abstraction used to identify a client machine in the grid. To register a task and its state, each MobiGrid client maintains a reference to a task manager (`ManagerAgent` class, see later).

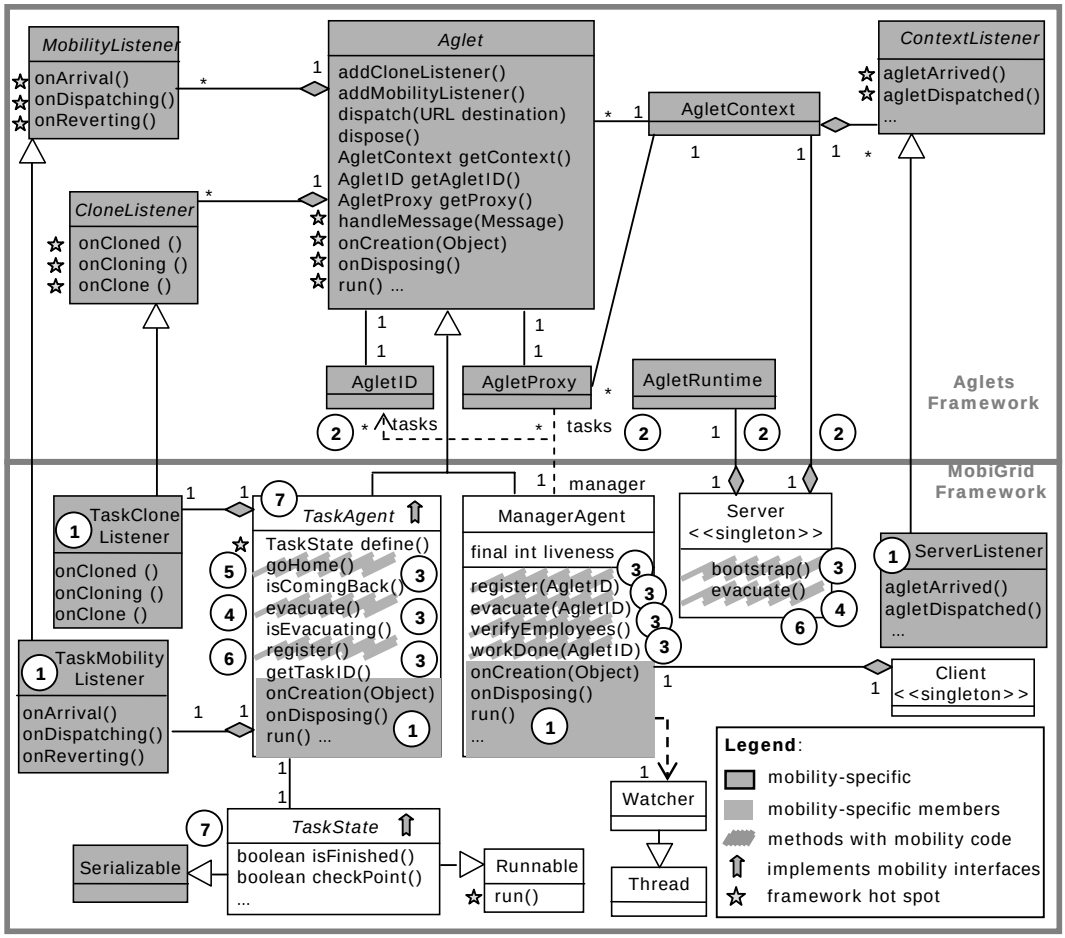

Figure 25. The MobiGrid Original Design

Note that the `TaskAgent` class extends the abstract `Aglet` class, which is the basic class for mobile agent specification on Aglets platform (Aglets hot spot). Note also the `TaskAgent` class maintains references to `MobilityListener` and `CloneListener` objects (Aglets hot spots), which correspond to ***event listeners*** used for specification of procedures that must be triggered at specific instants of a mobile agent lifecycle, such as immediately after a task's arrival at a new host. A task also maintains a reference to an `AgletProxy` object corresponding to the manager responsible for its migration and cloning. The ***proxies*** are mainly used for communication purposes in the Aglets platform (a frozen spot).

The `ManagerAgent` class allows the instantiation of a ***manager*** (MobiGrid frozen spot), which is a stationary agent that manages references to tasks and clones in a client (there is a manager on each machine where it is possible to submit a task). The manager exchanges messages with tasks during registering, migration and cloning scenarios. When a task is submitted, the manager queries the InteGrade [31] infrastructure searching for an idle machine to which the task may be dispatched. The manager also maintains a task clone to ensure the task liveness. In **Figure 25**, note that the `ManagerAgent` class extends the `Aglet` class (Aglets hot spot) even though a `ManagerAgent` object is not a mobile, but rather a stationary agent. The `ManagerAgent` class is implemented in this way in order to use the Aglets tools for messaging. A manager maintains references to `AgletProxy` objects corresponding to

the tasks and clones under its responsibility. Again, the `AgletProxy` class is used mainly for communication purposes.

The `Server` class represents an ***agent server*** (MobiGrid frozen spot) that is installed on each grid machine to provide Aglets resources to MobiGrid agents. To do this, the server is associated with `AgletRuntime` and `AgletContext` objects, which provide an execution environment for managers and tasks. When its user requests a machine, the server asks the evacuation of the local tasks. These tasks query their managers, which communicate with InteGrade looking for idle machines. When the managers get such information, they take action in order to evacuate the tasks to new machines. To ask for evacuation of local tasks, the server gets references to `TaskAgent` objects (through the `AgletContext` object associated with a server) and communicates with them through proxies (Aglets frozen spots). In each server there is a need for a `ContextListener` object (Aglets hot spot), which corresponds to a listener used for specification of procedures to be triggered at specific instants of a context manipulation.

7.2.2 Mobility-Specific Tangling and Scattering in MobiGrid Design

From the description above, we can observe that the composition of MobiGrid with the Aglets framework causes a high coupling between such frameworks. In fact, note in **Figure 25** the presence of numbers surrounded by circles throughout the figure. These elements are the same as defined in **Section 3.2** and are used to point out that the implementation of mobility concerns in the MobiGrid framework has a huge impact on the basic framework functionalities.

In the MobiGrid framework, agent types extend the `Aglet` class to incorporate the mobility capabilities. The use of inheritance results in code replication as well as code tangling and scattering (problem ①). The agent classes also need to hold explicit references to Aglets mobility elements (e.g. aglets ids, contexts, and proxies) as attributes (problem ②). These classes also manage these elements by Aglets-specific mobility methods (problem ③). As a consequence, the basic functionalities, context-specific services and messaging exchanges are amalgamated to mobility methods.

A unique method has mobility code in order to decide when a task should move (`evacuate()` method). This would not entail any problem except for the fact that in evolution scenarios, the code related to migration decision is replicated on several agent type methods. For example, these methods contain replicated mobility code relative to the decision about when an agent should move to a remote environment (problem ④), or when the agent should go back to the home location (problem ⑤). In addition, there is a spread of usual preconditions and postconditions when an agent moves to another host (problem ⑥).

Classes also have to implement the `Serializable` interface for allowing the objects, which are part of the agent, to be moved across hosts (problem ⑦). Again, the `Serializable` interface is just a representative example: OO APIs from platforms usually provide a number of interfaces with methods that are implemented by systems in order to ensure that actions can be automatically executed at specific moments through the mobile agent lifecycle; for instance, MobiGrid procedures are automatically executed immediately after the mobile agent cloning (`TaskAgent implements CloningListener`) or just before and/or immediately after migration (`TaskAgent implements MobilityListener`).

7.2.3 The MobiGrid Reengineering using AspectM Framework

Figure 26 presents a partial view of the MobiGrid reengineering using the AspectM.

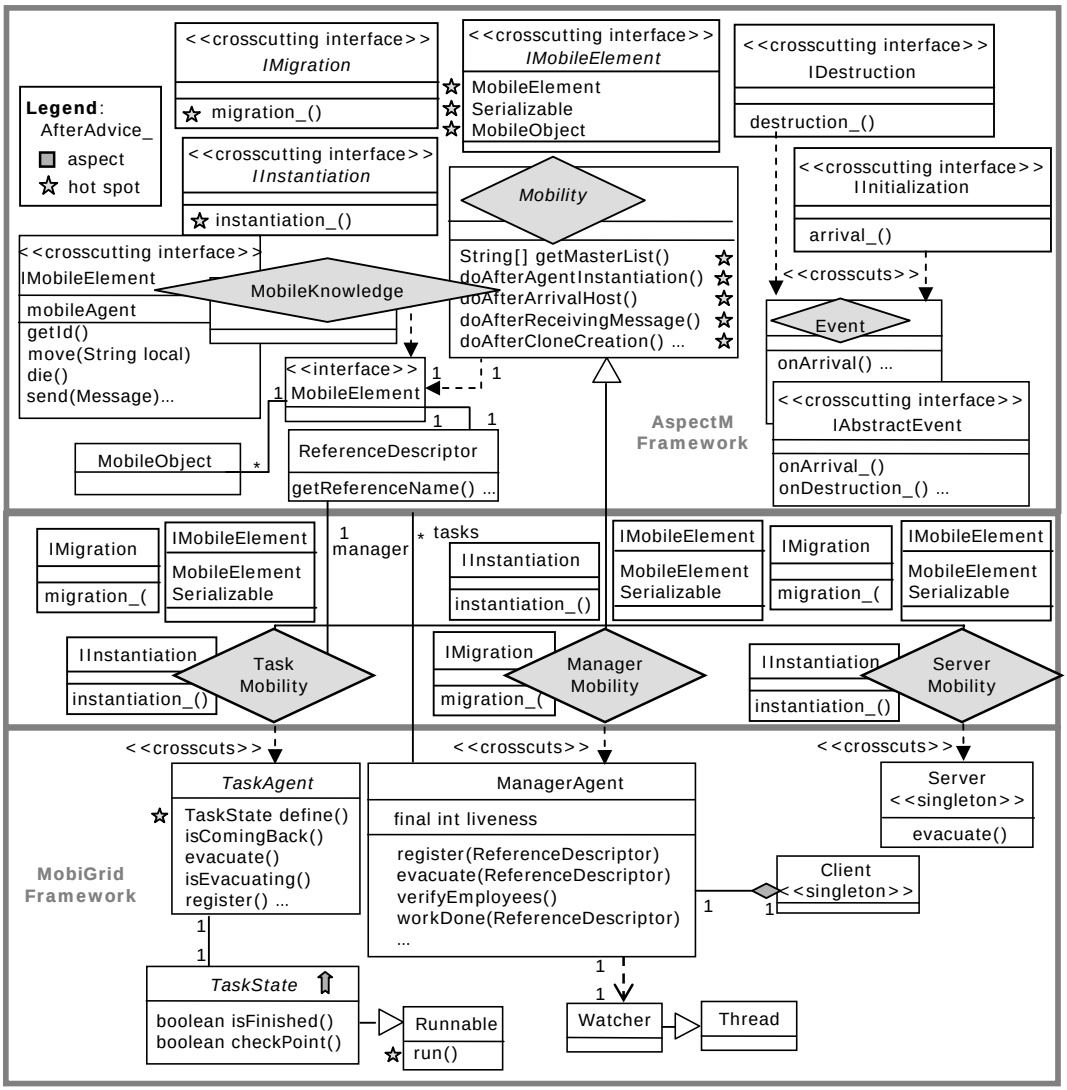

Figure 26. The MobiGrid Reengineering using the AspectM Framework

In order to perform the MobiGrid reengineering, we basically introduce the `Mobility` subaspects (one for each element to be defined as mobile). In order to reengineer the MobiGrid mobility-specific concerns, we have specified the `Mobility` hot spots for each element with mobility requirements (**Section 6.3**):

1. for each class with mobility requirements (`TaskAgent`, `ManagerAgent`, and `Server` classes in **Figure 25**), specify a `Mobility` subaspect (`TaskMobility`, `ManagerMobility` and `ServerMobility` subaspects in **Figure 26**);
2. make concrete the `Mobility IMobileElement` interface for each `Mobility` subaspect (declaration in a `Mobility` subaspect that the element with mobility requirements implements the `MobileElement` interface);

3. make concrete the `Mobility IInstantiation` interface for each `Mobility` subaspect (specification of the instantiation pointcut, which triggers the `instantiation_()` advice);
4. make concrete the `Mobility IMigration` interface for each `Mobility` subaspect (specification the migration pointcut that triggers the `migration_()` advice);
5. make concrete the `Mobility` methods corresponding to procedures invoked on agent instantiation (e. g. `getMasterList()`, `getItineraryType()`, etc.);
6. specify MobiGrid-specific instantiation and initialization procedures to be executed on `doAfterAgentInstantiation()` and `doAfterArrivalHost()` methods, respectively.

Figures 27 to 29 present some code of `ServerMobility`, `ManagerMobility` and `TaskMobility` aspects. In **Figure 27**, note the specification of the instantiation pointcut that triggers the creation of a mobile agent that represents a server on the platform (lines 3-6). The `ServerAgent` agent now sends evacuation messages through `MobileElement` methods in the `ServerMobility` aspect.

```
1: public aspect ServerMobility extends Mobility {
2: declare parents: ServerAgent implements MobileElement;
3: pointcut agentInstantiation(String referenceName,
4:    MobileElement agent): this(agent) &&
5:    args(referenceName,*) &&
6:    initialization(ServerAgent+.new(String,*));
7: void doAfterAgentInstantiation(MobileElement agent){ } ...
8: }
```

Figure 27. The MobiGrid Reengineering: ServerMobility Aspect

In **Figure 28**, we have specified the `doAfterReceivingMessage()` method (lines 8-10), which handles the messages received by the `ManagerAgent` agent. In other words, a mobile agent is instantiated on a platform to represent the MobiGrid manager in order to allow this agent exchange messages with other agents. The manager agent exchanges messages especially with the tasks under its control.

```
1: public aspect ManagerMobility extends Mobility {
2: declare parents: ManagerAgent implements MobileElement;
3: pointcut agentInstantiation(String referenceName,
4:    MobileElement agent):
5:    this(agent) && args(referenceName,*) &&
6:    initialization(ManagerAgent+.new(String,*));
7: void doAfterAgentInstantiation(MobileElement agent){ }...
8: void doAfterReceivingMessage(MobileElement agent, Message message){
9:   ManagerAgent manager = (ManagerAgent) agent;
10:   manager.handleMessage(message); } ...
11: }
```

Figure 28. The MobiGrid Reengineering: ManagerMobility Aspect

In **Figure 29**, in the `TaskMobility` aspect, several methods are specified in order to implement the `TaskAgent` mobility-specific behavior. However, even though a `TaskAgent` instance is effectively a mobile agent, the itinerary-specific methods have not been made concrete (lines 8-11). This is due to the next host selection strategy, which is determined by the InteGrade infrastructure itself [31]. In addition, the migration protocol pointcut also has not been used (line 20); the `TaskAgent` migration is triggered by events not established previously.

```
1: public aspect TaskMobility extends Mobility {
2: declare parents: TaskAgent implements MobileElement;
3: pointcut agentInstantiation(String referenceName,
4:    MobileElement agent): this(agent) &&
5:    args(referenceName,*) &&
6:    initialization(TaskAgent+.new(String,*));
7: void doAfterAgentInstantiation(MobileElement agent){...}
8: String getItineraryType(MobileElement agent)
9:    { return null; }
10: String[] getItineraryList(MobileElement agent)
11:    { return null; }
12: Task[] getTaskList(MobileElement agent)
13:    { return null; }
14: String[] getMasterList(MobileElement agent){
15:    Enumeration idList = agent.
16:    getAgentListOnTheContext(agent.getCurrentAddress());
17:    /* searching for the manager's descriptor */
18:    String[] masters = new String[1];
19:    masters[1] = managerName;
20:    return masters;
21: }
22: pointcut agentMigration(Task task);
23: void doAfterReceivingMessage(MobileElement agent,
24:    Message message){
25:    TaskAgent task = (TaskAgent) agent;
26:    task.handleMessage(message);
27: }
28: ...
29: }
```

Figure 29. The MobiGrid Reengineering: TaskMobility Aspect

7.2.4 Summary of The MobiGrid Reengineering Steps

In the following, we summarize the MobiGrid reengineering steps.

Removing inheritance relationships between MobiGrid and Aglets platform. After we specify the `Mobility agentInstantiation` pointcut corresponding to the MobiGrid object with mobility requirements, and declare through an inter-type declaration that the object class implements the `MobileElement` interface, the `Mobility` aspect introduces the mobility capabilities into an object (task, manager or server).

Removing references to Aglets contexts. The direct references to Aglet contexts have been substituted by invocations of the predefined `MobileElement` methods, which are common services abstracted from distinct mobility platforms, and available to all elements that implement the `MobileElement` interface. For example, code such as `getAgletContext().getAgletProxies()` is substituted with the corresponding `MobileElement` service call, the `getAgentListOnContext(String)` method.

Removing references to Aglets proxies. Proxies have been substituted by agent descriptors returned by methods such as `getAgentListOnContext()`. For example, MobiGrid code that invokes methods such as `proxy.sendMessage()` is substituted by `MobileElement` methods such as `sendMessage(Message)`. To set the `Message` argument, it must specify the agent descriptor that corresponds to the message receiver, which can be obtained through a search among the agent descriptors returned by the `getAgentListOnContext()` method.

Removing Aglets listeners. Once the `Mobility` aspect allows to specify procedures to be executed before and/or after operations as cloning, migration, and messaging, the `TaskCloneListener`, `TaskMobilityListener` and `ServerListener` procedures (**Figure 25**) have been transferred to corresponding methods in the `TaskMobility` aspect (**Figure 26**).

Removing coupling between servers and Aglets runtime. Once the `Mobility` aspect now encapsulates the Aglets internal elements manipulation, we do not need to make direct references to this platform runtime any more. In **Figure 1**, the `Server` has the `evacuate()` method, beyond the `bootstrap()` which manipulates `AgletContext` and `AgletRuntime` objects; in **Figure 26**, the `Server` class has only the `evacuate()` method.

8 Evaluation

The usefulness and usability of the ArchM architecture (**Section 5**) has been evaluated in the context of EC system (**Sections 3.1 and 7.1**) and MobiGrid framework (**Section 7.2**), two medium-sized case studies from different application domains and originally composed with two distinct mobility platforms. In this evaluation, we have not mentioned application-specific requirements or requirements related to platform models, since ArchM proposes an architectural solution to code mobility modularization that is independent on particular MAS platforms or applications. In the following, we describe the procedures we have applied to evaluate the ArchM architecture (**Section 8.1**). After that, we discuss some results on how ArchM architecture and AspectM framework were effective to address the architectural restrictions imposed by platforms on MAS design (**Section 8.2**) as well as the more fine-grained mobility tangling and scattering problems discussed (**Section 8.3**).

8.1 Evaluation Procedures and Assessment Metrics

We have used a suite of architectural metrics (**Table 1**) to support modularity evaluation of the ArchM software architecture (**Section 5**). We have not used conventional architectural assessment methods because they traditionally focus either on the architecture coverage of scenarios described in the requirements specification [14], or on the satisfaction of high-level non-functional requirements (e.g. [1]) without a clear focus on modularity assessment. Our goal here was to assess internal structural attributes in the architecture description with a direct impact on architecture

modularity. As a consequence, our investigation has provided us with a more fine-grained understanding of the overall architecture quality since modularity impacts a huge number of non-functional requirements in MAS, such as reusability, adaptability, flexibility, changeability and the like.

A discussion about each of those architectural metrics is out of the scope of this work. **Table 1** presents a definition for each of the used metrics and their association with distinct modularity attributes. This suite includes metrics for architectural separation of concerns (SoC), architectural coupling, component cohesion and interface complexity. We have already used similar categories of metrics [24, 51] for evaluating aspect and object-oriented designs in a number of systematic case studies [9, 10, 18, 28, 42] not related to mobile agent systems. They have been proved to be effective modularity indicators for detailed design and implementation artifacts. The metrics can also be classified in two categories according to the architectural viewpoint under assessment: concern viewpoint or component viewpoint. On one hand, the results of the SoC metrics are obtained for each concern of interest in the system. On the other hand, the results of the other metrics are all gathered for each component in the system architecture. **Table 1** also relates the metrics to the viewpoint from which their values are obtained. For all the employed metrics, a lower value implies a better result.

Table 1. Architectural Metrics Suite

Attribute	Metric	Definition	
Architectural Separation of Concerns	Concern Diffusion over Architectural Components (CDAC)	Counts the number of components that encompass a concern.	Concern
	Concern Diffusion over Architectural Interfaces (CDAI)	Counts the number of interfaces related to a concern.	
	Concern Diffusion over Architectural Operations (CDAO)	Counts the number of operations (defined in interfaces) that are related to a concern.	
Architectural Coupling	Architectural Fan-in	Counts the number of components that require service from a component (caller components).	Component
	Architectural Fan-out	Counts the number of components from which the component requires service (callee components).	
Component Cohesion	Lack of Concern-based Cohesion (LCC)	Counts the number of concerns addressed by a component.	
Interface Complexity	Number of Interfaces	Counts the number of interfaces of each component.	
	Number of Operations	Counts the number of operations in the Interfaces of each component.	

The metrics of SoC measure the degree to which a single concern in the system maps to the architectural elements (components, interfaces, operations and parameters). The interface complexity is measured in terms of the total number of interfaces, operations and parameters of each component. The coupling metrics measure the number of components connected to each component. The cohesion metric computes each component's semantic cohesion based on the number of concerns addressed by it. The higher the number of different concerns in the component the lower is the cohesion.

In order to proceed with the measurement of SoC, there is an architecture shadowing process in which the architect must assign every component element (interface, operation and parameter) to one or more concerns. In the EC as well in the MobiGrid, we treated the mobility concerns and the application itself as the driving

concerns to be modularized. After the shadowing of the architecture models, the data of the SoC metrics (CDAC, CDAI, and CDAO) was manually collected.

8.2 Architectural Evaluation

This section presents the results of the measurement process. The data have been collected based on the set of defined measures (**Section 8.1**) in the two case studies. The presentation is divided into three parts. **Section 8.2.1** presents the evaluation results for the separation of architectural concerns. **Section 8.2.2** presents the results for the coupling and cohesion metrics. **Section 8.2.3** presents the results for the interface complexity metrics. We present the results by means of tables that place the values of the metrics for the aspect-oriented and non-aspect oriented architectures of each system side-by-side.

8.2.1 Separation of Architectural Concerns

In the quantitative evaluation of the EC system, the data collected for both AO and non-AO architectures shows favorable results for the AO version for most of the metrics used. **Table 2** presents the complete data collected for both EC architecture versions considering the SoC metrics. The application of the SoC metrics allowed us to evaluate how effective was the separation of the agency concerns in the both EC architectures. These metrics count the total number of components, interfaces and operations dedicated to implement a concern.

Table 2. Expert Committee Architectures: Separation of Concerns Measures

Concern	#components (CDAC)		#interfaces (CDAI)		#operations (CDAO)	
	AO	Non-AO	AO	Non-AO	AO	Non-AO
Kernel	1	1	4	2	68	14
Interaction	1	2	3	9	10	22
Adaptation	1	2	2	6	5	34
Autonomy	1	2	3	7	31	80
Collaboration	1	2	4	6	37	87
Mobility	1	2	3	3	20	35
Learning	1	2	2	4	6	16

We can observe significant differences between the AO and non-AO versions for all the SoC metrics. **Table 2** shows that the non-AO architecture requires two components to address each of the system concerns (CDAC metric), except for the Kernel concern. It happens because the `Kernel` component needs to inevitably embody functionalities from the different concerns besides to implement the kernel-specific functionalities; the `Kernel` component plays the mediator role and, as a consequence, propagates information relative to every concern to the other "colleague" components (**Section 3.1**). On the other hand, each component in the AO version is responsible for implementing the functionalities associated with exactly one concern because such information is directly collected from the context where it is generated through crosscutting interfaces; as a result, the design of the `Kernel` component and its interfaces are not affected by other concerns.

We can also observe in **Table 2** that the AO version requires fewer interfaces (CDAI metric) and operations for most of the system concerns, with the exception of the Kernel concern. The Kernel concern in the AO version is represented by the `Kernel` component. This component needs to expose new interfaces in the AO version to enable the implementation of the different aspectual components. However, all these additional interfaces are part of the Kernel functionalities and separation of architectural concerns is not hindered. As we can see in **Table 2**, there is also a significant increase in the number of operations (CDAO metric) for almost all the agency concerns in the non-AO version; the only exception is the Kernel concern. The Interaction concern, for example, is addressed in the AO version by 3 interfaces and 10 operations. While the same Interaction concern in the non-AO version requires 9 interfaces and 22 operations. This growth in the non-AO architecture is mainly caused by the use of the mediator-based pattern, which requires the additional interfaces in the `Kernel` component with their associated operations.

Table 3 shows the results for the three SoC metrics for the MobiGrid architectures. The AO architecture performed better than the non-AO version in terms of SoC. As shown in **Table 3**, the mobility concerns are scattered over fewer architectural components in the AO architecture (CDAC metric). These concerns are present in 4 components in the non-AO architecture, whereas they crosscut only 3 components in the AO architecture. This occurs because, in the non-AO architecture, the `MobiGrid` component encompasses issues for explicitly handling of mobility lifecycle events. These events are captured by the `IMobileElement` crosscutting interface in the AO architecture, which makes the mobility-related interfaces unnecessary in the `MobiGrid` component.

Table 3. MobiGrid Architectures: Separation of Concerns Measures

Concern	#components (CDAC)		#interfaces (CDAI)		#operations (CDAO)	
	AO	Non-AO	AO	Non-AO	AO	Non-AO
Mobility	3	4	13	23	326	407
Application (MobiGrid)	1	1	1	1	18	18

The SoC metrics also showed better results for the AO architecture in terms of number of interfaces (CDAI metric) – 13 vs. 32 – and number of operations (CDAO metric) – 326 vs. 407. This is mainly caused because the `MobilityProtocol` and `MobilityManagement` aspectual components need fewer interfaces and operations for handling events. The aforementioned absence of mobility interfaces in the `MobiGrid` component also contributes to this difference.

8.2.2 Architectural Coupling and Component Cohesion

Tables 4 and 5 present the results for architectural coupling and component cohesion metrics for the Expert Committee and MobiGrid architectures, respectively. As in **subsection 8.2.1**, the tables in this subsection and in **subsection 8.2.3** place the metrics values for the AO and non-AO architectures side-by-side. However, since the values here are for each component (component viewpoint), the bottom of the tables also provides the total values (sum of all the component measures) that represent the results for the overall architecture viewpoint. Therefore, rows labeled "Total" indicate the tally for the system architecture, while rows labeled "Diff" indicate the percentage difference between the AO and non-AO architectures in the system viewpoint relative

to each metric. A positive value means that the non-AO architecture fared better, whereas a negative value indicates that the AO architecture exhibited better results.

As we can observe in **Table 4**, there is an expressive coupling increase in the non-AO Expert Committee architecture considering the number of requiring components (Architectural Fan-in metric). The fan-in is 12 in the non-AO architecture, while it is 9 in the AO architecture, representing a difference of 25% in favor of the latter. This occurs because in the AO version the services of several aspects (e.g. Adaptation, Autonomy, Learning) are not requested by other components granted to the dependency inversion promoted by AO architectures.

Table 4. Expert Committee Architectures: Coupling and Cohesion Measures

Component	Architectural Fan-Out		Architectural Fan-In		#Concerns (Lack of Cohesion)	
	AO	Non-AO	AO	Non-AO	AO	Non-AO
Kernel	0	6	5	5	1	7
Interaction	3	2	2	2	1	1
Adaptation	2	1	0	1	1	1
Autonomy	2	1	0	1	1	1
Collaboration	1	1	1	1	1	1
Mobility	2	1	1	1	1	1
Learning	1	0	0	1	1	1
Total:	11	12	9	12	7	13
Diff:	-8.3%		- 25.0%		-46.2%	

With respect to the architectural fan-out, the measures did not show an expressive difference from the system viewpoint; the difference was lower than 10% (**Table 4**). We assess the lack of cohesion of a component counting the number of distinct concerns addressed by it, which is captured by the Lack of Concern-based Cohesion (LCC) metric. LCC measurement resulted in better results for the AO version (13 vs. 7 = 46.2%). This superiority is justified by the fact that in the non-AO architecture, the `Kernel` component needs to implement required interfaces associated with the six system concerns (CBLC metric). Hence, there is an explicit architectural tangling in the `Kernel` component.

The AO architecture of the MobiGrid system presented better outcomes in terms of the two coupling metrics and in terms of the cohesion metric as well (**Table 5**). The non-AO architecture exhibited architectural fan-out 50% higher than the AO architecture. This difference is a consequence of the reduction of fan-out in both `MobiGrid` and `MobilityManagement` components in the AO version, since they do not have to explicitly call the `MobilityProtocol` component for notifying events. Being an aspectual component, `MobilityProtocol` captures the events by means of crosscutting interfaces. `MobilityPlatform` also contributes for decreasing the fan-out, because it does not need to be connected to the `MobilityManagement` component in order to notify events. In this case, the aspectual `MobilityManagement` component observes the events by means of its `IReferenceObserver` crosscutting interface. For the same reasons, the architectural fan-in metric also presented worse results for the publisher-subscriber version of the architecture (50% higher). In this case the fan-in reduction is observed in the `MobilityProtocol` and `MobilityManagement` components.

Table 5. MobiGrid Architectures: Coupling and Cohesion Measures

Component	Architectural Fan-Out		Architectural Fan-In		#Concerns (Lack of Cohesion)	
	AO	Non-AO	AO	Non-AO	AO	Non-AO
MobilityPlatform	0	1	1	1	1	1
MobilityManagement	1	2	1	2	1	1
MobilityProtocol	2	2	0	2	1	1
MobiGrid	0	1	1	1	1	2
Total:	3	6	3	6	4	5
Diff:	-50.0%		-50.0%		-20.0%	

Similar to the Expert Committee case, the cohesion measures in the MobiGrid architectures pointed out a difference in favor of the AO solution only in one of the components, namely the `MobiGrid` component. This component encompasses two concerns in the non-AO solution: the MobiGrid concern, which is the primary purpose of the original definition of this component, and the mobility concern. Conversely, in the AO solution, the `MobiGrid` component is not affected by the mobility concern and entirely dedicated to its main concern.

8.2.3 Interface Complexity

Tables 6 and 7 show the results for the interface complexity metrics for the Expert Committee and MobiGrid architectures, respectively. Regarding the Expert Committee system (**Table 6**), the metrics demonstrate the modularity benefits obtained in the AO version compared to the non-AO one. There was a bigger difference in the number of interfaces specified for each version (35 vs. 21 = 43.2%), which favors the AO version. This difference is mainly due to the additional interfaces of the `Kernel` component, but it is also a result of the values collected for other components. The increase in the number of interfaces metric for the non-AO version is also reflected in the number of operations. **Table 6** shows that the number of operations is 38.5% higher in the non-AO version. Again, it happens because the `Kernel` component plays the mediator role and, as a consequence, it has additional interfaces and operations to propagate information relative to every concern to the other "colleague" components.

Table 6. Expert Committee Architectures: Interface Complexity Metrics

Component	#Interfaces		#Operations	
	AO	Non-AO	AO	Non-AO
Kernel	4	16	68	115
Interaction	3	5	10	13
Adaptation	2	4	5	29
Autonomy	3	4	31	49
Collaboration	4	4	37	47
Mobility	3	2	20	19
Learning	2	2	6	16
Total:	21	35	177	288
Diff:	-43.2%		- 38.5%	

The use of aspects had a strong positive influence in the interface complexity of the MobiGrid architectural components, as shown in **Table 7**. For the non-AO architecture, the number of interfaces was more than 40% higher than in the AO solution. Also, the number of operations was higher in the non-AO solution (19.1%). The main reason for this result is the decrease in the number of interfaces of the `MobilityManagement` aspect. In the non-AO solution, the conventional component has interfaces to propagate mobility events relative to the initialization, migration, destruction and instantiation of agents. On the other hand, in the AO solution, the aspectual component `MobilityProtocol` crosscuts the `IReferenceObserver` interface and directly observes the events when `MobilityPlatform` notifies them. Hence, the interfaces to propagate them are not necessary.

Table 7. MobiGrid Architectures: Interface Complexity Metrics

Component	#Interfaces		#Operations	
	AO	Non-AO	AO	Non-AO
MobilityPlatform	3	4	176	185
MobilityManagement	4	9	124	155
MobilityProtocol	6	8	26	61
MobiGrid	1	3	18	24
Total:	14	24	344	425
Diff:	-41.7%		-19.1%	

8.3 Implementation Evaluation

Through the direct use of OO APIs provided by mobility platforms, several MAS classes that represent the agent types and roles need to extend the basic class for mobile agent instantiation to incorporate the mobility capabilities (e. g. `JADEAgent` class in **Section 2.3**). The use of inheritance results in code replication as well as in both code tangling and scattering; the agent basic functionalities and collaborative activities are amalgamated to mobility-specific methods (problem ①).

To solve this problem, AspectM provides the `agentInstantiation` pointcut. This pointcut is abstract and must be made concrete by AspectM users in the `Mobility` subaspect (a hot spot). In other words, an agent type constructor must be defined as the join point where the code mobility is introduced in the agent type class. The `agentInstantiation` advice then executes the instantiation protocol (a frozen spot), which instantiates a mobile agent corresponding to the application agent in the platform in use.

Before the `agentInstantiation` pointcut is made concrete, it is necessary to specify that the agent type implements the AspectM `MobileElement` interface through an intertype declaration in a `Mobility` subaspect. Thanks to this declaration, the agent type becomes a mobile element in an application; the agent can use the mobility-specific services independently on a particular platform. For example, itinerary-related attributes and methods are encapsulated in the AspectM, but their maintenance and configuration are implemented in a `Mobility` subaspect through the `MobileElement` interface methods.

As a consequence, the `MobileElement` interface also solves the problems ② and ③ described in **Section 3.2**. Using this interface, the agent and role classes do not need to hold an explicit reference to mobility elements (e.g. `itinerary`) as attributes (problem ②), or have additional methods to manage these elements (problem ③). Using the

AspectM, it becomes possible to isolate the itinerary-related and other `MobileElement` issues from the basic functionality and other system concerns.

In addition, thanks to the abstract `agentMovement` pointcut (a hot spot), AspectM users make concrete the migration points (**Section 2.2**) corresponding to a specific agent. Users can also specify the abstract mobility-specific methods, such as the `checkMobilityNeed()`, which is used to verify if an agent needs to migrate (**Section 6.2**). The migration points as well as the mobility-specific methods are called by the `agentMigration` advice, which implements the agent migration protocol (a frozen spot). This advice solves the problems ④ and ⑤, once mobility concerns are isolated from the basic functionality. The same idea is used to solve the spread of usual preconditions and postconditions that are independent on the MAS applications (problem ⑥). The instantiation, migration, arrival, and destruction advice implement together the generic mobility protocol.

A solution similar to the MobileElement interface also solves the problem ⑦ (**Section 3.2**). Through an intertype declaration, AspectM users specify in `Mobility` subaspects which elements are serializable. For example, an AspectM user must specify that his itinerary class is serializable, once itinerary attributes must be moved together with mobile agents; on the other hand, in the EC, the `Agenda` class is also specified as serializable, once an agenda object must be moved together with the chair role.

The problem ⑦ also refers to platform-specific interfaces (**Section 3.2**); the application agent types must declare and possibly specify the implementation of OO API interfaces (e. g. `AgletMobilityListener` interface in **Section 7.2.1**). As AspectM encapsulates the platform-specific issues, these interface declarations become unnecessary. AspectM users need only declare the types that implement the `MobileElement`, `Serializable`, and `MobileObject` interfaces (**Section 6.2**).

In addition, the `MobileElement` interface also solves conceptual mismatches and conflicts (**Section 3.2**). Remember that, using OO API interfaces, some types extend a platform-specific class for mobile agent instantiation (e.g. `JADEAgent`), even when agent or roles types are stationary; the introduction of mobility also causes implementation clashes, and requires the renaming of methods and changes in respective callers. AspectM solves these problems once the use of `MobileElement` interface is isolated from the basic functionalities and the other system concerns in the `Mobility` aspects.

Finally, we provide significant experimental evaluation of the benefits of the AspectM implementation in [52]. We have used the AJATO tool for measuring [17]. The comparative percentage of structural elements (classes, methods, etc.) in both OO and AO implementations of our case studies points out that the AspectM numbers are smaller by about 20%.

8.4 General Analysis

The use of the architectural modularity metrics allowed us to observe the following:

Addressing Restrictions and Conceptual Mismatches. First, after a careful joint analysis of the EC and MobiGrid architectures, we observed that both non-AO designs imposed some undesirable bidirectional couplings. For example, in the case of EC, all the "colleague" components need to inevitably contain references to the "mediator" component and vice-versa. Conversely, the AO architectural solutions for both EC and MobiGrid have reduced the overall architecture couplings by making almost all the

inter-component relationships unidirectional (aspects affect the components). This phenomenon is observed mostly from the fan-in and fan-out measures (**Section 8.2.2**). For example, the `Kernel` component has the fan-out zero in the AO version of the EC architecture, while it is 6 in the mediator-based version (**Table 4**). This coupling decrease in the EC and MobiGrid versions that follow the ArchM architecture are reflected in the AspectM use. Solutions for architectural restrictions and conceptual mismatches (**Section 6**) are achieved by replacing the direct use of OO abstractions and mechanisms, as inheritance and delegation, by AO abstractions and mechanisms.

Promoting Superior Variability and Enhanced Composability. Variability and adaptability were main driving requirements in the architecture design of EC and MobiGrid. In these systems, mobility issues should be modularized in order to promote easier variation of the mobility platforms. However, the non-modularization of mobility crosscutting concerns in the non-AO architectures hindered the satisfaction of these variability goals. This problem can be observed in the SoC measures (**Section 8.2.1**) where the results in **Tables 2 and 3** show the tangling and scattering of several concerns, such as mobility, learning, and collaboration. As a result, the plugability of elements realizing such concerns becomes cumbersome. Conversely, using ArchM facilitated the variability of the mobility platform in EC and MobiGrid systems. This is reflected in AspectM, which is generic enough and independent of specific mobility frameworks. It has some abstract intermediary classes and interfaces to bridge our framework with the chosen mobility platform (**Section 6.1.3**). Reuse of services of the mobility platform is achieved because AspectM provides a customization point to plug in a specific mobility platform. In order to change from one framework to another, MAS developers only have to perform some setups. There was no impact on the design and implementation of other agent concerns. Although the AO composition required some refactoring to expose certain join points to other aspects, it was more straightforward than the OO composition, including the scenarios involving the composition between the AspectM framework and infrastructures [5].

9 Related Work

Design support for mobile agents has been studied from different perspectives, including solutions supporting the structuring of code mobility [7, 8, 34, 35, 43, 48, 56]. Holder et al. [34] proposes a programming model for programming of the dynamic layout separately from the application's logic, including support to mobility. However, despite FarGo's programming model being very close to Java's own model, its focus is on general widely-distributed applications, not necessarily autonomous-agents-based applications [34]. Bouraqadi-Saâdani [8] presents a design of an infrastructure for applications where the mobility concerns are cleanly separated from other concerns, but his focus is mainly on strong mobility [8]. Ubayashi [56], Keeney [35] and Montanari [48] also propose a policy-based separation of concerns for code mobility. However, as the OO APIs from mobility platforms are largely in use [7, 21, 39, 43, 56], we consider that these approaches do not deal with the code mobility management at the same time they provide solutions to the usual fine-grained problems found in MAS development (**Section 3.2**). For example, the EpsilonJ reflective framework, which supports the RoleEP approach [56], supports mobility, but the agent programmers have to extend several EpsilonJ interfaces and abstract classes, which decreases the code mobility modularization (**Section 3.3**).

We introduced the use of AOP for modularization of code mobility in [45]. Subsequently, we presented the AspectM framework in [46]. Comparing to RoleEP/EpsilonJ approach [56], the use of binding-operation [56] eliminates the necessity of AOP-style weaving, and inter-type declarations in AspectJ can be replaced by adding role methods through binding-operation. However, the advice construct does not correspond to any model constructs in RoleEP. This is a weak point of RoleEP, which often causes code duplication [56]. The EpsilonJ framework is also dependent on the Aglets platform [56]. Now we present this paper as an extension of [46], including a number of improvements, especially explained. First, we provide an aspect-oriented software architecture, the ArchM, for code mobility that is independent on particular model implementations. Second, we survey the essential concerns in MAS development under the perspective of software engineering point of view. Third, our systematic analysis of the modularity problems caused by the crosscutting nature of code mobility encompasses the usual fine-grained problems found in MAS development. Fourth, we introduce the notion of architectural aspects to solve the problem of code mobility modularization. Fifth, we explain more detailed issues of the AspectM framework [46], including the dynamics and implementation of its internal elements. Finally, we present the case studies, including a number of code examples, used to evaluate the ArchM architecture, beyond the evaluation procedures, metrics and the results themselves.

10 Conclusion and Future Work

The facets of a mobility strategy should be transparent to the rest of a mobile agent system so that changes in the mobility concerns have no impact on the implementation of the other agent concerns. On the other hand, modularity occupies a pivotal position in the design of good mobility architectures: it is during architectural design that crucial modularity-related requirements in MAS such as adaptability, flexibility, reusability, maintainability, testability, etc., must be addressed. However, building modular MAS architectures is a challenging task mainly because they need to reason and make decisions with respect to a number of crosscutting architectural mobility concerns. This paper presents the ArchM, an aspect-oriented software architecture that ensures a clean modularization of the mobility concerns, a transparent introduction of code mobility into stationary agents, and an improved variability of the mobility concerns. Even though the ArchM is independent on particular mobility frameworks or applications, an ArchM implementation, the AspectM framework, was also presented to provide solutions to more fine-grained problems related to tangling and scattering of code mobility. The ArchM/AspectM allowed us not only to specify the basic mobility behaviors, but also the specification of the agent types or roles that are mobile, the declaration of the traveling circumstances, the calls to departure and the control of agent itinerary. The ArchM evaluation presented better outcomes in terms of the separation of concerns, coupling, cohesion and complexity metrics when compared to the original architectures of the case studies we have architected. A next step is to evaluate the ArchM architecture in the light of different architectural attributes other than modularity issues, such as performance and availability.

Acknowledgments. This work is partially supported by European Commission as part of the grant IST-2-004349: European Network of Excellence on AOSD (AOSD-Europe), 2004-2008. Alexander Romanovsky is supported by IST RODIN project.

References

[1] Al-NAEEM, T.; GORTON, I.; BABAR, M. A.; RABHI, F.; BENATALLAH, B. A Qualitydriven Systematic Approach for Architecting Distributed Software Applications. In Proc. of INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 244–253, NY, USA, 2005. ACM Press.

[2] AMOR, M.; FUENTES, L.; TROYA, J. Training Compositional Agents in Negotiation Protocols Using Ontologies. **Journal of Integrated Computer-Aided Engineering**, n. 2, v. 11, p. 179–194, 2004.

[3] ARIDOR, Y.; LANGE, D. Agent Design Patterns: Elements of Agent Application Design. Proceedings of the 2nd INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS (Agents '98), ACM Press, 1998, pp. 108-115.

[4] BANIASSAD, E. L. A.; CLEMENTS, P. C.; ARAÚJO, J.; MOREIRA, A.; RASHID, A.; TEKINERDOGAN, B. Discovering Early Aspects. **IEEE Software**, v. 23, 2006, p. 61-70.

[5] BARBOSA, R.; GOLDMAN, A. MobiGrid: Framework for Mobile Agents on Computer Grid Environments. Proc. of MOBILITY AWARE TECHNOLOGIES AND APPLICATIONS, 2005, Springer-Verlag, pp.147-157.

[6] BATISTA, T.; CHAVEZ, C.; GARCIA, A.; SANT'ANNA, C.; KULESZA, U.; RASHID, A.; FILHO, F. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. WORKSHOP ON EARLY ASPECTS - ASPECT-ORIENTED REQUIREMENTS ENGINEERING AND ARCHITECTURE DESIGN AT ICSE'06, Shanghai.

[7] BELLIFEMINE, F.; POGGI, A.; RIMASI, G. JADE: A FIPA-Compliant Agent Framework. Proceedings of the PRACTICAL APPLICATIONS OF INTELLIGENT AGENTS AND MULTI-AGENTS, April 1999; pp. 97-108.

[8] BOURAQADI-SAÂDANI, N. M. N.; LEDOUX, T.; SÜDHOLT, M. "A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation", INTERNATIONAL WORKSHOP ON EXPERIENCE WITH REFLECTIVE SYSTEMS, Japan, 2001.

[9] CACHO, N.; SANT'ANNA, C.; FIGUEIREDO, E.; GARCIA, A.; BATISTA, T.; Lucena, C. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proc. 5th INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, Bonn, Germany, 20-24 March 2006.

[10] CACHO, N.; BATISTA, T.; GARCIA, A.; SANT'ANNA, C.; BLAIR, G. Improving Adaptability of Reflective Middleware with Aspect-Oriented Programming. In Proc. INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, October 2006.

[11] CHAVEZ, C. **A Model-Driven Approach to Aspect-Oriented Design**. PhD Thesis, Computer Science Department, PUC-Rio, Rio de Janeiro, 2004.

[12] CHAVEZ, C.; GARCIA, A.; KULESZA, U.; SANT'ANNA, C.; LUCENA, C. Taming Heterogeneous Aspects with Crosscutting Interfaces. **Journal of the Brazilian Computer Society**, SBC, May 2006.

[13] CHITCHYAN, R. et al. A Survey of Analysis and Design Approaches. **AOSD-Europe Report D11**, May 2005.

[14] CLEMENTS, P.; KAZMAN, R.; KLEIN, M. **Evaluating Software Architectures: Methods and Case Studies**. Addison-Wesley, Boston, MA, USA, 2002.

[15] DAMASCENO, K.; CACHO, N.; GARCIA, A.; ROMANOVSKY, A.; LUCENA, C. Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case. In 5th INT. WORKSHOP ON SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS AT ICSE 2006, Shanghai, China, 2006.

[16] DELOACH, S., WOOD, M., SPARKMAN, C. Multiagent Systems Engineering. **International Journal of Software Engineering and Knowledge Engineering**, 11(3):231--258, 2001.

[17] FIGUEIREDO, E.; GARCIA, A.; LUCENA, C. AJATO: an AspectJ Assessment Tool. Proc. of the ECOOP.06, Demo Session, Nantes, France, July 2006.

[18] FILHO, F.; CACHO, N.; FERREIRA, R.; FIGUEIREDO, E.; GARCIA, A.; RUBIRA, C. Exceptions and Aspects: the Devil is in the Details. Proceedings of the 14th INTERNATIONAL CONFERENCE ON FOUNDATIONS ON SOFTWARE ENGINEERING, Portland, USA, November 2004.

[19] FILHO, F.; RUBIRA, C.; FERREIRA, R.; GARCIA, A. Aspectizing Exception Handling: A Quantitative Study. In **Advances in Exception Handling Techniques**, LNCS 4119, September 2006.

[20] FIPA. Foundation for Intelligent Physical Agents. http://www.fipa.org.

[21] FUGGETTA, A.; PICCO, G.; VIGNA, G. Understanding Code Mobility. **IEEE Transactions on Software Engineering**, v.24, n.5, p.342-361, 1998.

[22] GARCIA, A. et al. Engineering Multi-Agent Systems with Aspects and Patterns. **Journal of the Brazilian Computer Society**, n.1, v.8, Jul.2002, p. 57-72.

[23] GARCIA, A.; SANT'ANNA, C.; CHAVEZ, C.; SILVA, V.; LUCENA, C.; STAA, A. v. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: C. LUCENA et al, eds. **Software Engineering for Multi-Agent Systems II**. Springer-Verlag, LNCS 2940, January 2004.

[24] GARCIA, A. **From Objects to Agents: An Aspect-Oriented Approach**. PhD Thesis, Computer Science Department, PUC-Rio, April 2004.

[25] GARCIA, A.; LUCENA, C.; COWAN, D. Agents in Object-Oriented Software Engineering. **Software: Practice & Experience**, Elsevier, Volume 34, Issue 5, May 2004, pp. 489 - 521.

[26] GARCIA, A.; KULESZA, U.; SANT'ANNA, C.; LUCENA, C. The Mobility Aspect Pattern. Proc. of the 4th LATIN-AMERICAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMMING, August 2004, Fortaleza, Brazil.

[27] GARCIA, A.; KULESZA, U.; SARDINHA, J.; MILIDIÚ, R.; LUCENA, C. The Learning Aspect Pattern. In Proc. of the 11th CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, Monticello, USA, September 2004.

[28] GARCIA, A.; SANT'ANNA, C.; FIGUEIREDO, E.; KULESZA, U.; LUCENA, C.; Staa, A. Modularizing Design Patterns with Aspects: A Quantitative Study. **Transactions on Aspect-Oriented Software Development**, Springer-Verlag, Lecture Notes in Computer Science, pp. 36 - 74, Vol. 1, No. 1, February 2006.

[29] GARCIA, A.; LUCENA, C. Taming Heterogeneous Agent Architectures with Aspects. **Communications of the ACM**, July 2006.

[30] GARCIA, A. et al. On the Modular Representation of Architectural Aspects. Proc. of the 3rd. EUROPEAN WORKSHOP ON SOFTWARE ARCHITECTURE, Nantes, France, September 2006.

[31] GOLDCHLEGER, A. et al. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. **Concurrency and Computation: Practice & Experience**, v. 16, p. 449-459. March, 2004.

[32] HANENBERG, S.; UNLAND, R.; SCHMIDMEIER, A. AspectJ Idioms for Aspect-Oriented Software Construction. In Proceedings of the 8th EUROPEAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMMING AND COMPUTING, Irsee, Germany, June 2003.

[33] HARRISON, C.; CHESS, D.; KERSHENBAUM, A. **Mobile Agents: Are they a good idea?** Technical report, IBM, March 1995.

[34] HOLDER, O.; BEN-SHAUL, I.; GAZIT, H. "Dynamic Layout of Distributed Applications in FarGo", 21st INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ACM Press, 1999.

[35] KEENEY, J.; CAHILL, V. "Chisel: a policy-driven, context-aware, dynamic adaptation framework", **Policies for Distributed Systems and Networks**, 2003.

[36] KENDALL, E. A. Role Model Designs and Implementations with Aspect-oriented Programming. In OOPSLA '99: Proc. of the 14th ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, p. 353–369, NY, USA, 1999. ACM Press.

[37] KICZALES, G. et al. "Aspect-Oriented Programming". Proc. of the ECOOP'97, LNCS (1241), Springer-Verlag, Finland, June, 1997.

[38] KICZALES, G. et al. "An Overview of AspectJ". Proc. of ECOOP'2001, Budapest, Hungary, 2001.

[39] KINIRY, J.; ZIMMERMAN, D.: A Hands-On Look at Java Mobile Agents, **IEEE Internet Computing**, vol.1, No.4, 1997.

[40] KRECHETOV, I.; TEKINERDOGAN, B.; GARCIA, A.; CHAVEZ, C.; KULESZA, U. Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. 8th INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING, March 20-24, 2006, Bonn, Germany.

[41] KULESZA, U.; GARCIA, A.; LUCENA, C. Towards a Method for the Development of Aspect-Oriented Generative Approaches. WORKSHOP ON EARLY ASPECTS AT OOPSLA'04, November 2004, Vancouver, Canada.

[42] KULESZA, U.; SANT'ANNA, C.; GARCIA, A.; COELHO, R.; STAA, A. v.; LUCENA, C. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. Proceedings of the 9th INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, Philadelphia, USA, September 2006.

[43] LANGE, D.; OSHIMA, M. **Programming and Deploying Java Mobile Agents with Aglets**. Addison-Wesley, 1998.

[44] LIPPERT, M.; LOPES, C. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In: Proc. of INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Limerick, Ireland, may 2000, pp. 418–427.

[45] LOBATO, C; GARCIA, A.; ROMANOVSKY, A.; SANT'ANNA, C.; KULESZA, U.; LUCENA, C. Mobility as an Aspect: The AspectM Framework. Proc. of WASP At SBES'04, Brasilia, Brazil, 2004.

[46] LOBATO, C.; GARCIA, A.; LUCENA, C.; ROMANOVSKY, A. A Modular Implementation Framework for Code Mobility. 3rd IEE MOBILITY CONFERENCE 2006, 25-27 October 2006, Bangkok, Thailand.

[47] LOUGHRAN, N. et al. **A Domain Analysis of Key Concerns - Known and New Candidates**. Technical Report AOSD-Europe Deliverable D43, 2006.

[48] MONTANARI, R.; TONTI, G.; STEFANELLI, C. "Policy-based separation of concerns for dynamic code mobility management". COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2003.

[49] PACE, A.; TRILNIK, F.; CAMPO, M. Assisting the Development of Aspect-based MAS using the SmartWeaver Approach. In **Software Engineering for Large-Scale Multi-Agent Systems**, LNCS 2603, March 2003.

[50] RASHID, A.; GHITCHYAN, R. Persistence as an Aspect. In: Proceedings of the 2nd INTERNATIONAL CONFERENCE ON ASPECT-OORIENTED SOFTWARE DEVELOPMENT, USA, March 2003.

[51] SANT'ANNA, C.; GARCIA, A.; CHAVEZ, C.; STAA, A.; LUCENA, C. On the Reuse and Maintenance of Aspect-Oriented Software: An Evaluation Framework. In Proc. of the XVII BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p.19–34, October 2003.

[52] SANT'ANNA, C.; LOBATO, C.; KULESZA, U.; CHAVEZ, C.; GARCIA, A.; Lucena, C. On the Quantitative Assessment of Modular Multi-Agent Architectures. Proceedings of NETOBJECTDAYS, September 2006, Germany.

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

[54] SOARES, S.; LAUREANO, E.; BORBA, P. Implementing Distribution and Persistence Aspects with AspectJ. In: Proceedings of the ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2002, pp. 174-190.

[55] TAMAI, T.; UBAYASHI, N.; ICHIYAMA, R. An Adaptive Object Model with Dynamic Role Binding. ICSE 2005: 166-175.

[56] UBAYASHI, N.; TAMAI, T. Separation of Concerns in Mobile Agent Applications. Proceedings of INTERNATIONAL CONFERENCE REFLECTION 2001, LNCS 2192, Kyoto, Japan, September 2001, Springer, pp. 89-109.

[57] WEISS, G. et al. Capturing Agent Autonomy in Roles and XML. In Proc. INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTI-AGENT SYSTEMS, p.105–112, NY, USA, 2003.

[58] WHITE, J. E. **Mobile Agents**. Tec. Report, General Magic, Los Angeles, 1995.

[59] ZAMBONELLI, F., JENNINGS, N., WOOLDRIDGE, M. Organizational Abstractions for the Analysis and Design of Multi-agent Systems. In: CIANCARINI, P., WOOLDRIDGE, M., eds. **Agent-Oriented Software Engineering**, Springer-Verlag (2001).