On the Quantitative Assessment of Modular Multi-Agent System Architectures

Claudio Sant’Anna\textsuperscript{1,2}, Cidiane Lobato\textsuperscript{2}, Uirá Kulesza\textsuperscript{2}
Alessandro Garcia\textsuperscript{1}, Christina Chavez\textsuperscript{3}, Carlos Lucena\textsuperscript{2}

\textsuperscript{1} Computing Department, Lancaster University - UK
\{c.santanna,a.garcia\}@lancaster.ac.uk
\textsuperscript{2} Computer Science Department, PUC-Rio - Brazil
\{claudios,cidiane,uiru,lucena\}@inf.puc-rio.br
\textsuperscript{3} Computer Science Department, Federal University of Bahia - Brazil
flach@dcc.ufba.br

Abstract. A number of concerns in multi-agent systems (MAS) have a broadly-scoped impact on the system architectural decomposition, which in turn hinder the design of modular MAS architectures. These concerns inherently crosscut the boundaries of several architecture elements, such as components, connectors, and their interfaces. Typical examples of crosscutting concerns in MAS architecture include learning, mobility, coordination, context-awareness, and error handling. Nowadays there are some architectural proposals that envisage an emerging aspect-oriented architectural pattern as a potential solution to address modularity shortcomings of conventional architectural patterns for MAS designs. However, little effort has been dedicated to effectively assess when and which of these existing architectural solutions promote in fact superior modularity in the presence of typical crosscutting MAS concerns. This paper presents a quantitative comparison of aspect-oriented and traditional MAS architectures with respect to their support for promoting enhanced modularity in the presence of architectural crosscutting concerns in MAS design. Our evaluation used two medium-sized MAS applications and was centered on fundamental modularity attributes, such as separation of architectural concerns, architectural coupling, component cohesion, and interface simplicity.

1 Introduction

Software architecture is a fundamental element of multi-agent systems (MAS). Architects strive to develop reusable and adaptable MAS architectures that are resilient in face of changes especially for systems in volatile business domains such as eCommerce, banking, and telecommunications. However, in order to be effectively reusable and adaptable, MAS software architectures must be modular. It requires from architects to make proper early design decisions, such as the adequate selection and composition of architectural patterns in order to achieve satisfactory separation of concerns. Modular MAS architectures also demand
the careful employment of well-designed component interfaces, reduction of architectural coupling, maximization of components’ cohesion, and the like. These all concern places where we explicitly consider elementar requirements in our MAS designs, such as reusability, adaptability, flexibility, maintainability, and so forth.

In fact, conceiving modular software architectures for complex MAS is not a trivial task mainly due to a number of recurring widely-scoped MAS concerns and application-specific concerns. A number of these concerns have a crosscutting impact on the MAS architectural decomposition, and systematically affect the boundaries of several architectural elements, such as components and their interfaces [11, 24, 25, 19]. Typical examples of crosscutting concerns in MAS architecture include learning [23], mobility [22, 47], autonomy [17, 48], coordination [32, 3], context-awareness [40], and error handling [13]. The problem is that to build flexible MAS applications and frameworks, many of those typical crosscutting features should be modularized in order to be separated and optionally exposed as architectural variabilities in the MAS architecture decomposition. This scenario becomes even more challenging when there is a need for reusing and composing MAS frameworks (and platforms), in which their architectures partially addresses each of those broadly-scoped MAS concerns and, at the same time, embody a number of crosscutting architectural decisions [34].

In this context, there a number of architectural patterns proposed to build modular MAS [31, 10, 45]. For example, Kendall and colleagues [31] propose a specialization of the layered architecture in order to provide a clear separation of MAS concerns as layers, with narrow interfaces between adjacent layers. In previous work [24, 17], we have defined an aspect-oriented architectural pattern for a similar purpose. There are also some existing MAS frameworks and applications that follow a similar aspect-oriented architecture design [3, 40, 43, 41]. However, little effort has been dedicated to systematically assess to what extent this new architectural solution promotes in fact superior modularity in the presence of typical crosscutting concerns in MAS architecture designs. Without a clear evidence of each solution strengths and drawbacks, modular MAS architects are not guided on when to use each of the available architecture patterns and how to compose them.

This paper presents two complementary case studies where we have quantitatively evaluated the degree to which the aspect-oriented pattern scales up to promote improved architecture modularity when compared to other conventional patterns, such as mediator-based and publisher-subscriber styles. We have architected and implemented two medium-sized MAS with similar driving modularity-related requirements, from different domains, based on distinct MAS platforms and frameworks, and with distinct exploitation of typical MAS properties. Our evaluation is based on architectural metrics, which are rooted at fundamental modularity principles, such as separation of concerns, narrow interfaces, low architectural coupling, high component cohesion, and composition simplicity. This empirical assessment is one step ahead with respect to a first
similar empirical study that we have previously carried out to compare aspect-oriented and object-oriented implementations for the same MAS application [25].

The paper is organized as follows. Section 2 presents conventional patterns for architecting MAS and the application of an aspect-oriented architecture pattern for MAS design. Section 3 describes the two case studies, and our used assessment procedures and metrics. Section 4 presents and discusses the architecture measurement results. Section 5 introduces a more general analysis and related work. Section 6 presents concluding remarks.

2 Architectural Patterns and Modular MAS

Every application, in general, requires a recurring architectural pattern which is the most appropriate to satisfy its highest-priority requirements, while also promoting architecture modularity. Although of course there is no single architectural pattern that is the most suitable for all MAS applications, modularity is a stringent software engineering property that underlie almost all architectural decisions in MAS designs. Architectural patterns provide guidelines to build modular software architectures in the sense that they prescribe a set of architectural roles, their interactions, and associated constraints. All these architectural rules are typically targeted at preserving basic design principles, such as simple interfaces, low coupling, and so forth.

A number of architectural patterns [7] have been proposed for the modular design of MAS architectures. Section 2.1 revisits some of those architectural solutions and briefly discusses architectural crosscutting in MAS decompositions. This discussion serves as a twofold purpose: it describes the consequences of not having explicit architectural support for the modularization of crosscutting concerns in MAS designs, and works as a motivation for outlining an aspect-oriented architectural pattern [24] for MAS in Section 2.2.

2.1 Conventional Architectural Patterns for MAS

There are some architectural approaches [2, 10, 31] that promote enhanced modularity of architectural MAS concerns. They rest on traditional architectural patterns, such as the Layers pattern [31], Mediator pattern [10], the Reflection pattern [2], or the Publisher-Subscriber pattern [29]. For instance, Kendall et al [31] propose the Layered Agent architectural pattern with multiple layers for the separate representation of agent concerns (Figure 1 (a)). The layered architecture establishes a composition style in which all the interactions feature a two-way information flow and only subjacent layers communicate with each other.

The use of mediators [10] is also a common architectural approach to address the composition of agent concerns that interact in multiple ways. Composition patterns, such as the Mediator pattern [16] and the Composite pattern [16], are mediator-oriented solutions. They provide means of allowing integration of agent properties using a central component, the mediator (Figure 1 (b)).
pattern, for instance, defines a mediator component that encapsulates how a set of components, the colleagues, interact with each other. This solution promotes loose coupling by keeping components from referring to each other explicitly, and it lets the agent developers vary their interaction independently. The Skeleton Agent framework [10] realizes a mediator-based architecture by implementing the Composite pattern.

Our previous empirical investigation [25] has identified that implementation following conventional architectural patterns may hinder the development of modular agent architectures. The main reason is that such implementations fail to address the modularization of crosscutting MAS concerns. The crosscutting manifestation leads to two major problems – scattering and tangling [46] – which are fundamental anti-modularity factors in MAS architectures. Scattering in MAS architectures is the occurrence of architectural elements, such as interfaces, that belong to one agent concern in architectural components encapsulating other agent concerns [24]. For example, the interaction-related interfaces are scattered over the agent architectures in Figure 1. Architectural tangling is the mix of multiple agent concerns together in the same module [24]. For instance, tangling is evident in the agent kernel of Figures 1 (a) and 1 (b) as both are implementing interfaces associated with different agent concerns.

2.2 An Aspect-Oriented Pattern for MAS Architectures

In order to address such modularity breakdowns in MAS architectures, an aspect-oriented architectural pattern [24], and a number of aspectual MAS implementation frameworks [19, 3, 43] that realize this pattern have been defined. Table 1 summarizes the main roles defined by the aspect-oriented architectural pattern, their respective responsibilities, and interaction constraints. Section 3 will describe two different instantiations of the aspect-oriented pattern in the context of our case studies. This pattern documents the fundamental attributes of an aspect-oriented architectural design. Architectural aspects (or aspectual components) and aspectual interaction rules are exploited at the architectural level to capture the crosscutting architectural concerns that are hard to modularize with existing architectural abstractions defined by other patterns. An architectural aspect is used to represent each crosscutting concern as an individual component at an early stage of design [24].

In MAS design, architectural aspects should be used to modularize typical crosscutting MAS properties and to separate them from other architectural components. The architectural components expose through their interfaces certain join points that are used by the aspects. The goal of aspectizing MAS architectures is to allow the association of crosscutting agenthood properties with the non-crosscutting basic functionality at certain join points. The key abstraction to enable adjustable compositions is the notion of crosscutting interfaces (Table 1), which are modularity abstractions attached to the architectural aspects. A crosscutting interface is different from a conventional module interface in the sense that the latter essentially provides services to other components. Crosscutting interfaces provide services to the system, but also specify when and how
an aspect affects other architectural components. The relationship between an architectural aspects and other components by means of crosscutting interfaces is called **crosscutting relationship**. **Architectural join points** are the elements in an architecture specification that can be affected by a certain aspect.

Opposed to interfaces in traditional architecture styles, crosscutting interfaces flexibly determine which architectural join points the architectural aspect of a software agent will be connected to. With this dependency inversion, crosscutting interfaces overcome the problems associated with the rigidness implicit in traditional architectural styles (Figure 1). Each agent’s architectural aspect can be more flexibly composed with the agent kernel and with any other agent aspects depending on the requirements of a specific agent architecture. Each of the architectural aspects is related to more than one component, representing the crosscutting nature of agent properties in complex architectures.

### 3 Experimental Settings

This section describes the configuration of our empirical assessment. We have performed a pair-wise comparison about the modularity of an aspect-oriented (AO) architecture and a second non-aspectual (non-AO) architectural solution in the context of two MAS case studies, which are described in Section 3.1 and Section 3.2. The first case study involved the comparative evaluation of a mediator-based and aspect-oriented architecture for a MAS framework [19], called AspectT, which supports the development of applications with heterogeneous agent architectures. The second study encompassed a publisher-subscriber...
Table 1. The Aspect-Oriented Architectural Pattern: Architectural Elements, their Responsibilities and Interactions.

<table>
<thead>
<tr>
<th>Architectural Role</th>
<th>Responsibilities and Constraints</th>
<th>Interaction Constraints</th>
</tr>
</thead>
</table>
| Base Components (and Interfaces) | - modularize services that realize a non-crosscutting concern  
                                  - provide the architectural join points to aspects | - can be affected by one or more crosscutting interfaces  
                                  - access services available in other conventional interfaces |
| Architectural Aspects (or Aspectual Components) | - encapsulate an architectural crosscutting concern  
                                              - contain at least one crosscutting interface  
                                              - may implement conventional interfaces | - affect base components and other aspects through its crosscutting interfaces  
                                              - can be directly associated with other aspects |
| Crosscutting Interfaces | - provide crosscutting services and specify when and how each of them affects other interfaces | - are allowed to affect both conventional and other crosscutting interfaces  
                                              - affect base interfaces using aspectual composition operators, such as before, after, and around |

[4] and an aspect-oriented version [39] of the MobiGrid architecture [4], used to develop mobile agents in Grid environments. Other architectural patterns were also instantiated and composed in all the system versions, such the client-server. However, the aforementioned ones are the heart of the architecture design and, therefore, we concentrate our attention on them.

Both systems were ideal for our experimental investigation due to 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 [4]. Third, a preliminary qualitative assessment focused at the implementation level has been recently conducted and reported [17, 21, 39]. It has allowed us to supplement the qualitative focus on separation of concerns of our preliminary studies with both broader quantitative analysis and a systematic investigation about the scalability of aspect-oriented architectures in distinct MAS application scenarios.

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 MAS platforms and frameworks, such as JADE [6], Aglets [37], and TSpaces [38]. Sections 3.1 and 3.2 focus on describing the main AO and non-AO architectural choices for both
AspectT and MobiGrid case studies; each design choice for all the investigated solutions have been deeply discussed and documented elsewhere [17, 19, 21, 4, 39]. Section 3.3 presents our evaluation steps and the architectural metrics applied.

3.1 The AspectT Architecture

We developed an aspect-oriented agent framework [19], called AspectT, which defines an architecture for implementing different kinds of agents, such as, information and user agents. This framework has been implemented using the AspectJ and Java programming languages. We have used this framework in the implementation of two case studies: (i) Portalware – a web-based system for the development of e-commerce portals [19]; and (ii) ExpertCommittee – a conference management MAS [21]. AspectT was structured following the aspect-oriented pattern for MAS architectures detailed in Section 2.2. Figure 2 shows the framework architecture. It defines a set of components playing the roles of architectural aspects that address different crosscutting agent properties, such as interaction, autonomy, mobility, and learning. Following we describe the main components of the AO architecture and their respective relationships.

![Fig. 2. The AspectT Architecture.](image_url)

The aspect-oriented architecture has the **Kernel** component as a central element. This component defines two interfaces: (i) **KnowledgeUpdating** – used to update the agent knowledge (belief, goal and plan); and (ii) **Services** – which allows to expose the agent services. A set of aspectral components are used to address different crosscutting agent features. Each of them either introduces new behavior in other components or refines the components’ behavior by observing...
specific service execution. The interaction aspect is used to modularize the crosscutting impact on the use of communication architectures, such as JADE [37]. The Interaction aspectual component specifies crosscutting interfaces for message receiving (MessageReception) and for message sending (MessageSending).

The Adaptation component intercepts the MessageReception interface of the Interaction component by means of the KnowledgeAdaptation crosscutting interface to update the agent beliefs when new external messages are received. It also defines the BehaviorAdaptation interface to instantiate specific plans when the agent needs to achieve any goal. The Autonomy aspectual component defines three crosscutting interfaces: (i) GoalCreation – cuts across the Interaction component to create new reactive goals when new external messages are received; (ii) DecisionMaking – introduces in the Kernel component the behavior of creation of proactive goals; and finally (iii) ExecutionAutonomy – associates the agent with its own thread of control and also makes it possible the concurrent execution of agent plans. The Collaboration, Mobility and Learning components encompass crosscutting agent properties that are necessary only on specific agent architectures. The Collaboration component contains two interfaces: (i) Knowledge – introduces new knowledge associated with roles to be played by the agent; and (ii) Binding – affects specific services from the Kernel component in order to instantiate new roles and attach them to the agent according to certain conditions.

The Mobility aspect is used to overcome the crosscutting nature of mobility concerns by directly using existing mobility platforms. The Mobility component uses the Travel interface to introduce mobility capacities in the agent and to determine the execution points in which the agent can be moved. Finally, the Learning component is responsible for collecting information to execute its learning algorithms (InformationGathering interface). It also introduces new learning-specific knowledge associated with these algorithms (LearningKnowledge interface). The AspectT framework has been developed as an alternative to an equivalent object-oriented mediator-based architecture, presented in Section 2.2. The combination of these two architectural styles defines a central component which mediates all the communication between the other ones. The Kernel component plays this central role.

3.2 The MobiGrid Architecture

Our second case study was the MobiGrid framework [4], which is a mobile agent system within a grid environment project called InteGrade [27]. 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 [37], and follows a publisher-subscriber architecture pattern.

Due to the high coupling between the Aglets underlying model and the MobiGrid mobile agents, we have decided to reengineer the MobiGrid architecture
taking into account the following requirements: (i) to modularize the MobiGrid mobility concerns, that is, to promote an explicit separation between the crosscutting mobility concern and other non-crosscutting MobiGrid concerns; and (ii) to enhance the MobiGrid variability in terms of a flexible choice of distinct mobility platforms to be used (e.g., Aglets [37], JADE [6], etc.).

In the reengineering process, we have generated two versions of the MobiGrid architecture. In a first moment, we decided not to use the concept of crosscutting interfaces (Section 2.2); our goal was then to assess the applicability of conventional architectural components and interfaces in order to explicitly separate the mobility concern. After that, we provided an alternative version of this reengineering using aspectual components and crosscutting interfaces [39]. In both solutions, the separation of the mobility concerns and the integration between MobiGrid and distinct mobility platforms respectively resulted in the conception of two architectural components: the MobilityProtocol and the MobilityManagement.

Figures 3 and 4 illustrate the two versions resulted from the MobiGrid architecture reengineering: the non-aspectual architecture and the aspect-oriented one. The former follows the publisher-subscriber architectural pattern; the latter is based on the aspect-oriented architectural pattern (Section 2.2). The architectural designs presented in both figures respectively follow a simplified UML notation (Figure 3), and the AOGA notation [35] (Figure 4). Note that the main difference between the proposed architectures is the use of crosscutting interfaces and aspects for designing the two new components defined in the reengineered MobiGrid Architecture (Figure 4).

Despite this difference, in both cases, the MobiGrid architecture is composed of four kinds of components: (i) the MobiGrid component, which modularizes the basic concerns of an agent-based application; (ii) the MobilityProtocol component, which modularizes the mobility protocol execution – i.e., the instantiation, migration, remote initialization, and destruction of MobiGrid agents; (iii) the MobilityManagement component, which provides a flexible integration between MobiGrid and distinct mobility platforms; and (iv) the MobilityPlatform, which represents a specific mobility platform being used, such as Aglets [37]. In both architectures, the main purpose of the MobilityProtocol component is the explicit separation of the mobility concerns from the MobiGrid component. In addition, the MobilityManagement component connects the MobiGrid with the MobilityPlatform component, which modularizes and externalizes the platform services.

The IMobileAgentProtocol and IReferenceMobileAgent interfaces play a central role in the design of both non-AO and AO architectures. The IMobileAgentProtocol is the interface that delegates to the IReferenceMobileAgent the mobility services invoked by the MobiGrid; this delegation is independent from platform-specific issues. The IMobileAgent is the interface which is responsible for delegating to a specific platform the invoked services; to do that, it uses the IPlatformServices interface provided by
the MobilityPlatform component. The mobility services include the access to the agent ids, context ids, mobile agent lifecycle, messaging, and so on.

The AO architecture in Figure 4 uses the crosscutting interface abstraction (Section 2.2) to make it possible a clean modularization of the mobility concern in the MobiGrid. The MobilityProtocol component now implements a generic mobility protocol in order to prevent the explicit invocations of the mobility services by the MobiGrid component. Such explicit invocations happen in the non-AO architecture due to the interaction constraints imposed by the publisher-subscriber pattern. In other words, we invert the way in which access to the mobility services is typically designed in mobile agent systems. To do that, the IMobileElement crosscutting interface is used to determine when and how a mobile agent is instantiated on a platform to represent a specific agent on the MobiGrid. This interface also triggers the agent migration to other environments, since the mobile agent may have to migrate whenever elements of the MobiGrid are called or executed. That is, the IMobileElement interface is used to affect well-defined mobility join points in order to determine when MobiGrid agents should move. Thus, the IMobileElement interface allows an explicit separation of mobility issues from the other MobiGrid non-crosscutting concerns.

Other interfaces are used in both architectures in order to maintain a flexible integration between MobiGrid and distinct mobility platforms; they make information relative to the mobile agent lifecycle available to the MobiGrid component. However, they externalize the mobility concern through the MobilityProtocol component in different ways. In Figure 4, the IReferenceObserver interface crosscuts join points as calls of mobility platform services in order to maintain a consistence between the platform runtime and the MobiGrid. This is possible because the IReferenceObserver interface is also affected by the IInstantiationEvent, IMigrationEvent, IInitializationEvent, and IDestructionEvent interfaces, which respectively...
allow the agent instantiation, departure, arrival, and destruction handling in the MobilityProtocol component.

A different architectural scheme is used to propagate the platform events in Figure 3. The IReferenceObserver interface still observes these events, but now following the publisher-subscriber architectural pattern. Additional required interfaces are necessary in order to delegate platform events to the MobilityProtocol component: IMigrationPropagator, IInstantiationPropagator, IInitializationPropagator, and IDestructionPropagator. These interfaces are respectively connected to the IInstantiationEvent, IMigrationEvent, IInitializationEvent, and IDestructionEvent interfaces in order to allow event handling in the MobilityProtocol component. Finally, note that in both AO and non-AO architectures the MobilityManagement component realizes a conventional interface called the IReferenceTable. This interface is used to abstract the context and messaging services provided by different platforms.

3.3 Evaluation Procedures and Assessment Metrics

Our evaluation has strictly focused on the evaluation of architecture artifacts, since we are concerned with both: (i) understanding the suitability of existing MAS architecture-level solutions in order to address the modularity problems associated with widely-scoped crosscutting properties, and (ii) investigating to what extent the crosscutting nature of certain MAS properties entail design anomalies visible earlier at the architecture stage. The used architecture descriptions were based on conventional and aspect-oriented component-and-connector models [35, 5, 18].

We have used a suite of architectural metrics (Table 2) to support modularity evaluation of the investigated software architectures (Sections 3.1 and 3.2).
have not used conventional architectural assessment methods because they traditionally focus either on the architecture coverage of scenarios described in the requirements specification [12], 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. The outcomes of our analyses can be used in conjunction with other architectural assessment methods, such as ATAM, for performing a trade-off evaluation with respect to other architectural qualities, such as performance and availability.

A discussion about each of those architectural metrics is out of the scope of this work. Table 2 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, architectural coupling, component cohesion and interface complexity. We have already used similar categories of metrics [17, 42] for evaluating aspect- and object-oriented designs in a number of systematic case studies [26, 9, 15, 36, 8] not related to multi-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 2 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.

The metrics of separation of concerns (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 the cohesion is.

In order to proceed with the measurement of separation of concerns, there is an architecture shadowing process in which the architect must assign every component element (interface, operation and parameter) to one or more concerns. The chosen architectural concerns to be assessed are related to the driven architectural issues that should be modularized in each system. For example, in the AspectT case, we have shadowed the architecture artifacts with respect to the kernel, adaptation, interaction, autonomy, collaboration, mobility, and learning concerns because these are the properties that should be adaptable, reusable, and easily (un)plugged from the system. As in the MobiGrid system the design
Table 2. Architectural Metrics Suite.

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

was much more focused on modularizing specific mobility issues (Section 3.2), we treated each of the core mobility features (mobility platform, mobility protocol, and mobility management) and the MobiGrid application as driving concerns in order to investigate their associated crosscutting structures in both architectural solutions. After the shadowing of the architecture models, the data of the separation of concerns metrics (CDAC, CDAI, and CDAO) was manually collected.

4 Empirical Results

This section presents the results of the measurement process. The data have been collected based on the set of defined measures (Section 3.3) in the two case studies. The presentation is broken in three parts. Section 4.1 presents the evaluation results for the separation of architectural concerns. Section 4.2
presents the results for the coupling and cohesion metrics. Section 4.3 presents the results for the interface complexity metrics. We present the results by means of tables that put side-by-side the values of the metrics for the AO and non-AO architectures of each system.

4.1 Separation of Architectural Concerns

In the quantitative evaluation of the AspectT framework, the data collected for both AO and mediator-based architectures shows favourable results for the AO version for most of the metrics used. Table 3 presents the complete data collected for both AspectT 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 AspectT architectures. These metrics count the total number of components, interfaces and operations dedicated to implement a concern (Section 3.3).

We can observe significant differences between the AO and non-AO versions for all the SoC metrics. Table 3 shows that the mediator-based 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. It occurs because the Kernel component plays the mediator role and, as a consequence, propagates information relative to every concern to the other “colleague” components. 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 3 that the AO version requires fewer interfaces (CDAI metric) and operations for most of the system concerns with 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 3, 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 mediator 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 (see Figure 1) with their associated operations.

Table 4 shows the results for the three SoC metrics for the MobiGrid architectures. As discussed in Sections 3.1 and 3.2, the MobiGrid application and agent
mobility issues are the two architectural concerns of interest in the MobiGrid system. The AO architecture performed better than the publisher-subscriber one in terms of SoC. As shown in Table 4, 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 two mobility-related interfaces – IMobilityLifeCycleObserver and IMobilityLifeCycleSubscriber – for explicitly handling of mobility life-cycle 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.

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. This will be further discussed in subsection 4.3. The aforementioned absence of mobility interfaces in the MobiGrid component also contributes for that difference.

### Table 4. MobiGrid Architectures: Separation of Concerns Measures.

<table>
<thead>
<tr>
<th>Concern</th>
<th>#components (CDAC)</th>
<th>#interfaces (CDAI)</th>
<th>#operations (CDAO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobility</td>
<td>3</td>
<td>13</td>
<td>326</td>
</tr>
<tr>
<td>Application</td>
<td>1</td>
<td>1</td>
<td>18</td>
</tr>
</tbody>
</table>
4.2 Architectural Coupling and Component Cohesion

Tables 5 and 6 present the results for architectural coupling and component cohesion metrics considering, respectively, the AspectT and MobiGrid architectures. As in subsection 4.1, the tables in this subsection and in subsection 4.3 put side-by-side the metrics values for the AO and non-AO architectures. However, since the values here are for each component (component viewpoint), the bottom of the tables also provide the total values (sum of all the component measures) that represent the results for the overall architecture viewpoint. Therefore, rows labelled “Total” indicate the tally for the system architecture, while rows labeled “Diff” indicate the percentual 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 5, there is an expressive coupling increase in the non-AO AspectT architecture considering the number of requiring components (Architectural Fan-in metric). The fan-in is 12 in the mediator-based architecture, while it is 9 in the AO architecture, representing a difference of 25% in favour 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. With respect to the architectural fan-out, the measures did not show an expressive difference from system viewpoint; the difference was lower than 10%.

As stated in Section 3.3, 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 mediator-based 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.

Table 5. AspectT Architectures: Coupling and Cohesion Measures.

<table>
<thead>
<tr>
<th>Component</th>
<th>Architectural Fan-Out AO</th>
<th>Architectural Fan-Out Non-AO</th>
<th>Architectural Fan-In AO</th>
<th>Architectural Fan-In Non-AO</th>
<th>#Concerns AO</th>
<th>#Concerns Non-AO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Interaction</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Adaptation</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Autonomy</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Collaboration</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mobility</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Learning</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total:</td>
<td>11</td>
<td>12</td>
<td>9</td>
<td>12</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>Diff:</td>
<td>-8.3%</td>
<td>-25.0%</td>
<td>-25.0%</td>
<td>-46.2%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 6). 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 showed 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.

Similarly to the AspectT case, the cohesion measures in the MobiGrid architectures pointed out a difference in favour 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. The IMobilityLifeCycleObserver and IMobilityLifeCycleSubscriber interfaces contain operations related to the mobility concern, which reduce the cohesion of the component; in fact, these operations have a purpose different from the main purpose of the component. On the other hand, these interfaces are not necessary in the AO solution and, as a consequence, the MobiGrid component is not affected by the mobility concern and is, therefore, entirely dedicated to its main concern.

Table 6. MobiGrid Architectures: Coupling and Cohesion Measures.

<table>
<thead>
<tr>
<th>Component</th>
<th>Architectural Fan-Out</th>
<th>Architectural Fan-In</th>
<th>#Concerns (Lack of Cohesion)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AO</td>
<td>Non-AO</td>
<td>AO</td>
</tr>
<tr>
<td>MobilityPlatform</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MobilityManag.</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MobilityProtocol</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MobiGrid</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>3</strong></td>
<td><strong>6</strong></td>
<td><strong>3</strong></td>
</tr>
<tr>
<td><strong>Diff:</strong></td>
<td><strong>-50.0%</strong></td>
<td><strong>-50.0%</strong></td>
<td><strong>-20.0%</strong></td>
</tr>
</tbody>
</table>

4.3 Interface Complexity

Tables 7 and 8 show the results for the interface complexity metrics for the AspectT and MobiGrid architectures, respectively. Regarding the AspectT system
(Table 7), 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 favours the AO version. This difference is mainly due to the additional interfaces of the Kernel component, but it is also thanks to the values collected for other components. The increase in the number of interfaces metric for the mediator version is also reflected in the number of operations. Table 7 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 (Section 4.1).

<table>
<thead>
<tr>
<th>Component</th>
<th>#Interfaces</th>
<th>#Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AO</td>
<td>Non-AO</td>
</tr>
<tr>
<td>Kernel</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Interaction</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Adaptation</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Autonomy</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Collaboration</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Mobility</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Learning</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total:</td>
<td>21</td>
<td>35</td>
</tr>
<tr>
<td>Diff:</td>
<td>-43.2%</td>
<td>-38.5%</td>
</tr>
</tbody>
</table>

The use of aspects had a strong positive influence in the interface complexity of the MobiGrid architectural components, as shown in Table 8. 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 on the number of required interfaces of the MobilityManagement aspect. In the non-AO solution, the conventional component has four required interfaces to propagate four mobility events relative to the initialization, migration, destruction and instantiation of agents. These events are observed by the IReferenceObserver interface and propagated to the MobilityProtocol component. On the other hand, in the AO solution, the aspectual component MobilityProtocol cross-cuts the IReferenceObserver interface and directly observes the events when MobilityPlatform notify them. Hence, the required interfaces to propagate them are not necessary. Moreover, the inferiority of the non-AO version in the number of interfaces is granted to the fact it needs additional pairs of subscription interfaces involving the collaboration of the components MobilityManagement and MobilityPlatform, and the components MobilityProtocol and MobiGrid components.
Table 8. MobiGrid Architectures: Interface Complexity Metrics.

<table>
<thead>
<tr>
<th>Component</th>
<th>#Interfaces</th>
<th>#Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AO</td>
<td>Non-AO</td>
</tr>
<tr>
<td>MobilityPlatform</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>MobilityManagement</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>MobilityProtocol</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>MobiGrid</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>14</strong></td>
<td><strong>24</strong></td>
</tr>
<tr>
<td><strong>Diff:</strong></td>
<td><strong>-41.7%</strong></td>
<td><strong>-19.1%</strong></td>
</tr>
</tbody>
</table>

5 General Analysis and Related Work

This section provides a more general analysis with respect to modularity breakdowns observed in the results previously presented in Section 4. This section also brings discussions on how such results and conclusions are connected to observations made by related work.

5.1 Detecting Early Modularity Anomalies in MAS Architectures

The use of the architectural modularity metrics (Section 3.3) allowed us to observe some early design anomalies in the investigated MAS architectures. Our observations are classified into three main categories: (i) bidirectional architectural coupling, (ii) architectural interface bloat, and (iii) hindering of architectural variability and adaptability.

**Bidirectional Architectural Coupling.** First, after a careful joint analysis of the MobiGrid and AspectT architectures, we observed that both non-AO options – i.e. the mediator-based and the publisher-subscriber designs – imposed some undesirable bidirectional couplings. In the mediator architecture, all the “colleague” components need to inevitably have references to the “mediator” component and vice-versa. Similarly, in mediator-based architectures, all the “subscriber” components need to know the “publisher” components and vice-versa. Even though these architectural solutions overcome the problem of direct couplings between colleagues and between subscribers, the AO architectural solutions for both MobiGrid and AspectT systems have reduced even more 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 4.2). For example, the **Kernel** component has the fan-out zero in the AO version of the AspectT architecture, while it is 6 in the mediator version (Table 5). Also, **Adaptation**, **Autonomy** and **Learning** components have a fan-in zero.

**Architectural Interface Bloat.** The inter-component interaction constraints defined by the mediator-based and publisher-subscriber architectures did not scale respectively in the AspectT and MobiGrid systems, causing a complexity increase in the component interfaces. Such constraints have influenced the definition of extra operations and additional interfaces for the sake of realizing certain
crosscutting MAS concerns, such as mobility and learning issues. For example, the evidence of interface bloat can be observed in several parts of both non-AO architectures. As discussed in Section 4.3, the Kernel component in the AspectT design (Table 7) and the MobiGrid component (Table 8) had clearly much “wider boundaries” respectively due to their needs of mediating inter-component conversations and handling event subscriptions and notifications. In the particular case of MobiGrid, the event propagation is a issue that crosscuts the modularity of all the four architectural components.

_Hindering of Architectural Variability and Adaptability._ Variability and adaptability were main driving requirements in the architecture design of both multi-agent systems. For example, the design of the AspectT framework had the stringent requirement of making it optional the usage of the components Learning, Mobility, and Collaboration in order to allow for the flexible definition of heterogeneous agent architectures. In addition, the architecture design also required support for: (i) the adaptability of the Kernel elements, such as the provided services and agent plans, and (ii) the adaptability of agent roles and protocols, which should be dynamically plugged and unplugged. In the MobiGrid architectures, mobility issues should be modularized in order to promote easier variation and adaptation of the mobility platforms and protocols. However, the non-modularization of architectural crosscutting concerns in the mediator-based and publisher-subscriber architectures hindered the satisfaction of these variability and adaptability goals. This problem can be observed in the SoC measures (Section 4.1) where the results in Tables 3 and 4 show the tangling and scattering of several concerns, such as mobility, learning, and collaboration. As a result, the plugability and adaptation of elements realizing such concerns become cumbersome. Moreover, we have observed a certain rigidity in the composition rules defined by conventional patterns to support alternative compositions between agent components in order to smoothly produce heterogeneous agent architectures. This inflexibility is visible in both mediator and publisher-subscriber architectural patterns through the join analysis of all the metrics. The architectural composition rules of such patterns required the definition of higher coupling, more complex interfaces (as discussed above), and inferior separation of concerns.

The AO pattern also facilitated the variability of the mobility platform in the MobiGrid system. Although the non-AO architecture in Figure 3 allows a flexible integration between the MobiGrid and distinct mobility platforms, it does not provide a clean separation of mobility issues. This purpose is not reached because it involves the direct use of the mobility platforms, even though there are no references to a specific platform in the MobiGrid component. In fact, the MobiGrid component can indistinctly use mobility platforms such as Aglets and JADE, but it is still connected to the MobilityProtocol component through the IProtocolServices interface. This means that MobiGrid component makes explicit calls to MobilityProtocol services. In addition, the MobilityProtocol component externalizes the mobile agent lifecycle in a scheme based on the publisher-subscriber architectural pattern (see Figure 3). Then the high coupling
between MobiGrid and platform models still remains, and the mobility concern is not totally separated from the MobiGrid application concern.

5.2 Related Work

The body of knowledge on the interplay of MAS and Software Architecture has been mostly concentrated on suggesting and implementing new MAS architectures [24, 28, 2, 49, 10]. In particular, aspect-oriented software architectures [24, 25, 3, 43] are emerging as a promising way to develop modular MAS. However, less attempts has been done in studying how these architectures may be characterized and evaluated [14, 44]. To the best of our knowledge, no work has clearly indicated a framework on how to assess MAS architecture modularity and performed systematic case studies on the basis of such modularity evaluation framework.

Davidsson and colleagues [14] have used an Analytic Hierarchy Process to compare six MAS architectural styles for handling the task of dynamic and distributed resource allocation. Woods and Barbacci [50] have used the Architecture Tradeoff Analysis Method (ATAM) [12] for evaluating quality attributes of the agent-based system architectures. Similarly, Ivezic and colleagues [30] use an ATAM-based methodology to assess different architecture alternatives for a supply chain management MAS. However, none of this previous work has focused on supporting the assessment of modularity attributes in MAS architectures.

In a previous work, we have defined how to use an aspect-oriented design pattern to flexibly build heterogeneous agent architectures [24] (Section 2.2). We have also described guidelines to transit from the aspect-oriented definition of agent architectures to AspectJ-like [33] implementations. In [17, 19], we have defined the aspect-oriented agent framework, which we have qualitatively assessed in three different case studies. In [25] we have quantitatively compared OO and AO implementations of the Portalware MAS [19]. However, our evaluation was focused at the implementation stage and has used to simple systems. We have now exploited two multiple case studies in order to perceive how our proposed aspect-oriented architectural rules (Table 1) are reusable and scalable to different contexts, and on which circumstances they succeed or fail in order to promote improved MAS architecture stability.

In this work, we have not focused on assessing existing reference architectures [12] for MAS such as the one proposed in [49], which may be seen as a complementary abstraction to be exploited at architectural stage. Reference architectures can embody one or more architectural patterns. As a result, we have centered on the assessment and comparison of architectural patterns, which may be seen as one of the most basic building blocks in MAS architectural decompositions. Further work could explore the evaluation of existing MAS reference architectures.
6 Concluding Remarks and Future Work

Modularity occupies a pivotal position in the design of good MAS architectures. It is during architectural design that crucial modularity-related requirements in MAS such as adaptability, flexibility, reusability, maintainability, testability, etc., must be addressed. Yet the task of considering the multi-dimensional facets of modularity remains a deep challenge to MAS architects. As adaptability, reusability, and maintainability are typically driving requirements in MAS designs, the architectural pattern that provides the most appropriate balance between the modularity attributes should be selected. 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 concerns. MAS architects need to deal with issues such as making an agent interact appropriately, handling the agents’ adaptive behavior, structuring the agents’ autonomous behavior, designing the agent roles and protocols for inter-agent collaboration purposes, and incorporating learning mechanisms into the agent’s structure in a modular manner.

This paper is a first attempt to systematically evaluate the added value of aspect-oriented software architectures for designing MAS. Without a clear evidence of its strengths and drawbacks, MAS architects are not guided on when to use them. As previously discussed (Section 5.2), there is no systematic assessment on what extent such an emerging aspect-oriented MAS architectures scale in different circumstances and applications. We should emphasize here that the conclusions drawn from this first study are limited to the particular two case studies and selected optional architectures. However, it provides a first understanding towards how to determine for the use of aspects early at the design of adaptable and reusable MAS architectures in the presence of crosscutting concerns. A number of additional studies in the future should be performed to improve our body of knowledge regarding the interplay of aspects and MAS architectures. A next step in our assessments is to evaluate aspect-oriented MAS architectures in the light of different architectural attributes other than modularity issues, such as performance and availability. It would allow us to have a more broad understanding of when and when not using such an architectural style for MAS.

Acknowledgements. Claudio is supported by CAPES-Brazil under Grant No. 3651/05-3. This work has been also partially supported by FAPERJ under grant No. E-26/151.493/2005 for Uira. Alessandro is supported by European Commission as part of the Grant IST-2-004349: AOSD-Europe, 2004-2008. Christina has been partially supported by CNPq-Brazil, under Grant No. 479395/2004-7 and by Fapesb (The State of Bahia Research Foundation).

References

2. A. Amandi and A. Price. Building Object-Agents from a Software Meta-
Architecture. In SBIA ’98: Proceedings of the 14th Brazilian Symposium on Arti-
Protocols Using Ontologies. Journal of Integrated Computer-Aided Engineering,
4. R. M. Barbosa and A. Goldman. MobiGrid. In A. Karmouch, L. Korba, and
E. R. M. Madeira, editors, MATA, volume 3284 of Lecture Notes in Computer
5. T. Batista, C. Chavez, A. Garcia, C. Sant’Anna, U. Kulesza, A. Rashid, and
F. Filho. Reflections on Architectural Connection: Seven Issues on Aspects and
ADLs. In International Workshop Aspect-Oriented Requirements Engineering and
6. F. Bellifemine, G. Rimassa, and A. Poggi. JADE A FIPA-compliant Agent Frame-
work. In 4th International Conference on the Practical Applications of Agents and
7. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-
8. N. Cacho, T. Batista, A. Garcia, C. Sant’Anna, and G. Blair. Improving Adapt-
ability of Reflective Middleware with Aspect-Oriented Programming. In Fifth In-
ternational Conference on Generative Programming and Component Engineering
(GPCE’06) - (submitted), October 2006.
In AOSD ’06: Proceedings of the 5th international conference on Aspect-oriented
10. D. Camacho. Coordination of Planning Agents to Solve Problems in the Web. AI
11. R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tek-
inderogan, and A. J. Siobhán Clarke and. Survey of Aspect-Oriented Analysis and
Design Approaches. Technical Report AOSD-Europe-ULANC-9, AOSD-Europe,
May 2005.
12. P. Clements, R. Kazman, and M. Klein. Evaluating Software Architectures: Meth-
13. K. Damasceno, N. Cacho, A. Garcia, A. Romanovsky, and C. Lucena. Context-
Aware Exception Handling in Mobile Agent Systems: The MoCA Case. In 5th
International Workshop on Software Engineering for Large-Scale Multi-Agent Sys-
of Multi-agent System Architectural Styles. In A. Garcia, R. Choren, C. J. P.
de Lucena, P. Giorgini, T. Holvoet, and A. B. Romanovsky, editors, SELMAS,
Exceptions and Aspects: The Devil is in the Details. In Proc. of FSE-14, 14th Int’l
16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns - Elements of
Reusable Object-Oriented Software. Addison-Wesley, 1996.
Pontifical Catholic University of Rio de Janeiro, Brazil, April 2004.


