Aspects in Agent-Oriented Software Engineering: Lessons Learned

Alessandro Garcia[1], Uirá Kulesza[2], Cláudio Sant'Anna[2], Christina Chavez[3], Carlos J. P. de Lucena[2]

[1]Lancaster University, Computing Department, InfoLab 21
Lancaster - United Kingdom
a.garcia@lancaster.ac.uk

[2]PUC-Rio, Computer Science Department, LES
Rio de Janeiro - Brazil
{uira,claudio,lucena}@les.inf.puc-rio.br

[3]Federal University of Bahia (UFBA), Computer Science Department
Salvador - Brazil
flach@im.ufba.br

Abstract. Several concerns in the development of multi-agent systems (MASs) cannot be represented in a modular fashion. In general, they inherently affect several system modules and cannot be explicitly captured based on existing software engineering abstractions. These crosscutting concerns encompass internal agent properties and systemic properties, such as learning, code mobility, error handling, and context-awareness. In this context, it is important to systematically verify whether emerging development paradigms support improved modularization of the crosscutting concerns relative to MASs. This paper reports some lessons learned based on our experience in using aspect-oriented techniques and methods to address these problems. In the light of these lessons, related work and a set of future research directions are also discussed.

1 Introduction

Software engineering of large multi-agent systems (MASs) involves a number of concerns, including autonomy, roles, learning, mobility, error handling, fault tolerance, and context-awareness. The modeling, design, and implementation of many of these concerns are challenging because they are inherently crosscutting as the system complexity increases. In other words, these concerns crosscut several agent actions and plans, which implement the agents' basic functionality and other agent concerns. Several system quality attributes, such as reusability and maintainability, depend largely on the ability of software engineering techniques and methods to support the explicit separation of MAS concerns throughout the design and implementation stages.

Existing modeling languages [8, 20] and design and implementation approaches [6, 9, 11, 20] are not able to provide explicit support for the separation of crosscutting MAS-related concerns. In this context, it is important to systematically verify whether emerging development paradigms support improved modularization of the crosscutting concerns relative to MASs. Aspect-oriented software development (AOSD) [12] is a promising paradigm to promote improved separation of concerns, leading to the production of software systems that are easier to maintain and reuse. AOSD is centered

on the aspect notion as an abstraction aimed to modularize crosscutting concerns throughout the software lifecycle. Hence, aspect-oriented approaches are candidates to address the crosscutting property of some concerns in multi-agent systems. However, up to now AOSD research has focused on trivial or well-known crosscutting concerns, such as logging, tracing, distribution, and persistence. There are very few reported experiences involving aspects in the MAS domain. For example, Kendall et al focus on the use of aspects to modularization of roles [10].

There is a pressing need for understanding the interplay between agent-oriented software engineering (AOSE) and AOSD. This paper reports some lessons learned based on our experience in applying both aspect-oriented techniques and methods to the construction of MASs. We have developed and applied aspect-oriented approaches to specify [22], architect [21], design [6], and implement [23] multi-agent systems. We have also conducted some qualitative [7] and quantitative [24] empirical studies. Our lessons learned are related to four different inter-related dimensions of software engineering, which are captured in the following research questions:

(i) What are the main motivations to use AOSD techniques for MAS development?
(ii) What are the MAS-related concerns which were well modularized with aspects according to our experimental settings?
(iii) What are the limitations of existing aspect-oriented techniques, methods and tools to address crosscutting concerns in MASs?
(iv) What are some future directions that naturally emerged from the practical exploration of AOSD in the context of MASs?

The lessons learned presented here provide a clear understanding of important strengths and weaknesses of the investigated aspect-oriented approaches as well as their compatibility and divergences. The results are important sources towards a potential integration of AOSD and AOSE. They are also useful for engineers of realistic MASs who need to model, design and implement their systems in the presence of crosscutting concerns. The conclusions may also be of interest to agent-oriented methodologists since they may decide to incorporate solutions for problems detected in our experiences directly as part of their methodologies.

The remainder of this paper is organized as follows. Section 2 presents some typical examples of crosscutting concerns in MASs. Section 3 introduces relevant AOSD terminology and overviews our aspect-oriented approach to support the modularization of MASs. Section 4 presents the lessons learned. Section 5 discusses related work. Section 6 includes some concluding remarks and directions for future work.

2 Crosscutting Concerns in MASs

Several authors have identified that some agent properties are often crosscutting, such as mobility [25], interaction [23, 24], learning [26], autonomy [27, 28], and collaboration [10, 25]. Some empirical studies confirm their findings [7, 15, 24]. This section presents some examples of crosscutting concerns in MAS development. A concern is some part of a MAS that we want to treat as a single conceptual unit. Concerns are modularized throughout software development using different abstractions provided by techniques, methods, and tools.

Fig. 1 shows a partial representation of a multi-agent system [23], which was modeled with an agent-oriented extension to UML (based on stereotypes), and implemented using the Java programming language. The JADE platform was also used to support inter-agent collaborations and agent mobility. Machine learning techniques were designed and implemented to address the learning-related requirements of this application. Role modeling was used to structure the collaborative capabilities of the agents. Each set of classes, surrounded by a gray rectangle, has the main purpose of modularizing a specific agent concern, namely interaction, environment, basic concerns, learning, and collaboration. This MAS includes other MAS concerns, such as mobility and error handling, wich are not represented in the figure for simplicity purposes.

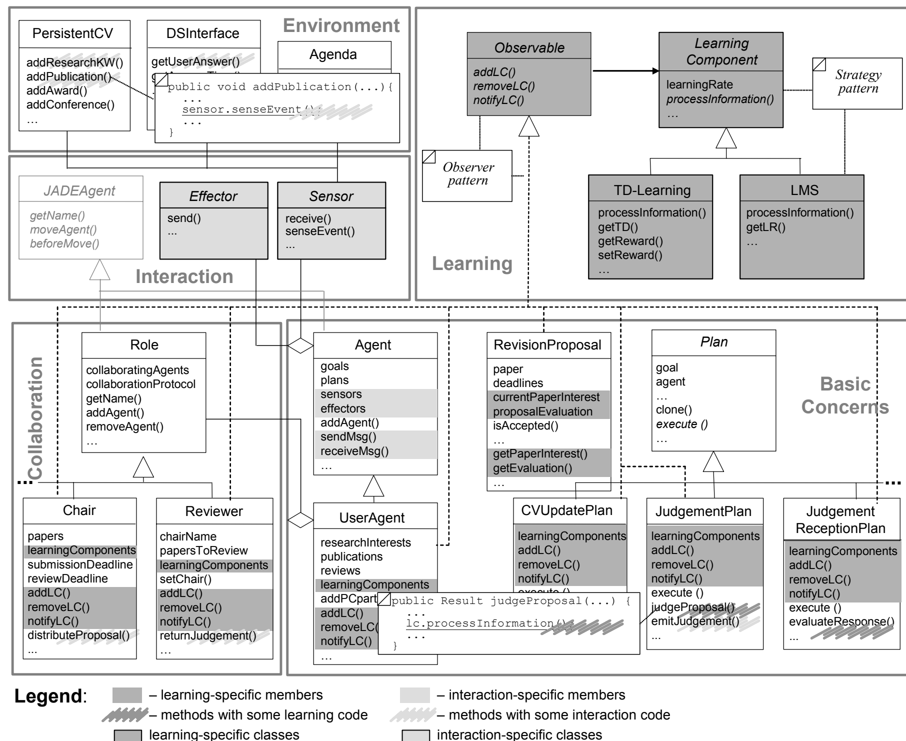

Fig 1 Crosscutting Agent Concerns

Note that, for example, the learning concern crosscuts several agent actions and plans implementing other agent concerns; it has a huge impact on the basic agent structure and the collaboration design. Although part of the learning concern is localized in the classes of the Strategy and Observer patterns, learning-specific code replicates and spreads across several class hierarchies of a software agent. Several participants have to implement the observation mechanism and the gathering information and, as a consequence, have learning code in them. Some classes (e.g. RevisionProposal class) have learning-specific knowledge. Adding or removing the learning code from classes requires invasive changes in those classes. Note that even if we try to refactor the design solution presented in Fig. 1, we cannot find a more modular solution. This problem

happens because learning is a crosscutting concern independently of the system decomposition used [30]. Fig. 1 also illustrates similar problems for the interaction concern, which is usually crosscutting.

3 Modularizing Multi-Agent Systems with Aspects

This section presents relevant terminology of AOSD (Section 3.1), overviews our aspect-oriented approach to deal with crosscutting concerns in MAS development (Section 3.2), and illustrates the modularization of a crosscutting concern based on our approach (Section 3.3).

3.1 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) [12] has been proposed as a technique for improving separation of concerns in software construction and support improved reusability and maintainability. Aspect-oriented (AO) techniques are not restricted to the object paradigm [12], but it has been their main focus up to now. The central idea is that while pure abstractions of exiting paradigm (such as object-orientation, component-orientation, and agent-orientation) are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. Thus, the goal of the AO techniques is to deal with crosscutting concerns, by providing abstractions that make it possible to separate and compose them to produce the overall system. Crosscutting concerns are defined as system concerns that crosscut conventional system modules (such as objects, components, and agents) in the system development.

Aspects are modular units of crosscutting concerns that are associated with a set of classes (for example). Obliviousness and quantification are often considered as fundamental properties of aspectual modules. Central to the process of composing aspects and classes is the concept of join points, the elements that specify how classes and aspects are related. Join points are well-defined points in the structure and dynamic execution of a system. Examples of join points are method calls, method executions, and field sets and reads. An aspect defines sets of join points and advice. Advice is a special method-like construct attached to join points. An aspect may also define attributes and methods to be introduced into the classes. Weaver is the mechanism responsible for composing the classes and aspects. AspectJ [13] is a practical aspect-oriented extension to the Java programming language. AspectJ supports the definition of aspects, advices, join points, and pointcuts. Pointcuts are collections of join points and are used in advice definitions.

3.2 Aspect-Oriented Modeling, Design and Implementation of MASs

The basic idea of our approach is the use of aspect-oriented abstractions to enable improved separation of crosscutting concerns in the software engineering artifacts associated with MASs. Aspects are used as unifying abstractions to capture the agent concerns that are hard to modularize with both existing agent-oriented at high-level specifications and object-oriented abstractions at detailed design and implementation. These aspects are supported from high-level specifications and architecture design to the detailed design and implementation. The goal is to obtain untangled software artifacts

and promote an improved separation of concerns. The proposed approach is independent of MAS implementation frameworks, such as JADE [1] and ZEUS [14].

Our approach supports MAS developers with four main elements. The first element of our approach is a domain-specific language (DSL) [22], called Agent-DSL, that supports the high-level modeling of the MAS at hand. Agent-DSL supports the modeling of fundamental abstractions in AOSE, such as agents, plans, actions, and goals, as well as the modeling of crosscutting agent-related features as separate modules, i.e. aspects. Aspects are used to model concerns like mobility, learning, roles, and adaptation.

The second element of our approach is an aspect-oriented software architecture [21] for structuring the basic architecture of a software agent. Since the agents have been specified using our DSL, the internal architecture of each agent type in the system needs to be defined. As the system specification is refined, new crosscutting concerns manifest and must be modularized at the architectural stage. Our architecture provides a set of constraints to support the modularization of crosscutting concerns as architectural aspects. This aspectual agent architecture is flexible to support different compositions of agent concerns for heterogeneous agent types [21]. In addition, our approach provides a set of guidelines [6, 31] to refine the specification of aspectual agent architectures in terms of detailed design.

The third element is a language of design patterns that provides solutions for the detailed design of crosscutting MAS-related concerns, such as mobility [29], learning [30], roles [23], interaction [23], autonomy [23], and adaptation [23]. The patterns can be directly mapped into implementation elements. The proposed design patterns have been implemented in AspectJ. In this context, the fourth element of our approach is an implementation framework [23, 22], called AspectT, which materializes those design patterns in AspectJ and provides support for the implementation of the crosscutting MAS concerns with a set reusable classes and aspects. Finally, we have a set of prototype tools [22] to support the DSL-based modeling of the multi-agent system and the partial code generation of this system based on those provided models.

3.3 An Example

This section illustrates how the use of our aspect-oriented approach supports the modularization of the learning concern (Section 2). Due to space limitations, in this work, we focus on the *design* of the learning concern. A more detailed description for the other MAS concerns and other development stages, the reader should refer to the publications described in Section 3.2. In fact, the full description of our approach is outside the scope of this paper. Learning aspects encapsulate the entire implementation of the learning concern, including the learning-specific knowledge and the information gathering (Fig. 2).

The design notation in Fig. 2 is based on an aspect-oriented modeling language [2]. This language extends UML with notations for representing aspects. The notations provide a detailed description of the aspect elements. In this modeling language, an aspect is represented by a diamond; it 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 [2]. Each crosscutting interface is presented using the rectangle symbol with compartments.

A crosscutting interface is composed of inter-type declarations, pointcuts and advices. The notation uses a dashed arrow to represent the crosscutting relationship, which relates one aspect to classes and/or aspects.

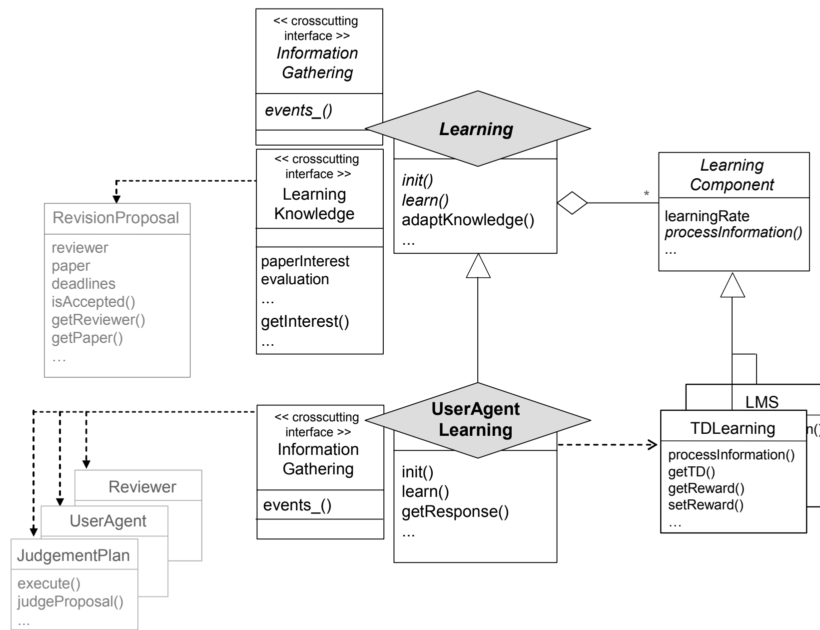

Fig 2 The Detailed Design of the Learning Component.

Fig. 2 shows that the Learning aspect separates the learning protocol from the kernel and other aspects, such as UserAgent class, Plan classes, and role aspects. The Learning aspects connect the execution points (events) on different agent classes with the corresponding learning components, making it transparent to the agent's basic functionality the particularities of the learning algorithms in use. These aspects are able to crosscut some agent execution points in order to change their normal execution and invoke the learning components. The execution points include the change of a knowledge element, execution of actions on plans, roles, and agent types, or still some threw exception.

Auxiliary classes are used to implement different learning techniques. This learning experience is indirect because the agent will build its knowledge through the results of the inter-agent negotiations. Machine learning is used to address the knowledge acquisition. Distinct learning techniques are used in the EC system: Temporal Difference Learning (TD-Learning) [23] and Least Mean Squares (LMS) [23]. TD-Learning is used by the reviewer role in order to learn the user preferences in the subjects he/she likes to review. LMS is used by the chair to learn about the reviewer preferences. Note that the scattering and tangling relative to the learning concern presented in Fig. 1 is overcome in the aspect-oriented solution (Fig. 2).

4 Lessons Learned

We discuss below important benefits and pitfalls in using AO techniques for the development of MASs. Our lessons learned are related to four different inter-related dimensions of software engineering, which are presented in the following subsections.

4.1 Motivation for AOSD in MAS Development

The main motivation for using AOSD techniques is the increasing complexity of today's agent-based applications. The advent of novel and innovative networking technologies makes it necessary for software systems to incorporate and deal with an ever greater variety of agent-specific concerns such as mobility, adaptation, and learning. Underlying all of these special purpose concerns is the basic concern responsible for the basic functionality of the system. According to our experience [4, 23, 24], the agent properties are typically overlapping and crosscut the agent's basic functionality. The basic functionalities of agents already are quite complicated, and so agent properties should be designed separately from the agents' basic behaviors [31].

With MASs growing in size and complexity, the separation of their concerns throughout the different development phases is crucial to MAS engineers. Separation of crosscutting concerns is an important principle in software engineering to achieve improved reusability and maintainability of complex systems. The lack of modularization of crosscutting concerns raises a number of problems:

- Designing intertwined behaviors is hard and complex since all concerns have to be dealt with at the same time and at the same level. Agent-oriented modeling languages and OO programming languages provide no adequate abstractions for separation of crosscutting concerns in the modeling and implementation levels.
- Intertwined behaviors are hard to understand because of a lack of abstraction.
- Intertwined behaviors are both hard to maintain and reuse because the concerns are strongly amalgamated.
- Intertwined behaviors give rise to inheritance anomalies due to the strong connection of the different agent concerns. It becomes impossible to change a method's implementation or an intertwined special concern in a subclass without changing both.

4.2 Separable and Inseparable MAS Concerns

Separable Concerns. According to our experience, there was a number of crosscutting MAS-specific concerns which aspect-oriented abstractions succeeded to cope with their modularization. This was often the case for mobility, learning, roles, and autonomy concerns. For these agent properties, the design and implementation have shown expressive improvements in terms of separation of concerns. This observation provides evidence of the effectiveness of AO abstractions for segregating crosscutting structures.

The use of aspects in these cases was also useful to reduce the coupling between the design modules for these concerns and increase their cohesion, since the aspect-oriented mechisms enabled the modularization of all the behaviours relative to these agent concerns. We have captured a common characteristic of these aspects: they exihibit a high connection between the internal elements of these aspects, i.e. a high interaction

between internal aspect attributes, methods, inter-type declarations and advices, which is fundamental to improve their cohesion and minimize the system coupling.

Aspects Emerging in the Software Lifecycle. Many agent properties in a MAS will likely not be designed from scratch as aspects. Rather, many crosscutting concerns will emerge as a MAS evolves. An assessment framework based on a metrics suite [17], for instance, could help the detection of crosscutting concerns in the MAS at hand. Capturing such agent concerns as aspects sometimes also requires restructuring of the classes and methods implementing the agents' basic functionality and other agent aspects to expose suitable join points (Section 4.3).

Inseparable Concerns. There were also some crosscutting concerns, which aspect-oriented solutions failed to improve their modularization. For example, the Interaction aspects do not modularize the message assembling from different plans or roles; the message needs to be prepared within a method on plan classes or on role aspects because its assembling is very coupled to the role or plan context of the respective agent. One solution would be to separate the message assembling with aspects, but it would result in higher complexity.

The design of the adaptation concern, for example, also sounds to be natural in the OO fashion, and it does not seem reasonable or even possible to isolate the adaptation behavior into aspects. The AO design of the adaptation aspects somewhat improved the concern locality, but the differences in terms of coupling and cohesion are not significant. In fact, an additional interesting observation in our studies is that sometimes the crosscutting MAS concerns can be expressed separately as aspects, but it remains non-trivial to specify how these separate aspects should be recombined into a simple manner. A lot of effort is required to compose the participant classes and the aspects that modularize the agent concern. Hence, there are some cases where the separation of the pattern-related concerns leads to more complex solutions.

Inter-Aspect Relationships. Many aspects are orthogonal and interact with each other. For example, code mobility affects not only the agent kernel, but also other important agent concerns such as roles, interaction, and learning. Since the mobility concern is related to these concerns, the presence of sophisticated composition mechanisms is important to specify the relationships between the mobility aspects and these other agent aspects.

Complex Structure for Simple Agents. We have often found that the choice between using aspects or not depends highly on the complexity of the crosscutting concern in the specific agent-oriented application at hand. For example, we have decided to use aspects for modularizing the autonomy property of software agents in a reactive MAS. However, some simple reactive agents do not require thread control, react only to few events, make very simple decisions, and do not have proactive behavior. In this case, the autonomy code tends to be localized in fewer methods. The use of aspects in this specific situation can increase rather than decrease the agent design's complexity.

Overlapping Concerns. There were some concerns that have shown themselves as overlapping. For example, adaptation and learning are a classical example of overlapping concerns. The implementation of the learning aspects includes the same behaviors already implemented by the adaptation aspects. In order to avoid code duplication, we have exposed this common behavior as part of the interface of the adaptation aspects so that the learning aspects can access them.

Aspects as 'Glue' between Agents' Basic Concerns and MAS Frameworks. Several of our agent aspects at the detailed design level achieved a common structure: the aspect behavior forms the glue between the OO structure implementing the agents' basic functionality and the specific MAS frameworks used to support some concerns. For instance, the Learning aspects [30] work as a glue between the hierarchy of agent types and the hierarchy of modules implementing the learning strategies. This design structure is beneficial because it allows to express the agent's basic functionality in its own object structure and to use an aspect to inject that agent property into the basic functionality in a way that is transparent. In our case studies, the same design solution was also applied to glue the basic agent structure and the mobility frameworks and platforms.

Incremental Process vs. Iterative Process. During our case studies [7, 23, 24], we have tried to "incrementally" deal with agent concerns at the specification (using Agent-DSL mentioned in Section 3.2), architectural, design and implementation stages. We have found that, as the MASs increases in complexity, the boundary between increments is not as transparent as expected. For example, the design and implementation of the mobility aspects required the creation of new pointcuts in the interaction aspects previously defined. In this way, we mostly had to follow an iterative process rather than an incremental approach in order to specify and implement the aspect-oriented agent architectures.

Mastering Aspects Complexity. In the design and implementation of the agent aspects, we have observed that it is easier to build an aspect-oriented system when the interface between aspects and classes is narrow and unidirectional. *Unidirectional* means that the aspect code refers to the classes but not vice versa, although many AO solutions do not follow this constraint. In fact, central to the quality achieved in our MAS is the notion of structuring crosscutting concerns separately from the "primary" agent concerns, using aspects that cannot be referenced back by the objects. Narrow means that the aspect code has a well-defined effect on particular points in the code.

4.3 Limitations of Existing Aspect-Oriented Techniques to MAS Development

Inter-Aspect Conflicts. As mentioned previously, we have used AspectJ to implement the agent aspects. The modularization of some agent concerns with AspectJ caused aspectual conflicts. For example, roles were implemented as AspectJ aspects in our case studies. Each role aspect introduces the role behavior to the respective classes that represent the agents playing that role. This behavior introduction was implemented as AspectJ inter-type declarations. However, some of the roles encompassed similar structural and behavioral elements due to their very nature. Hence, this property of roles imposed conflicts in their AspectJ implementations. As an agent can play more than a role, those attributes and methods had to be renamed and changed so that two or more conflicting role aspects could be added to the same agent class.

Lack of Obliviousness. As mentioned previously, capturing agent concerns as aspects sometimes required restructuring of the classes and methods implementing the agents' basic functionality and other aspects to expose suitable join points. For instance, we have extracted code from existing methods of a plan class into a new method to expose a method-level join point so that a *role aspect* could intercept it. In these cases, the aspect obliviousness is not complete, although many AOSD researchers (e.g. [32]) argue that obliviousness is a fundamental property of AOSD. In this context, intimacy has been

defined as the additional effort required to prepare the classes and methods for the incorporation of aspects into the system [32].

Repetitive and Time-Consuming Definitions. We have modularized the interaction concern (Section 3.3) as aspects using both Agent-DSL and AspectJ. As part of the aspectization process of the interaction concern at the implementation level, all the message senders of the system must be specified in the pointcut inside the Interaction aspects. This might indeed be repetitive and tedious, suggesting that AspectJ, the language used in our studies, should have more powerful metaprogramming constructs. However, this is not an unsolvable problem because code-generation tools can assist MAS engineers in this development step (Section 4.4). In addition, we can establish a naming convention and use wildcards supported by most aspect-oriented languages. The initial implementation of our case studies used naming conventions.

Naming Conflicts. The use of AspectJ and existing MAS frameworks/ platforms to implement the aspect-oriented agent architectures have led to some architectural mismatches. For example, we have used JADE to support code mobility in our case studies. The JADE architecture imposes on the application developers the extension of the jade.Agent class to make the application agents (or specific roles) mobile. This abstract class provides a number of mobility services, such as a method getName() which is responsible for generating an unique name for the mobile agent instances in a distributed context. However, we have previously defined a Agent class in the agent architectures that implement the agents' basic services, such as a method getName() with a different purpose. As we have used AspectJ aspects to implement the mobility concern, we used inter-type declarations to specify this extension in the aspects and inject the implementation of the jade.Agent class in the Agent class in our application. This architectural mismatch caused by the AspectJ mechanism required the renaming of this method and changes in the respective callers.

Implementation Limitations. In addition, some AspectJ restrictions complicated the materialization of some architectural and design solutions. For example, each agent instance must often have its own mobility aspect. As a consequence, mobility aspects must be instantiated per Agent instance (or Role instance). The current version of AspectJ supports the specification of per-object aspects. We could describe the instantiation of the Mobility aspect using the `perthis` mechanism, such as:

```
public abstract aspect Mobility perthis(Agent) {…}
```

However, the use of `perthis` restricts the scope of the aspect. When one AspectJ aspect is declared to be singleton or static, its scope is the whole system and the aspect can crosscut all system classes. Per-object aspects can only crosscut the object with which it is associated. Since the mobility protocol crosscuts several classes, not only the Agent class or the Role class, the perthis clause cannot be used in this context. As a result, we have to declare mobility aspects as singletons and introduce the methods and attributes to the Agent and Role classes. The use of inter-type declarations complicates the design of the Mobility aspect since it requires the agent or role instance to be exposed as a parameter in each advice of the Mobility aspect.

4.4 Research Directions

Need for Improved Traceability. According to our experience, there is a need for handling aspects in a uniform way throughout the different development stages. In our

approach, agent aspects were represented as first-class elements in the system specification (using the Agent-DSL), in the detailed design and in the implementation level (using AspectJ). However, we missed some support for describing the agent aspects in intermediary modeling languages, such as agent-oriented design languages in order to support a better traceability between the software artifacts.

Code Generation. The definition of some agent aspects involves some time-consuming tasks, such as extensive description of pointcuts. Tools should be developed to maximize the automatic generation of the pointcuts and overcome this time-consuming task. We are improving a generative approach [22] that supports the code generation of the agent aspects. The idea is to support our methodology (Section 3.2) with an additional number of tools and wizards that automate the code generation in AspectJ.

Aspects in Agent-Oriented Modeling. According to the experience of this research, there are some crosscutting concerns even at the agent-oriented modeling level, such as coordination, exception handling, and context awareness. Our work has been more concerned with an architectural and design approach. There is a need to extend existing AOSE meta-models (e.g. TAO [33]) and agent-oriented modeling languages (e.g. Gaia [19]) with aspect-oriented abstractions to support the representation of crosscutting concerns in agent-oriented models.

Integration with another development methods. With the growing and dissemination of the use of agent technology in the development of software systems, it is also important to adapt current methods of agent-oriented software engineering to be integrated with existing development methods. In this sense, we have particular interest in how we can integrate our approach with other Web modeling and design methods, such as, OOHDM [34] and WebML [35], in order to introduce software agents in web-based information systems.

5 Related Work

Dealing with several crosscutting agent concerns, such as mobility and learning, has been recognized as a serious problem that has not received enough attention [15, 25]. However, research in agent-oriented software engineering has concentrated on high-level methodologies and modeling languages [20], without giving enough attention to the role of aspects in the context of AOSE. In addition, implementation frameworks [1, 14] provide object-oriented APIs for MAS development, without providing guidelines for the modularization of crosscutting agent concerns.

Some researchers have recognized these problems for some single agent concerns, such as roles [26] and mobility [25], and have proposed techniques for dealing with these concerns. However, such techniques are for separating only these individual concerns. In addition, most of them are focused only on the implementation stage and do not provide explicit support for the separation of crosscutting concerns at early development stages.

7 Conclusions and Ongoing Work

Aspect-oriented software development is gaining wide attention both in research environments and in industry [110, 149, 156, 230]. AOSD is a promising paradigm to promote improved separation of concerns, leading to the production of software systems

that are easier to maintain and reuse. The separation of MAS concerns is essential to software engineers since they may decide to extend and modify such concerns as the system evolves. Hence it is important to systematically verify whether emerging development paradigms support improved modularization of the crosscutting concerns relative to MASs. More generally speaking, there is a need for understanding the relationships between objects and agents.

This paper has presented some lessons learned with the use of aspect-oriented techniques and methods to develop some multi-agent systems. This paper complements our previous work on empirical studies [7, 23, 24] by focusing on more generic software engineering questions. Our aspect-oriented approach has been evaluated using representative systems from different domains. An initial quantitative study [24] has provided evidences of the benefits of the aspect-oriented approach. Although experimental studies are time-consuming, it is necessary to replicate this study with a variety of agent applications and with different heterogeneity facets. These replicated studies would build an improved body of knowledge about the interplay among aspects and MAS concerns. For example, it would be desirable to conduct quantitative studies with the application of AI techniques in a large-scale fashion in order to understand how the aspect-oriented approach scales in this context.

References

[1] Bellifemine, F., Poggi, A., Rimassi, G. "JADE: A FIPA-Compliant agent framework". Proc. Practical Applications of Intelligent Agents and Multi-Agents, April 1999, pp. 97-108.

[2] Chavez, C., Lucena, C. "Design Support for Aspect-oriented Software Development". Doctoral Symposium at OOPSLA'2001, Tampa Bay, USA, October 2001, pp. 14-18.

[3] Finin, T. et al. "KQML as an Agent Communication Language", Proc. of the 3rd Intl. Conference on Information and Knowledge Management, ACM Press, 1994, pp. 456-463.

[4] FIPA Specifications. "FIPA ACL Message Structure Specification". http://www.fipa.org/specs/fipa00061/

[5] Gamma, E. et al. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, Reading, 1995.

[6] Garcia, A., Lucena, C., Cowan, D. "Agents in Object-Oriented Software Engineering". Software: Practice and Experience, Elsevier, April 2004.

[7] Garcia,A., Silva,V., Chavez,C., Lucena,C. "Engineering Multi-Agent Systems with Aspects and Patterns". Journal of the Brazilian Computer Society, July 2002, v. 8, no. 1, pp. 57-72.

[8] Iglesias, C. et al. "A Survey of Agent-Oriented Methodologies", Proceedings of the ATAL-98, Paris, France, July 1998, pp. 317-330.

[9] Jennings, N., Wooldridge, M. "Agent-Oriented Software Engineering". In: J. Bradshaw (Ed). "Handbook of Agent Technology". AAAI/MIT Press, 2000.

[10] Kendall, E. "Role Model Designs and Implementations with Aspect-oriented Programming". OOPSLA 1999, pp. 353-369.

[11] Kendall, E., Krishna, P., Pathak, C., Suresh, C. "A Framework for Agent Systems", In: Fayad, M. et al (Eds), "Implementing Applications Frameworks: Object Oriented Frameworks at Work". John Wiley & Sons, 1999.

[12] Kiczales, G. et al. "Aspect-Oriented Programming". European Conference on Object-Oriented Programming (ECOOP), LNCS (1241), Springer-Verlag, Finland., June 1997.

[13] Kiczales, G. et al. "Getting Started with AspectJ". Communication of the ACM, vol. 44, no. 10, October 2001, pp. 59-65

[14] Nwana, H., Ndumu, D., Lee, L. "ZEUS: An advanced Toolkit for Engineering Distributed Multi-Agent Systems", Proceedings of PAAM'98, 1998, pp. 377-391.

[15] Pace, A., Trilnik, F., Campo, M. "Assisting the Development of Aspect-based MAS using the SmartWeaver Approach". In: Garcia, A. et al (Eds). "Software Engineering for Large-Scale Multi-Agent Systems". Springer-Verlag, LNCS 2603, April 2003.
[16] Rashid, A. "A Hybrid Approach to Separation of Concerns: The Story of SADES". Proceedings of the Reflection 2001, LNCS 2192, pp. 231-249.
[17] Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., Staa, A. "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework". Proc. of the XVII Brazilian Symposium on Software Engineering, Manaus, Brazil, October 2003, pp. 19-34.
[18] Sycara, K., Paolucci, M., Velsen, M., Giampapa J. "The RETSINA MAS Infrastructure." Journal of Autonomous Agents and Multi-Agent Systems, v. 7, n. 1/2, July/September 2003.
[19] Wooldridge, M., Jennings, N., Kinny, D. "The Gaia Methodology for Agent-Oriented Analysis and Design". Journal of Autonomous Agents and MAS, 3:3, 2000, pp. 285-312.
[20] Bergenti, F., Gleizes, M.-P., Zambonelli, F. (Eds.). Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. Springer-Verlag, 2004.
[21] Garcia, A., Lucena, C. "Taming Heterogeneous Agent Architectures with Aspects". Communications of the ACM, March 2005. (submitted)
[22] Kulesza, U., Garcia, A., Lucena, C. A Generative Approach for Multi-Agent System Development. "Software Engineering for Multi-Agent Systems III", Springer, LNCS 3390, December 2004, pp. 52-69.
[23] Garcia, A. From Objects to Agents: An Aspect-Oriented Approach. PhD Thesis, Computer Science Department, PUC-Rio, Brazil, April 2004.
[24] Garcia, A. 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, February 2004.
[25] N. Ubayashi, T. Tamai. Separation of Concerns in Mobile Agent Applications. Proc. of the 3rd Conference Reflection 2001, LNCS 2192, Kyoto, September 2001, pp. 89-109.
[26] M. D'Hondt, K. Gybels, V. Jonckers. Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis. Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004), Nicosia, Cyprus, March 2004.
[27] Z. Guessoum, J. Briot. From Active Objects to Autonomous Agents. IEEE Concurrency, Special Series on Actors and Agents, Vol. 7, N. 3, 1999, pp. 68-76.
[28] A. Amandi, A. Price. Building Object-Agents from a Software Meta-Architecture. In: Advances in Artificial Intelligence, LNAI, vol. 1515, Springer-Verlag, 1998.
[29] A. Garcia et al. The Mobility Aspect Pattern. Proc. of the 4th Latin-American Conference on Pattern Languages of Programming, SugarLoafPLoP'04. August, 2004, Fortaleza, Brazil.
[30] A. Garcia et al. The Learning Aspect Pattern. Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Monticello, USA.
[31] A. Garcia, U. Kulesza, C. Lucena. Aspectizing Multi-Agent Systems: From Architecture to Implementation. "Software Engineering for Multi-Agent Systems III". Springer-Verlag, LNCS 3390, December 2004, pp. 121-143.
[32] Filman, R. What Is Aspect-Oriented Programming, Revisited. Proceedings of the Workshop on Advanced Separation of Concerns at ECOOP'01, June 2001.
[33] V. Silva et al. "Taming Agents and Objects in Software Engineering". In: "Software Engineering for Large-Scale Multi-Agent Systems", Springer, LNCS 2603, March 2003.
[34] Ceri, S., Fraternali, P: Web Modeling Language (WebML): a modeling language for designing Web sites. Proceedings of the 9th. International World Wide Web Conference, Elsevier 2000, pp 137-157.
[35] Schwabe, D., Rossi, G.: "An object-oriented approach to web-based application design". Theory and Practice of Object Systems (TAPOS), Special Issue on the Internet, vol. 4, pp.207-225, October, 1998.