On the Modularity Assessment of Aspect-Oriented Multi-Agent Systems Product Lines: a Quantitative Study

Camila Nunes[1], Uirá Kulesza[2], Cláudio Sant'Anna[1], Ingrid Nunes[1], Carlos Lucena[1]

[1]PUC-Rio – Computer Science Department, LES, Rio de Janeiro, Brazil

[2]UFRN - Federal University of Rio Grande do Norte – Natal, Brazil

{cnunes,claudios,ioliveira,lucena}@inf.puc-rio.br, uira@dimap.ufrn.br

Abstract. *This paper presents a quantitative study of development and evolution of a multi-agent systems product line (MAS-PL). The investigated MAS-PL is obtained from the initial version of a conference management web-based system, named Expert Committee (EC), that is gradually evolved to incorporate a series of change scenarios related to new agency features (autonomous behavior). The quantitative study consists of a systematic comparison between two different versions of the MAS-PL: (i) one version implemented with object-oriented techniques and conditional compilation; and (ii) the other one using aspect-oriented techniques. Our analysis was driven by fundamental modularity attributes, such as: separation of concerns, interaction between concerns, and program size.*

1. Introduction

One of the latest trends in software engineering is to produce techniques and tools that allow the development of families of similar products, instead of individual products. With the aim to address this need, over the last years, many approaches have been proposed for software product line development [6, 8, 14, 28]. Software product lines (SPLs) [6, 28] refer to engineering techniques for creating similar software systems from a shared set of software assets using a systematic method to build individual applications. Most of the existing SPL approaches [6, 14, 28] motivate the development of a flexible and reusable architecture to enable large-scale reuse. An SPL architecture addresses a set of common and variable features of a family of products. A feature [8] is a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in SPLs.

Similar to development of single-purpose systems, SPLs approaches also need to consider how to deal with the existence of evolution scenarios. In fact, due to frequent changes the evolution of SPL needs to be conducted with as minimum impact as possible, such as: introduction, modification or removal of optional, alternative, and crosscutting features. Thus, SPL architectures need to be stable and flexible to support such frequent changes. Therefore, different variability mechanisms must be analyzed and chosen to promote the stability of the architecture during the SPL evolution.

Recent research presents some studies and benefits of using aspect-oriented programming (AOP) techniques to improve the modularization of features in SPLs [2, 3, 10, 16, 24], object-oriented frameworks [22] or multi-agent systems [12, 30]. The increasing complexity of modern applications motivates the use of AOP [21], because it

avoids that crosscutting features will produce tangled, scattered and replicated code. All these problems can cause difficulties regarding the management, maintenance and reuse of common and variable features in SPLs. In fact, AOP has been proposed to allow a better modularization of crosscutting concerns. Therefore, it is supposed to improve the reusability and maintenance of complex systems. Although there are recent research and empirical studies exploring the use of AOP to modularize features, none of them analyzes the impact of adding agency features to an existing system.

Over the past few years, the agent technology has emerged as a new software engineering paradigm to allow the development of distributed complex systems [19, 32]. Only some recent research has investigated the integration synergy of multi-agent systems (MASs) and SPLs technologies, characterizing the development of Multi-Agent Systems Product Lines (MAS-PL) [26, 27]. An MAS-PL defines an SPL that uses software agents to model, design and implement its common and variable features.

In this context, this paper presents an empirical study of development and evolution of an MAS-PL with the aim to compare the modularity of object-oriented (OO) and aspect-oriented (AO) different implementations of the MAS-PL. Our MAS-PL has been developed from the evolution of a conference management web-based system, called ExpertCommittee (EC) [25]. In this MAS-PL, we have developed seven releases, focusing on several change scenarios. Each release of our MAS-PL contains new optional and alternative features that the previous version does not address. Each release was implemented separately using two sets of technologies: (i) an OO implementation in Java language with conditional compilation support; and (ii) an AO implementation in AspectJ[1]. Most of the new features are related to the introduction of agency features in the original system using MASs abstractions, such as, agents, roles and their associate behaviors. Our study is based on existing metrics suites for modularity analysis [29, 30]. These metrics have already been used in other case studies [10, 13, 23, 30]. Furthermore, we also report some initial lessons learned in how to design MAS-PL stable architectures in the context of evolution scenarios. The results of our study show that the AO implementation of the investigated MAS-PL presented better results in terms of separation and interaction between concerns; however, it exhibited worst results in relation to size metrics.

The remainder of this paper is organized as follows. The study settings are presented in Section 2. Section 3 presents the study results. Section 4 discusses some lessons learned. Related works and study constraints are presented in Section 5. Finally, final remarks are presented in Section 6.

2. Study Settings

This section describes the MAS-PL used in the context of our study. Initially, the feature model of the MAS-PL is described (Section 2.1). After that, we describe the development process of the MAS-PL releases (Section 2.2). The MAS-PL architecture is then presented in terms of the components and agents that compose the system (Section 2.3). The AO and OO versions of the MAS-PL design are presented in Section

[1] http://www.eclipse.org/aspectj/

2.4. Finally, the suite of metrics used in our quantitative study is described in Section 2.5.

2.1. The ExpertCommittee Web based System

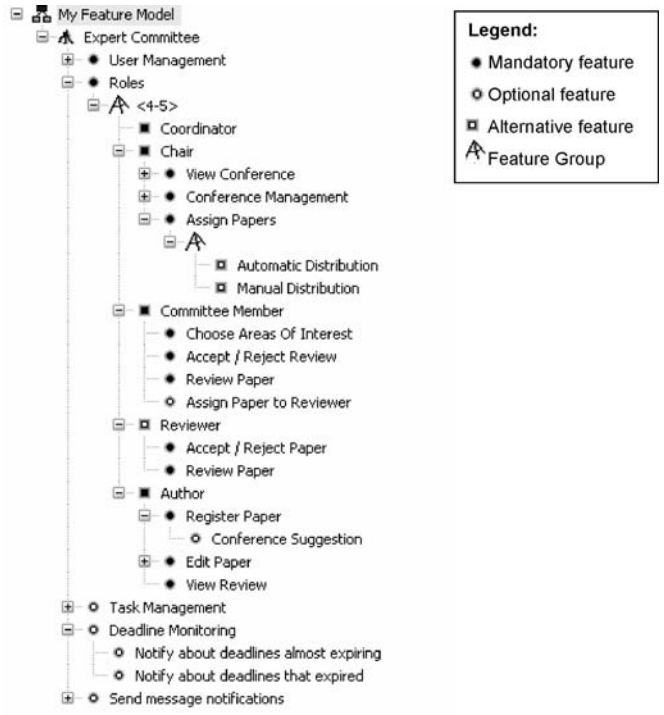

Figure 1. Simplified EC feature model.

The ExpertCommittee (EC) [25] is a typical web-based application that aims at managing the paper submission and reviewing processes from conferences and workshops. The EC system provides functionalities to support the complete process of conference management. Each of these functionalities can be executed by an appropriate user type, such as, chair, coordinator, program committee members and authors. Figure 1 presents a partial view of the EC feature model.

2.2. Generation of the MAS-PL Releases

During the development and evolution of our MAS-PL, we first implemented the SPL base architecture of the EC. After that, we applied a series of change scenarios, adding optional and alternative features in the SPL architecture. Seven new releases of the EC MAS-PL were generated. Each release of our MAS-PL was always implemented in two different versions: (i) one codified in Java with conditional compilation; and (ii) the other one codified in AspectJ. Each new release was also implemented based on the previous one. For example, the OO release 2 represents the evolution of the OO release 1.

Most of the change scenarios are related to the addition of new agency features. In order to implement these features, new software agents and roles have to be added. Table 1 summarizes the changes undertaken to implement the releases. During the evolution of the EC MAS-PL, we basically added three types of optional/alternative features: (i) *new conference management features* – these features introduce new functionalities related directly to the conference management process, such as the addition of support to program committee members assign papers to reviewers (reviewer role); (ii) *new autonomous behavior* – several software agents were introduced in the EC MAS-PL architecture (releases R3, R4, R5 and R7). Different agents were introduced in the system with the purpose to implement autonomous behavior related to recommendations to researchers (paper authors), deadline monitoring, pending tasks monitoring; and (iii) *new behaviors and roles for an agent* – added internal variabilities to the agents, such as new agent roles or behaviors. These types of features were modularized as: specific plans to be executed by the agent under

specific conditions (releases R5 and R6) and specific roles to be played by the agent in a specific context (release R3).

Table 1. Scenarios of change in MAS-PL.

Releases	Description	Change Type
R1	ExpertCommittee core	
R2	Addition of the Reviewer role.	Inclusion of optional feature.
R3	New feature added to include user agents including the author and chair roles. New feature to allow the suggestion of conferences to the authors.	Inclusion of optional feature.
R4	Addition of a Notifier agent to send messages to the system users through email and SMS.	Inclusion of optional and alternative feature.
R5	Addition of the Deadline agent. This agent is responsible for monitoring the conference deadlines.	Inclusion of optional feature.
R6	Addition of a feature that allows the chair to automatically assign papers to reviewers. Extension of the deadline agent to allow reminder deadlines.	Inclusion of alternative feature and extension of deadline monitoring feature.
R7	Addition of a Task agent.	Inclusion of optional feature.

2.3. The MAS-PL Architecture

The EC MAS-PL was structured according to the Layer architectural pattern [5]. It is composed of the following components/layers: (i) ***GUI***: this layer is responsible for processing the web requests submitted by the system users; (ii) ***Business***: is responsible for structuring and organizing the business services provided by the EC system; and (iii) ***Data***: aggregates the classes of database access, and it was implemented using the Data Access Object (DAO) design pattern [1]. Figure 2 illustrates the architecture of the EC web-based system and highlights the core architecture. In our implementation, the JADE framework[2] was used as the base platform to implement the software agents. These agents are responsible for monitoring the execution of different functionalities of the EC core system in order to provide the new agency features. The integration between the web architecture and the agents was accomplished by means of the adoption of the Observer pattern [33]. Next, a brief detail about the agents of the EC MAS-PL is presented: (i) `EnvironmentAgent` - this agent monitors the EC system by observing the execution of specific business execution and its aim is to notify the other agents of the MAS-PL about the system changes; (ii) `User Data Agent` – this agent receives notifications when new users are created in the database; (iii) `User Agent` – each user stored in the system has an agent that represents him/her in the system; (iv) `Deadline Agent` – this agent is responsible for monitoring the conference deadlines; (v) `Notifier Agent` – this agent receives requests from other agents to send messages to the system users; and (vi) `TaskAgent` - this agent is responsible for managing the user tasks. It receives requests for creating, removing and setting the execution date of tasks. The requests are made by the user agents.

[2] http://jade.tilab.com/.

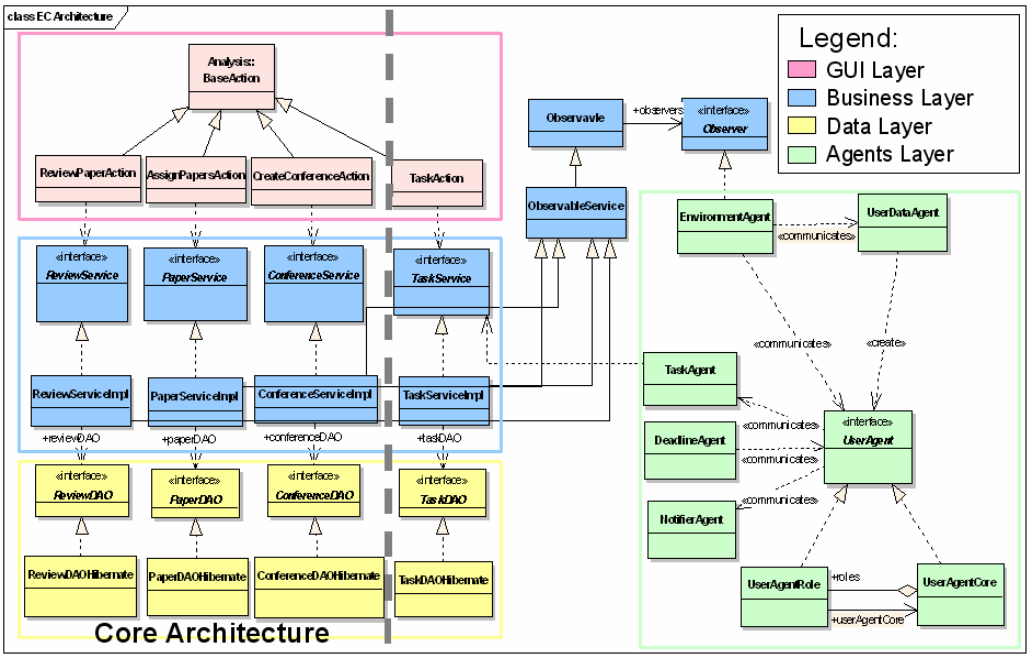

Figure 2. Expert Committee MAS-PL Architecture.

2.4. MAS-PL Object and Aspect Oriented Design

Figure 3 presents a partial class diagram of the OO implementation of the MAS-PL, illustrating the main components that were affected during the evolution of the system architecture. The OO releases were implemented using the Java programming language. The AO implementation is also structured following the Layer architectural pattern. Figure 4 shows a partial diagram of the AO implementation of the MAS-PL, illustrating a subset of its aspects. The `<<aspect>>` stereotype represents the aspects of the system. The dependency arrows represent that an aspect "crosscuts" the structure of system classes. The classes and aspects were marked in Figures 3 and 4 with a sequence of *Rs* above of them. This indicates whether a class or aspect was added (+Rx) or changed (~Rx) during the implementation of the release X. In the AO implementation (Figure 4), we cannot observe any changes in its classes and aspects, it happens because only new aspects were added.

2.5. The Metrics

In our study, we use a suite of metrics for quantifying three modularity attributes, named separation of concerns, interaction between concerns, and size [29, 30]. Initially, we have decided to focus on a restrict set of measures to evaluate the modularity of the OO and AO versions of the investigated system. The main goal of this study was to evaluate the separation of concerns (scattering and tangling), interaction between concerns, and program size of the MAS-PL. These metrics have already been used in other case studies [10, 13, 23, 30, 34]. The metrics capture important attributes in terms of design and code - for example, components (classes and aspects), operations (methods and advices), and lines of code (LOC). The counting of the separation of concerns (SoC) and interaction between concerns metrics requires a mapping ("shadowing") of the features to the source code [29, 30]. Table 2 briefly presents each metric used in this work.

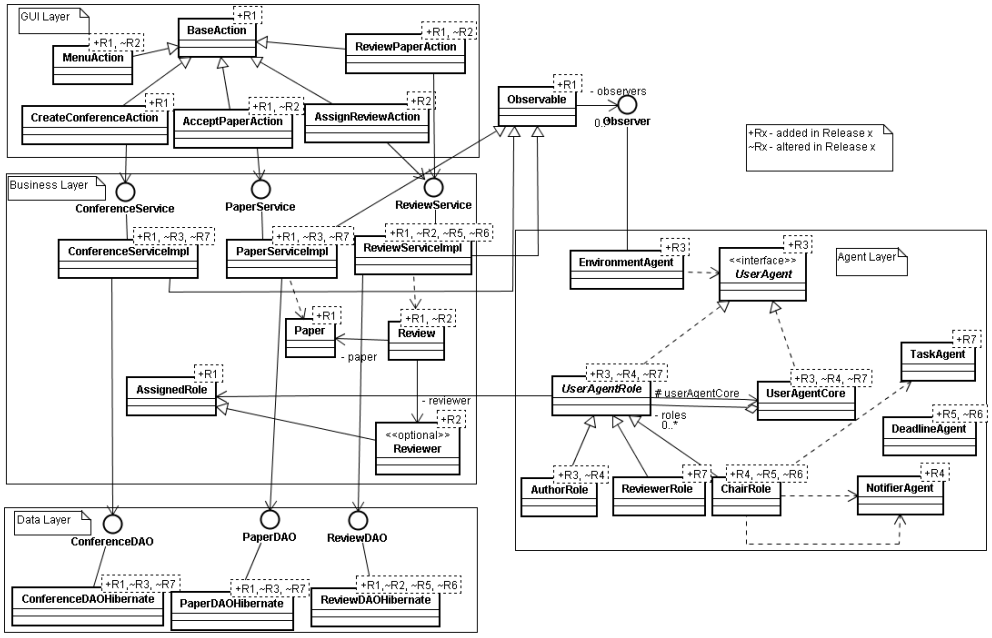

Figure 3. OO EC MAS-PL Simplified Architecture.

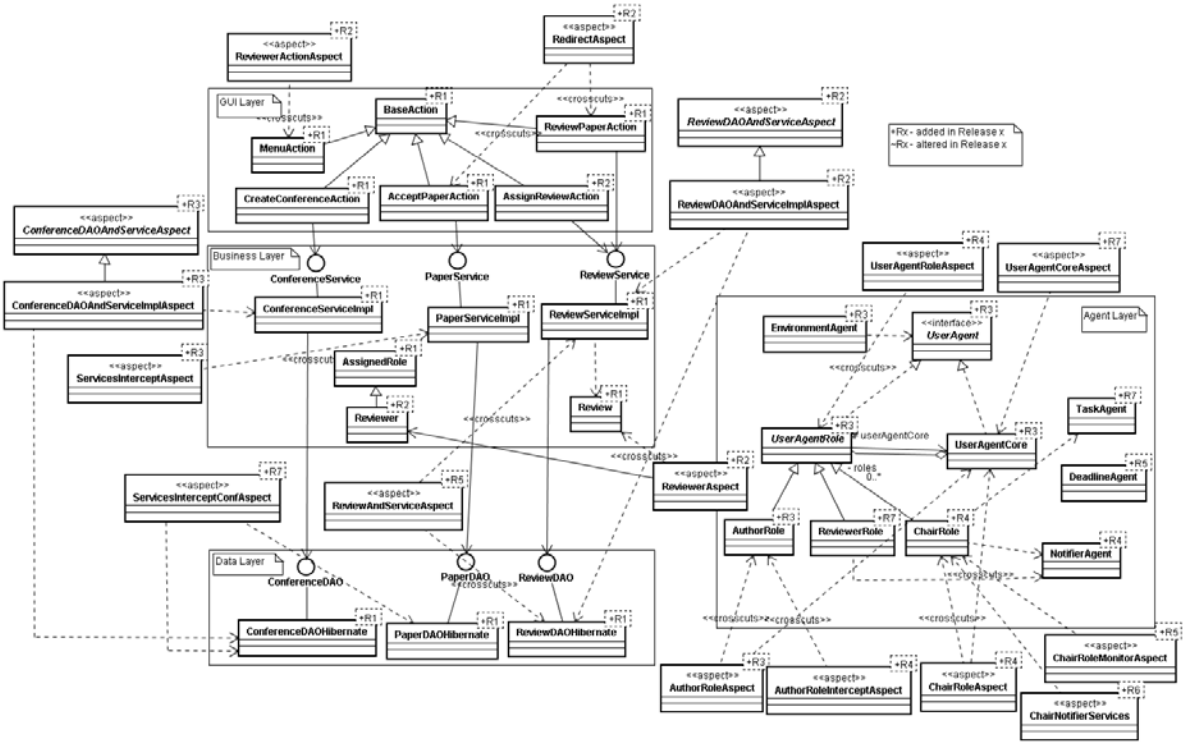

Figure 4. AO EC MAS-PL Simplified Architecture.

Table 2. The Metrics applied.

Attributes	Metrics	Definition
Separation of Concerns	Concern Diffusion over Components (CDC)	It counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.

	Concern Diffusion over Operations (CDO)	It counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.
	Concern Diffusion over LOC (CDLOC)	It counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Interaction Between Concerns	Component-level Interlacing Between Concerns (CIBC)	It counts the number of other concerns with which a given concern shares at least a component.
Size	Lines of Code (LOC)	It counts the lines of code.
	Number of Components (NOC)	It counts the number of components (classes and aspects).
	Number of Operations (NOO)	It counts the number of operations of a given component.

3. Modularity Analysis

This section presents the obtained results of the metrics for separation of concerns (Section 3.1), size (Section 3.2), and feature interaction (Section 3.3).

3.1. Separation of Roles and Agents Concerns

In our study, we have analyzed three optional features of the EC MAS-PL using the separation of concerns (SoC) metrics. These selected features represent multi-agent abstractions (roles or agents) that modularize relevant agency features of the MAS-PL. In addition, these features were also chosen because they were added to the SPL during the first three evolution scenarios. This allowed us to analyze the behavior of these features throughout the last three releases (Section 2.2). Figure 5 presents the results for the Reviewer role, which is an optional feature added in release 2 (Table 1). The results show that the Reviewer role is scattered over fewer components and operations (CDC and CDO metrics) and tangled with fewer features in the AO implementation (CDLOC metric). This indicates that the AO implementation was more effective to modularize this feature when compared with the OO implementation. This occurred because in the AO solution the codes in charge of realizing the optional roles are transferred from classes to a set of dedicated classes and one or more glue aspects. In the AO implementation of SPLs, aspects usually play an excellent role as the glue between the core and optional features [3, 23]. The conditional compilation technique, adopted in the OO solution, lacks this ability because it has a somewhat intrusive effect on the code, due to the need to add the *#ifdef* and *#endif* clauses locally at the places where features intersect.

In the OO implementation, the Reviewer feature is spread over a number of classes, such as: `Reviewer`, `Review`, `ReviewPaperAction`, `AssignReviewAction`, and `ReviewDAOHibernate`. Note, in Figure 3, that these classes underwent changes in release 2 (symbol ~R2). These changes were carried out in order to introduce code related to Reviewer role in the mentioned classes. Also note that the `Review` class has a direct association with the `Reviewer` class. The `Reviewer` class was introduced in release 2 and is totally dedicated to implementing the `Reviewer` role. In the AO implementation (Figure 4), part of the Reviewer role is implemented by the Reviewer class and three aspects: `ReviewerAspect`, `RedirectAspect`, and

ReviewDAOAndServiceImplAspect. These aspects introduce the Reviewer role behavior in the classes Review, ReviewPaperAction, AssignReviewAction, and ReviewDAOHibernate, which are free from code related to the Reviewer role. This is the main reason for the decreasing of the degree of scattering and tangling in the AO solution, reflected in the SoC metrics. Note that these four classes were not changed in release 2 of the AO implementation (Figure 4). Also note that the ReviewerAspect aspect works as glue between the Reviewer and Review classes (Figure 4). In Figure 5, we can see that the tangling of the Reviewer feature with other features is largely higher in the OO implementation (CDLOC metric). On the other hand, the scattering of the Reviewer feature over operations (CDO metric) and components (CDC metric) is almost the same in both implementations. This occurred because while the aspects modularize and isolate the Reviewer concern in the AO implementation, this concern does not present similar pieces of code in different classes, such as a Logging concern, for example. In the OO and AO implementations of release 7, there is a significant increase in all the metrics because the inclusion of the Task Agent feature includes several event classes that communicate with the Reviewer feature. Besides, there are changes in other classes that represent the roles and they also communicate with the reviewer feature, such as: chair and committee member.

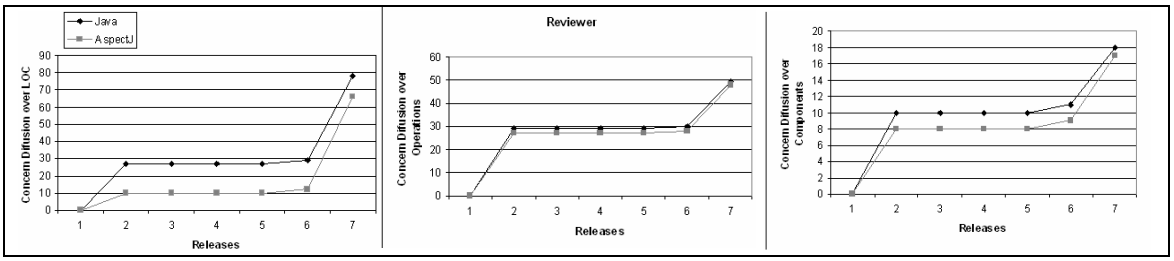

Figure 5. SoC Metrics for Reviewer feature.

Figure 6 shows the results for the User Agents feature, which is also an optional feature, in terms of SoC metrics. For this feature, the collected values for the AO solution did not present better results compared to the OO solution in terms of CDO and CDC metrics. The number of operations of the User Agents feature increased through the evolution of the MAS-PL because new operations were added in the MAS-PL core using inter-type statements to implement this feature in order to enable the aspects to affect specific join points of the MAS-PL core. Figure 3 shows that in the OO implementation, the User Agents feature is spread over fewer components (classes or aspects). This happens because with conditional compilation in OO implementation, it is only necessary to add the *#ifdef /#endif* clauses locally in a few classes. Thus, the degree of scattering presents low values in the OO solution. The UserAgent, UserAgentCore, and UserAgentRole classes were introduced in release 3 and are totally dedicated to implementing the User agents feature in the OO and AO implementations. Note in Figure 3 that the classes added in release 3 are modified during the evolution of the MAS-PL. The changes in these classes increase the tangling (CDLOC metric) as can be seen in Figure 6. In the AO implementation (Figure 4), a significant part of the User Agents feature is implemented by the UserAgent, UserAgentCore, and UserAgentRole classes and a set of aspects that affects the specified roles such as: AuthorRoleAspect, ChairRoleAspect. Because of this, the degree of scattering is high in the AO solution (CDC metric). Thus, the User Agents

feature is more scattered in the AO solution (CDC and CDO), but less tangled than the OO solution (CDLOC). This high number of aspects can be a negative point to AOP, because the AO solution can harm the understanding of the feature, since there are more components to deal with.

Figure 7 shows the results of the CDC metric for the Notifier Agent feature. During the evolution of the MAS-PL, there is a significant increase in the number of components for the AO implementation from release 5. This occurs because, in the OO implementation, the notification code from specific system events was codified directly in the classes using *#ifdef/#endif* clauses. On the other hand, in the AO implementation, different aspects were created to intercept these existing classes. Different aspects were created because each of them is related to one specific role of user agent which needs to be managed separately in order to guarantee an easy inclusion/removal of the optional feature that it represents. Thus, in the AO implementation the code was modularized in separated aspects, such as: `AuthorRoleInterceptAspect`, `ChairNotifierServices` (Figure 4).

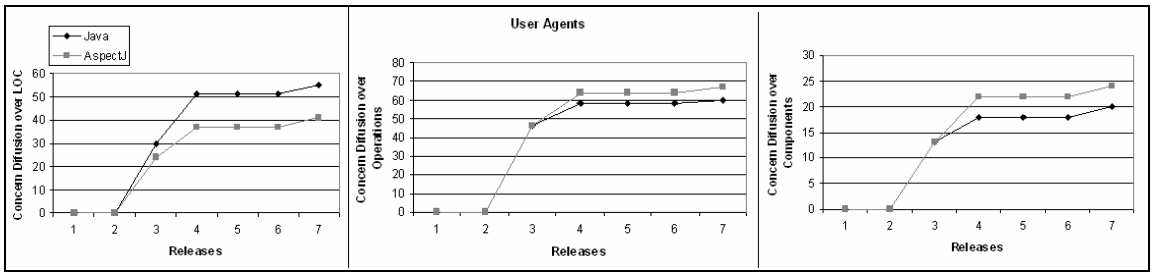

Figure 6. SoC Metrics for User Agents feature.

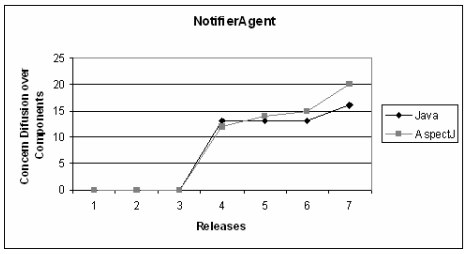

Figure 7. CDC Metric for Notifier Agent feature.

3.2. Size

Figure 8 presents the results of the following size metrics: Lines of Code (LOC), Number of Components (NOC) and Number of Operations (NOO). The results showed that the collected values for the AO implementation were higher when compared to the OO implementation. This happened because most of the aspects created during the evolution are heterogeneous. A heterogeneous aspect is an aspect that affects multiple classes and join points in different ways by introducing different behavior in each of them. This causes the creation of many aspects (increasing the values collected for the NOC metric), each of them with different advices and pointcuts (LOC and NOO metrics are higher in the AO implementation) affecting the system classes. While in the OO implementation, the use of conditional compilation with the addition of AND and OR operators in the existing classes were sufficient to support the combination of determined features, such as: User Agents and Notifier Agent. The AO implementation

required the creation of new and heterogeneous aspects to represent those combinations of features, such as: `AuthorRoleAspect`, `AuthorRoleInterceptAspect` (Figure 4).

The results obtained for the size metrics showed that although the AO implementation improved the modularization of agency features (Section 3.1), the high values obtained for the NOC, LOC, and NOO metrics can bring difficulties to the understanding and evolution of the SPL, because new aspects needs to be managed.

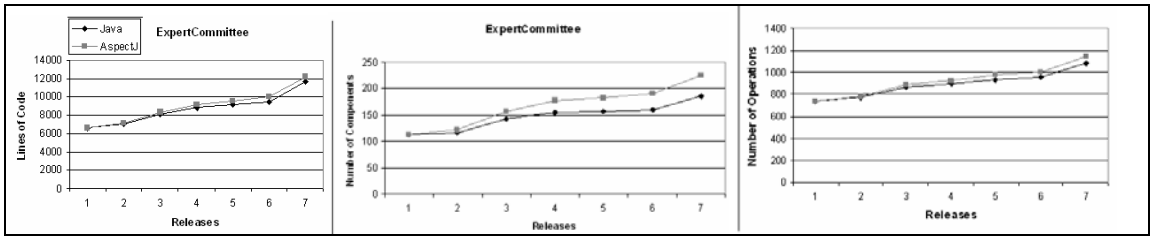

Figure 8. Size Metrics of the Expert Committee.

3.3. Feature Interaction Analysis

Figure 9 shows the results of the Component-level Interlacing Between Concerns (CIBC) metric [29] (Table 2). This metric aims at quantifying the interaction between concerns. Figure 9, for example, shows the interaction of the Reviewer and User Agents features with the other MAS-PL concerns (Roles: Author, Chair, CommitteeMember, Coordinator; ACLMessage, Persistence, Review, and MessageFactory). According to Figure 9, the Reviewer feature is tangled with fewer concerns in the AO implementation. This occurred because the AO implementation transferred almost all the elements in charge of realizing this feature from conventional components (`Reviewer`, `Review`, `ReviewPaperAction`, `AssignReviewAction`, and `ReviewDAOHibernate`) (Figure 3) to some aspects (`ReviewerAspect`, `RedirectAspect`, and `ReviewDAOAndServiceImplAspect`) (Figure 4). This contributed to separating this feature from the other concerns. Figure 9 also shows the CIBC metric for the User Agent feature. Note that the degree of interaction between the User Agent feature and other concerns presented lower values in the AO solution along the MAS-PL evolution. This was due to the same reasons noted for the Reviewer feature.

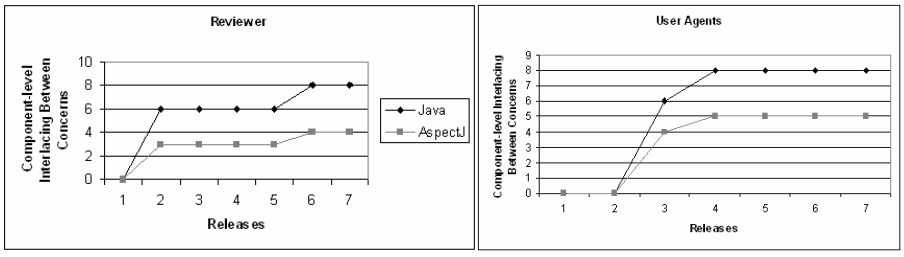

Figure 9. CIBC Metric.

4. Lessons Learned

This section discusses some benefits and drawbacks of using AO techniques in order to facilitate the inclusion of optional and alternative features in MAS-PLs.

Aspects as "glue" between Roles and Agents. During the evolution of the MAS-PL, several aspects in our MAS-PL worked as a "glue" code between the OO core structure and the different optional and alternative agency features added to this core. This design decision is very useful because it allows injecting new properties and behaviors (agency features) in a transparent way into the base OO structure. Also, the use of aspect-oriented technologies makes it easy to remove specific agency features or replacing them with other implementation (alternative features). The main example in our study of the use of the aspects as "glue" code was the implementation of the Reviewer feature (Section 3.1).

Feature Management. During the MAS-PL evolution the addition of some agency features, such as: User Agents and Notifier Agent, caused a high number of new components (classes and aspects) in the AO solution. Althought the use of aspects increases the number of components in these cases, it was also useful to reduce the tangling and coupling between the concerns/features. Thus, the AO solution was more effective to modularize these features (Section 3.1). With this modularization these aspects can be better managed, because it allows the easy inclusion and removal of the feature that it represents.

Code Understanding. In our study, it was observed that the AO solution exhibits higher values for all the size metrics (LOC, NOC, and NOO). This was a negative finding of the AO implementation because it can demand the understanding of additional code in the new aspects added to the system, and thus harming the evolution of the MAS-PL. Therefore, a trade-off analysis is required to determine if the benefits in terms of separation of concerns, demonstrated for the MAS-PL features in Sections 3.1 and 3.2, can overcome the occasional difficulties to deal with the additional components (classes and aspects), operations and lines of code brought by an aspect-oriented implementation.

5. Related Work and Study Constraints

5.1. Related Work

Recent research presents some studies with the use of AOP in SPL development [3, 7, 16, 18]. There are also some empirical studies comparing objects, aspects and agents technologies [12, 15, 30].

Figueiredo et. al. [10] present an empirical study focusing on evolution requirements of two product-lines, called MobileMedia and BestLap. This work analyzes the evolution of product lines in terms of metrics for modularity, change propagation and feature interaction. In order to provide the variation of SPL, two implementation techniques were considered: conditional compilation and AOP.

Apel & Batory [4] present a study comparing the feature-oriented programming (FOP) and aspect-oriented programming (AOP) mechanisms to implement features of a product line. In order to compare these mechanisms, they used a restricted set of metrics. The SPL implementation used AML (Aspectual Mixin Layers), which is an approach to integrate FOP and AOP. The metrics used in SPL were: lines of code and number of components (classes, mixins and aspects).

Kastner et. al. [20] present a case study on refactoring legacy application into an SPL using aspects to implement features. Their case study was the Berkeley DB

database system. The goal of their work was to implement features using AspectJ in order to show the suitability of this language. As a general result, they observed a strong coupling between classes and aspects that makes the maintenance and evolution of the SPL difficult.

Although there are some empirical studies exploring the use of AOP, most of them focus on the modularization of conventional crosscutting concerns such as: persistence [23, 31], exception handling [11] and design patterns [13, 17]. None of the cited works analyze the impact of adding agency features in evolution scenarios of a MAS-PL. We consider a different approach of other works, that is the MAS-PL and the several change scenarios applied to the core architecture focusing on the quantitative assessment of AO and OO solutions. Figueiredo et al [10] present an empirical study focusing on the evolution of SPLs, although in different domains from the MAS domain that we explored in our work. The other works do not also focus on MAS-PL domain and they do not apply the same suite of metrics we apply. The goal of the work was to compare quantitatively and qualitatively the use of OO techniques with conditional compilation against AO techniques/mechanisms in a specific scenario: the evolution of web-based systems with the incorporation of new agency features in a MAS-PL.

5.1. Study Constraints

Although only one case study was presented in this paper, it is a representative web-based system, implemented using several mainstream technologies. Thus, we tried to perform real change scenarios that could be applied in other web-based modern systems. The goal of our study was to compare two different implementation technologies (OO with conditional compilation and AO techniques) in the development and evolution of an MAS-PL. The AO implementation was developed using the AspectJ language. We use the AspectJ due to its stability and because it is widely used and the most consolidated aspect-oriented language. Moreover, other works cited above also used AspectJ to implement different SPLs. Another important point was the metrics used in this work. The metrics used in this work have already been used and validated in several recent empirical studies [10, 15, 23, 30, 34].

In order to count the separation of concerns metrics, it is necessary to do the “shadowing” of the code to verify the piece of code that implements a determined feature in MAS-PL. This process is done manually. For gathering the values of the size metrics, we used a plug-in for Eclipse (`www.eclipse.org`). The changes applied to the OO and AO versions have been done by two people and one person, respectively. Both with good knowledge on Java and AspectJ programming. First, we have implemented the OO version and after the AO version. During the development we have used the same design practices throughout all OO and OA EC MAS-PL releases, such as, design and layer architectural patterns.

6. Final Remarks

This paper presented a quantitative study of the development and evolution of a multi-agent system product line (MAS-PL). In this study, we compared two different versions of the MAS-PL implemented using the following technologies: (ii) OO with conditional compilation using the Java language; and (ii) AO programming using the AspectJ language. We initially developed a traditional web-based system to support the conference management process. Subsequently, we evolved this system to incorporate a

series of change scenarios in MAS-PL. Most of changes were related to new agency features (autonomous behavior). To compare the different releases of the OO and AO implementations, we used a suite of metrics already used in other case studies that makes it possible to analyze the following properties of systems/product lines: separation of concerns, interaction between concerns, and size. The results of our quantitative study showed that the AO implementation exhibits a better separation of concerns/features (CDLOC metric) and reduced values for the interaction between concerns, but in terms of the CDC and CDO metrics the AO implementation presented superior values. Besides, the NOC, NOO and LOC increased in the AO implementation, bringing complexity to manage the new aspects that modularize some agency features. In addition, we also presented some initial lessons learned from our study and we discussed the benefits of modularizing the features in an MAS-PL.

References

[1] Alur, D., et al. (2001). Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[2] Alves, V., et al. (2006). Refactoring product lines. In *Proceedings of the 5th GPCE'05*, pp. 201–210, New York, NY, USA. ACM.

[3] Alves, V., et al. (2005). Extracting and evolving mobile games product lines. In *Proceedings of the 9th SPLC'05*, pp. 70–81. Springer.

[4] Apel, S. and Batory, D. (2006). When to use features and aspects?: a case study. In *Proceedings of the GPCE'06*, pp. 59–68, New York, USA. ACM.

[5] Buschmann, F., et al.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons (1996).

[6] Clements, P. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, USA.

[7] Colyer, A., et al. (2004). On the separation of concerns in program families. Technical report, Computing Department, Lancaster University.

[8] Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison Wesley Longman.

[9] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

[10] Figueiredo, E., et al. (2008). Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th ICSE'08*, pp. 261-270.

[11] Filho, F. C., et al. (2006). Exceptions and aspects: the devil is in the details. In *SIGSOFT FSE*, pp. 152–162.

[12] Garcia, A., et al. (2003). Agents and Objects: An Empirical Study on the Design and Implementation of Multi-Agent Systems. In *SELMAS'03*, pp. 11–21, USA.

[13] Garcia, A., et al. (2005). Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th AOSD'05*, pp. 3–14, NY, USA. ACM Press.

[14] Greenfield, J., et al. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley and Sons.

[15] Greenwood, P., et al. (2007). On the impact of aspectual decompositions on design stability: An empirical study. In *Proceedings of ECOOP'07*, LNCS, pp. 176–200.

[16] Griss, M. L. (2000). Implementing product-line features by composing aspects. In *Proceedings of the first SPLC*, pp. 271–288, Norwell, MA, USA.

[17] Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in Java and aspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN*, pp 161–173.

[18] Hunleth, F. and Cytron, R. K. (2002). Footprint and feature management using aspect-oriented programming techniques. *SIGPLAN Not.*, 37(7):38–45.

[19] Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41.

[20] Kastner, C., et al. (2007). A case study implementing features using aspectj. In *Proceedings of the 11th SPLC'07*, pp. 223–232,Washington, DC, USA.

[21] Kiczales, G., et al. (1997). Aspect-Oriented Programming. In *Proceedings of ECOOP 1997*, v. 1241, pp. 220–242, Berlin, Heidelberg. Springer-Verlag.

[22] Kulesza, U., et al. (2006a). Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In *ICSR'06*, pp. 231–245, Torino.

[23] Kulesza, U., et al. (2006b). Quantifying the effects of aspect-oriented programming: A maintenance study. In *Proceedings of the 22nd ICSM'06*, pp. 223–233.

[24] Lee, K.,et al. (2006). Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *Proceedings of the 10th SPLC'06*, pp. 103–112,Washington, DC, USA. IEEE Computer Society.

[25] Nunes, I., et al. (2008). Developing and evolving a multi-agent system product line: An exploratory study. *In 9th AOSE'08*, pp. 177-188, Estoril, Portugal.

[26] Pena, J., et al. (2006a). Managing the Evolution of an Enterprise Architecture Using a MAS-Product-Line Approach. *Software Engineering Research and Practice*, pp. 995–1001. CSREA Press.

[27] Pena, J., et al. (2006b). Building the core architecture of a multiagent system product line: with an example from a future nasa mission. In *7th AOSE'06*. LNCS.

[28] Pohl, K., et al. (2005). *Software Product Line Engineering:Foundations, Principles and Techniques*. Springer-Verlag, New York,USA.

[29] Sant'Anna, C., et al. (2007). On the modularity of software architectures: A concern-driven measurement framework. In *ECSA'07*, v. 4758, pp. 207–224.

[30] Sant'Anna, C., et al. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings XVII of SBES'03*, pp. 19–34.

[31] Soares, S., et al. (2006). Distribution and persistence as aspects. *Software Practice Experience*, 36(7):711–759.

[32] Wooldridge, M. and Ciancarini, P. (2000). Agent-Oriented Software Engineering: The State of the Art. *First Int. Workshop on AOSE*, v. 1957, pp. 1–28.

[33] Gamma, E., et al.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1995).

[34] Eaddy, M., et al. (2008): Do crosscutting concerns cause defects?. *IEEE Transactions on Software Engineering*, to appear.