

Scenario Based Resolution of Aspect Interactions With Aspect Interaction Charts

Shubhanan Bakre and Tzilla Elrad
Computer Science Department
Illinois Institute of Technology
Chicago, Illinois, 60616 U.S.A.
{bakrshu, elrad}@iit.edu

ABSTRACT

Introduction of aspects into the system raises the level of separation of concerns within the system. At the same time it also raises the level of interactions among the various components and features within the system. Current modeling techniques (sequence diagrams, live sequence charts) are inadequate in handling this added level of interaction. A higher level of abstraction is needed in order to capture the interactions among aspects/features/core and provides two immediate benefits - better modularization of the requirements, and better adaptability for the resulting model. We propose the Aspect Interaction Charts (AIC) that build on top of the Live Sequence Charts (LSC) [3] in order to capture the interactions among various aspects at joinpoints. With the AIC we foresee the ability to capture aspect interactions at a joinpoint in a common specification in the form of use-case scenarios, and the ability to execute these scenarios while non-invasively manipulating the interactions among the various aspects. In addition to the aforementioned benefits, we plan on leveraging the tools that come with the LSC language, i.e. the Play Engine. The AIC would provide us with the ability to model, view and manipulate aspect interactions at joinpoints.

Categories and Subject Descriptors

D.2.1 [Requirements/Specifications]: Languages; D.2.2 [Design Tools and Techniques]: Miscellaneous

General Terms

Design, Languages

Keywords

Aspect interactions, scenarios, LSC, aspect oriented modeling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop AOM '07, March 12-13, 2007 Vancouver, British Columbia, Canada

Copyright 2007 ACM 1-59593-658-5/07/03 ...\$5.00.

1. INTRODUCTION

Aspects improve the modularity of a software system. At the same time aspects introduce new forms of interactions among the different components. Three kinds of interactions among aspects/core are found in aspect oriented literature - spectative, regulative and invasive. Spectative aspects only observe the state at joinpoints. Regulative aspects observe states at joinpoints and control the flow at joinpoints based on the state. Invasive aspects, in addition to the above can also modify states at joinpoints. Understanding and architecting these interactions is a very complex task, considering the fact that a) aspects apply across the whole system b) multiple aspects act at common joinpoints c) aspects can directly/indirectly modify the state at joinpoints thereby impacting the behavior of the other components/aspects of the system.

Various attempts have been made in order to address the issues related to aspect interactions. [2] propose a language for detection and resolution of conflicts among aspects. [7] investigate the use of slicing for analyzing feature interactions. [5] makes use of superimposition in order to detect and resolve conflicts among aspects.

We believe that the problem of architecting aspect interactions and resolving aspect interference is a responsibility of the system designer and hence it should be addressed at the modeling stage of the software life cycle. But, current modeling languages and tools are not adequate to capture these interactions first-hand. In order to fill in this gap, we propose the aspect interaction chart (AIC).

An AIC captures the interactions among various aspects and objects at a common joinpoint (or a set of joinpoints) in a graphical but formal manner. AIC builds on top of the live sequence charts (LSC) language, which captures interactions among components in the form of scenarios. LSCs capture requirements in a formal specification. The Play Engine enables the user to feed in scenarios using the play in mechanism and test them using the play out mechanism [4].

We plan on leveraging and extending the capabilities of the play engine in order to enable us to play in scenarios of aspect interactions (AIC) and to play out these scenarios in order to be able to analyze and refine the interactions.

The paper is organized as follows. We give a brief introduction to LSCs in section 2, followed by the motivation behind AIC in section 3. Section 4 lays out the proposed AIC language extension in more detail and an example. Discussions and related work follow in section 5 and 6 followed by conclusions in section 7.

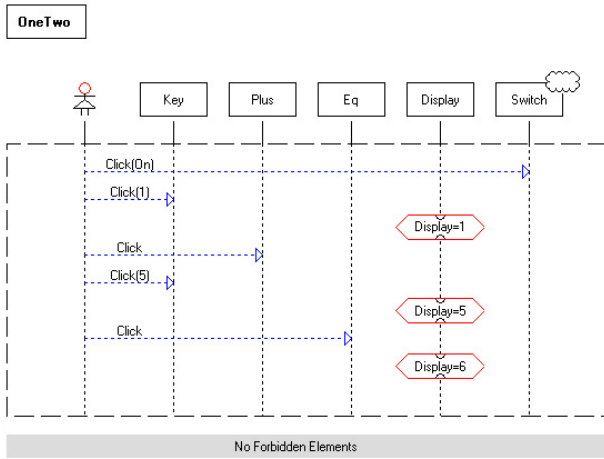


Figure 1: A Sample Existential LSC

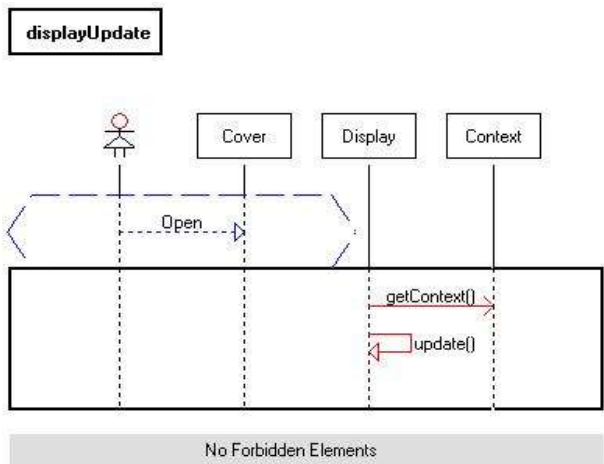


Figure 2: A Sample Universal LSC

2. BACKGROUND

Live sequence charts(LSC) is a language to specify scenarios of interaction between objects. This approach fits into the requirements stage of software development, however as illustrated in [3], when used along with **play-in**, **play-out** and the **Play Engine** it could replace some of the later phases of software development as well. Use cases in UML provide similar functionality, however LSC has a richer set of constructs. LSCs are an extension of message sequence charts (MSCs), in that they improve it by adding abilities for specifying liveness and timing constraints (and a few other improvements like expressing real-time behavior).

An LSC is made up of the following components. Object instances are shown as boxes with vertical lines below them. Messages are shown as arrows and can be synchronous or asynchronous. The beginning and end of a LSC are marked with small circles. These are called instance locations. Messages can be **hot** (solid red lines) or **cold** (dotted blue lines) depending upon its behavior. A hot message must be received after it was transmitted. A cold message may be sent and not received. An LSC can be either **existential** or **universal**.

The semantics of an existential LSC states that there should be at least one possible way in which the LSC can be satisfied. Existential LSCs are suitable for designing test cases. Figure 1 illustrates an existential LSC. Interactions within an existential LSC are enclosed in a dotted border line. A universal LSC (figure 2), on the other hand specifies behavior that must be satisfied by all possible system runs. A universal LSC consists of two parts: the **pre chart** and the **main-chart**. The pre chart specifies the precondition that must be satisfied in order for the main chart to hold. That is, if the pre chart does not hold then the main chart is not obligated to be satisfied. An LSC can also contain specification of forbidden behavior using forbidden messages, conditions and events. The events within an LSC are synchronized at the start and the end of a chart. However, there is no restriction on the sequence of events within the chart. An important property of a universal (and existential) LSC is that an event can activate more than one chart at a time (similar to the concept of pointcuts in AOP). The play engine is a graphical editor that can be used for inputting LSCs and executing them. The process of specifying an LSC in the play engine is called **play-in**. **Play-out** is the process of executing the LSCs. During the play-out process the user interacts with the system and the system reacts to the input based on all the scenarios present within it. More than one LSCs can become active for a given input.

LSC has a rich set of constructs that can be employed to AOSD in a very natural manner. For example, LSCs provide constructs for expressing forbidden interactions, which can be used for expressing forbidden aspect interactions. The pre chart of LSC can be employed to specify the pointcuts.

3. MOTIVATION

Aspects provide a new level of abstraction to software design. It introduces new levels of interactions within the software systems. Current modeling languages and tools are inadequate to capture these interactions in a modular fashion. LSCs and UML sequence diagrams capture scenarios of object interactions effectively, but interactions between aspects tend to remain scattered across different use cases.

Capturing aspect interactions in a single specification provides adaptability to the design. In an executable model it provides the designer the ability to swiftly try different possible combinations of aspect interactions. It also gives the designer the opportunity to detect potential conflicts. In general, a specification of aspect interactions provides a common place for laying out all the aspects interacting at a joinpoint and evolving the specification to derive the constraints needed for achieving the desired behavior.

LSC and the play engine provide important properties in order to achieve these goals. A universal LSC can be viewed as an aspect scenario with the pre chart forming the pointcut designator and the main chart forming the advice. The play engine provides the UI for play in and play out of these scenarios. However, the activation of the various LSCs (i.e. interaction among LSCs) is a part of the operational semantics of the LSCs. In other words, it is embedded in various use case scenarios scattered throughout the whole system. This is due to the fact that LSCs are limited in that they can express object interactions effectively. But, aspects being at a higher level of abstraction need a higher level construct in order to express interactions among aspects.

In order to address these issues the AIC was conceived.

We chose the LSC and the Play engine as the base for the AIC due to the important properties and tools that they provide as discussed earlier in this section and in section 2.

4. ASPECT INTERACTION CHARTS

In this section we specify the language constructs within AIC. AIC includes all basic language constructs of the LSC. That is, constructs like the pre chart, main chart, messages (cold and hot), conditions (cold and hot), and subcharts are included in AIC. However more advanced constructs like timing constraints and symbolic instances will be added as and when required. We also leverage the forbidden scenario language construct. The forbidden section specifies a scenario that should not occur in the main chart.

The formal semantics of these constructs is not discussed here. However, we discuss the informal behavior of these constructs within the AIC. Pre chart consists of message events between components and LSCs. Just as in an LSC, a pre chart in AIC can also have subcharts and conditions. In addition, a new message event is introduced in the pre chart of AIC. That is the before method event. This event helps us capture the before method call joinpoint.

The contract between the main chart and the pre chart remains the same as in an LSC. That is, when a pre chart is satisfied, the main chart is required to be satisfied. This contract is essential especially in detecting cases where two or more AICs are active (i.e. their main charts are executing) and have conflicting execution sequences. We could extend the LSC concept of hot/cold events to AIC messages and conditions in order to determine what happens when a main chart is not satisfied. So, in case of a cold event being invalidated, the chart exits gracefully. In case a hot event is invalidated the chart exits while raising an exception.

A constraint on the main chart of an AIC is that it cannot have interactions between object instances; LSCs should be used for that. Same is true for forbidden scenario section. In AIC, forbidden section should only express interactions between LSCs and not object instances. The behavior of other language constructs like the condition, sub-chart, message remains the same as in LSC.

In the rest of the section we illustrate the rudimentary structure of AIC with the help of an example. As an example we use the cell phone application. The use case scenario is as follows:

The user opens the flap of the cell phone. The system checks if enough battery power is available. If the keypad is locked, the display shows the unlock screen. The system communicates with the base station in order to establish a network in case a call is being received. Depending upon the context, speakers/mic or both are turned on. The display is updated according to the context.

Figures 3 to 7 depict the various scenarios involved in the use case. The LSC in figure 3 specifies that if the battery is low then shut down the system. The LSC in figure 4 specifies that if the security is on (i.e. if the keypad is locked) then display the login screen and prompt for the user login information. The LSC in figure 5 specifies that if the context is that of a receiving call then send an acknowledgment to the base station. The LSC in figure 6 specifies that if the context is that of the user receiving a call then switch on both the speaker and mic. The LSC in figure 7 specifies that when the cell phone flap opens the Display object should get the current context of the user and display the appropriate

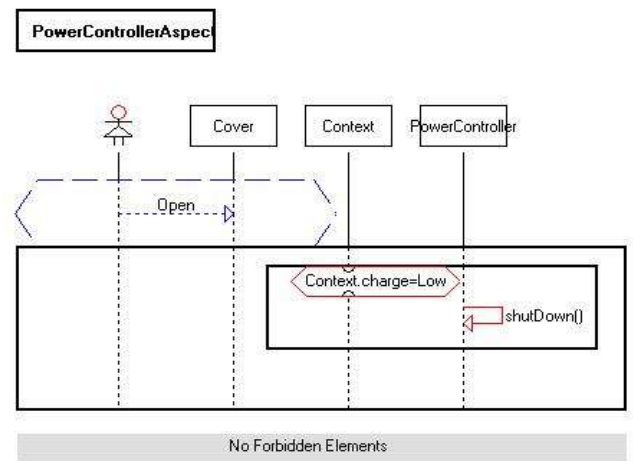


Figure 3: Power Control Scenario

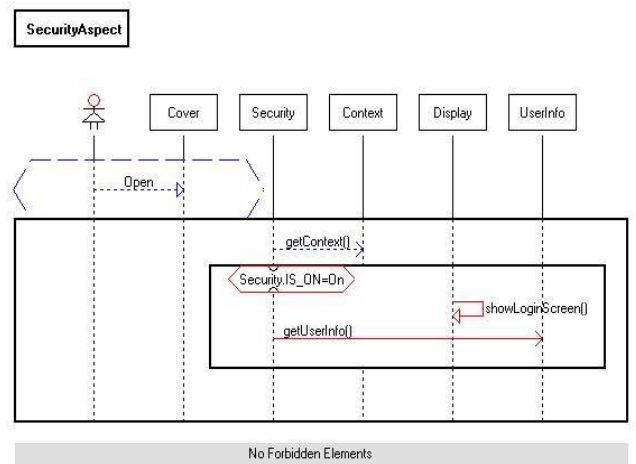


Figure 4: Keypad Locked Scenario

screen.

Note that these LSCs do not have any ordering information embedded in them at this point. So the Play Engine will execute them simultaneously till the LSC reaches the end or fails. In order to place an order on the execution of the LSCs each LSC would need constraints embedded in their pre chart. For example the pre chart of the Device Controller LSC should check for the ACK message being sent from the NetworkManager to the base station. In this manner the information about the dependencies among these scenarios is scattered throughout the different LSCs.

A higher level of abstraction would allow us to capture these dependencies in one place. We have two prospective solutions to address this issue. They are illustrated in figures 8 and 9.

An AIC looks similar in structure to an LSC in that it is made up of a pre chart and a main chart. If the pre chart is satisfied only then the main chart is obliged to be satisfied. However in addition to object instances, the AIC also consists of LSC instances. Since the AIC is designed for capturing interactions among the LSCs, interactions between object instances in the main chart are not allowed.

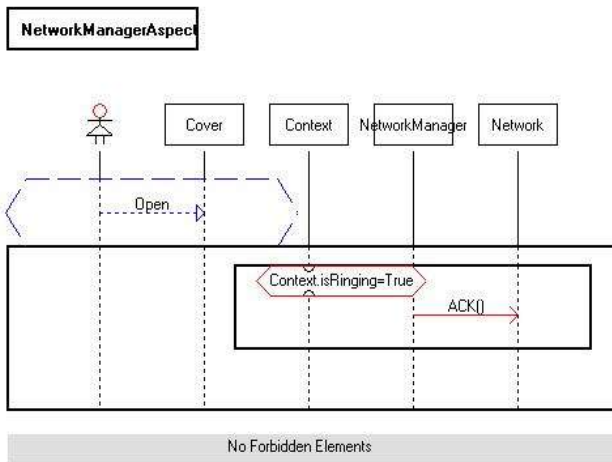


Figure 5: Network Management Scenario

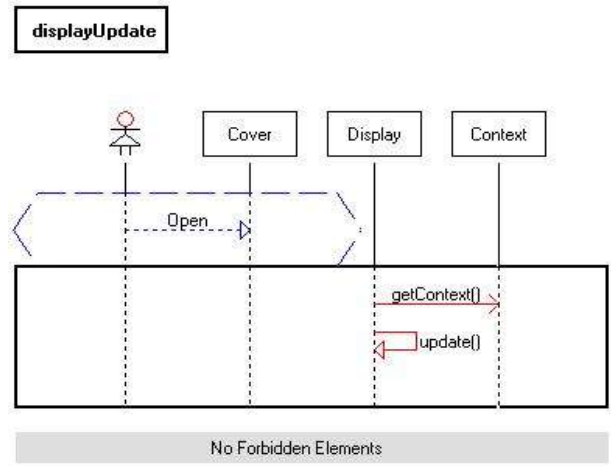


Figure 7: Display Update Scenario

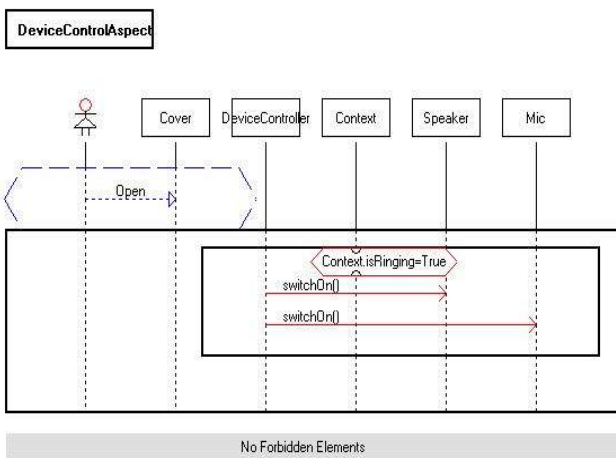


Figure 6: Device Control Scenario

Cell Phone Open Flap AIC

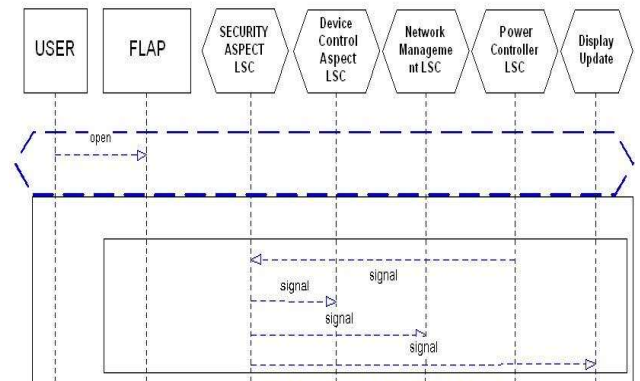


Figure 8: Open Cell Phone Flap AIC (method 1)

The execution within an AIC proceeds as follows. The pre chart consists of a set of conditions to be satisfied in order to trigger off execution in the main chart. This is similar in structure to the LSC pre chart, which consists of conditions and message events between object instances. The main chart is made up of LSC instances. Note that AIC inherits the ordering behavior of LSC, hence unless an order is placed explicitly on the LSC instances, a non-deterministic behavior will be seen in regards to the execution of the main chart.

The first proposed solution illustrated in figure 8 uses a signaling mechanism in order to trigger the LSC instances. In this method the LSC instance that completes execution signals the other LSCs that depend on it. The signal could be without any parameters or it could include parameters like message/event name to indicate the completion of that message/event. This could be useful in cases where an LSC is waiting on a subset of events/messages within the other LSC. For example the network management LSC of figure 5 might have additional steps to perform, but the device controller LSC just needs to know if the connection was established (i.e. ACK happened).

The down side of this approach is that it forces a strict sequence of execution on the LSC instances. This problem can be overcome by introducing signal-all or signal-multicast messages. Note that this increases the number of message events triggered in the system.

The alternative solution illustrated in figure 9 uses a message invocation mechanism. In this method we use the LSC construct SYNC in order to specify the LSC instances that need to be synchronized. And the AIC keyword **go** invokes the LSC instance. The SYNC could be annotated with condition, which could be of the form of a message event or a boolean predicate. This will facilitate the dependent LSC instance to proceed as soon as the annotated condition is satisfied instead of waiting for the conclusion of the other LSC.

Using this approach the user is not obliged to place a strict order on the execution of the LSC instances. In the example of figure 9 the SecurityAspect LSC and NetworkManagement LSC can proceed in any non-deterministic order after the PowerController LSC has completed execution. Method 2 appears to be more promising based on the current level of understanding, but further investigation is needed in order

Cell Phone Open Flap AIC

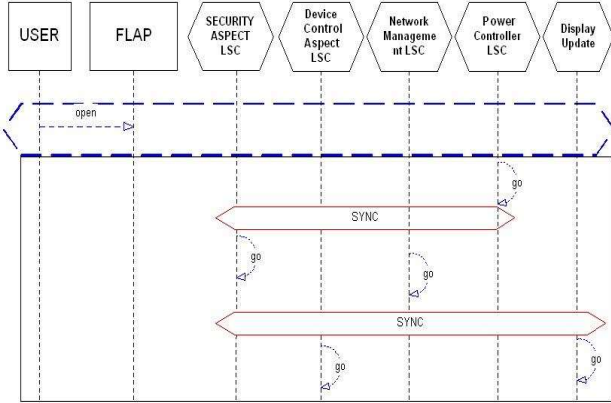


Figure 9: Open Cell Phone Flap AIC (method 2)

to arrive at a conclusion.

Introduction of AIC changes the behavior of LSCs in the following manner. Firstly, in order for an AIC to dictate the execution of an LSC, their pre charts should be similar. The execution in the pre chart of an AIC proceeds simultaneously with the pre chart execution of the LSC. When the pre charts of both AIC and LSC are satisfied, then the AIC main chart begins execution and invokes the LSC main chart as and when specified in the main chart of the AIC.

5. DISCUSSION

AICs introduce a level of abstraction to address the problem of expressing aspect interactions. A new level of abstraction provides better modularity to the use case scenarios which in turn makes it easy to (re)configure. The LSC language provides language constructs which are suitable for expressing aspect scenarios. The Play engine helps in feeding in and simulating the system model. This is very useful for simulating the system under different configurations. Note that AIC provides the mechanism to visualize and resolve aspect interactions. In case where a conflict occurs between two aspects, it would show up as an exception raised by an invalidated AIC or LSC during play out. This feedback can be used by the designer to refine the system further.

With respect to the two approaches for the implementation of the AIC, the first approach (figure 8) uses the signaling mechanism. The advantage of this approach is that the signals could contain parameters like event name in order to denote the completion of the respective event. However the disadvantage of this approach is the increased overhead on the system due to the signal-all and signal-multicast messages. The second approach (figure 9) uses an invocation mechanism. This method is more elegant in that it depicts the non-determinism in a natural manner (i.e. without the use of explicit signal-all or signal-multicast messages).

In order to support the AIC play in and play out, the Play engine will need to be extended. In order to improve the process of specifying aspect interactions, we visualize the following feature support for the Play engine. The AIC should be generated automatically when more than one LSC has the same pre chart. In other words, when two aspects

share a joinpoint, it needs to be captured in an AIC. The generated AIC will impose no constraint over the LSC instances within it. It is the designer's responsibility to add constraints over the LSC instances as and when necessary. The Play engine plays an important role in the scheme of things because it facilitates the translation of informal requirements into formal notations of LSC [3] and AIC via play in and play out.

In our view, interaction among aspects introduces three main issues - 1) synchronization, 2) data/state sharing and mutual exclusion between aspects, 3) access control at joinpoints. Our proposed solution solves the first problem completely by providing language support for constraining the execution of aspects. It solves the second problem partially, in the sense that we can use synchronization in order to achieve mutual exclusion. However, this can be over restrictive and more investigation is needed to provide language support for achieving this. Our solution currently does not solve the third problem. This problem involves controlling access to aspects at joinpoints. One should be able to specify within the AIC what (type) of aspects can be triggered at the respective joinpoints.

6. RELATED WORK

Investigations in the field of aspect interactions is still at an early stage. Attempts have been made at the different stages of software development in order to address this problem.

[2] provides criteria for strong independence and, independence with respect to an aspect. Conflict resolution of parallel composition of aspects is discussed. Here the assumption is that sequential composition of aspects will not cause conflict between aspects. The authors also provide means for defining scope of an aspect, which can be used to define aspects of aspects. This is a plus since it enables support for symmetric AOP approach.

[7] discuss an approach that involves slicing of aspect and core code in order to create models that can be used for analyzing feature interactions. But due to inadequate information the models generated were not able to convey any meaningful information. However, this does not mean that slicing is not a good technique for capturing aspect interactions. Far from that, it only shows that additional information is necessary in order to be able to gain meaningful information.

The approach in [5, 9] categorizes aspects into three types: speculative, regulative and invasive. Composition of superimpositions and aspects can be done using sequential or merging combination. Whether a combination is successful depends upon the pre-condition, post-conditions of the individual elements and also the global condition. This type of conflict detection is absent in current AOP languages.

Some aspect oriented languages address the issue of execution of two or more advices at a joinpoint by introducing precedence relation among the advices. Examples of this approach are [1, 10]. This approach suffers from the problem of tangling of such information within the aspect specification. Information related to the ordering of aspects must be independent of the aspect specification in order to preserve the modularity of the aspect specification.

[8] makes use of a composition strategy in order to impose an order on the composition of aspect models with the primary model.

7. CONCLUSION

In this paper we have presented new level of abstraction for handling aspect interactions. We have proposed the AIC which would allow us to specify aspect interactions in a modular manner. The AIC is an extension to the LSC language. The set of tools available with the LSC would enable us to create executable models which can be created and executed using the Play engine.

Currently, work is on to formalize the AIC language and semantics. The current challenges in front of us are providing the AIC with the ability to address the issues related to data sharing and mutual exclusion and, access control at joinpoints.

8. REFERENCES

- [1] AspectJ. <http://www.eclipse.org/aspectj>, 2002.
- [2] Remi Douence, Pascal Fradet, and Mario Sudholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*. Springer-Verlag, Lecture Notes in Computer Science 2487, 2002.
- [3] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, 2003.
- [4] David Harel and Rami Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and System Modeling*, 2(2):82–107, 2003.
- [5] Shmuel Katz and Joseph Gil. Aspects and superimpositions. In *ECOOP Workshops*, pages 308–309, 1999.
- [6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland., Springer-Verlag LNCS 1241, June 1997.
- [7] M. Monga, F. Beltagui, and L. Blair. Investigating feature interactions by exploiting aspect oriented programming, 2002.
- [8] Y. R. Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. pages 75–105, 2006.
- [9] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.
- [10] SpringAOP. <http://www.springframework.org>, 2007.