Visual Scenarios for addressing the Aspect Interference Problem*

Fernando Asteasuain and Víctor Braberman

Facultad de Ciencias Exactas y Naturales - Universidad de Buenos Aires
{fasteasuain,vbraber}@dc.uba.ar

Abstract. One of the most challenging problems in the aspect-oriented community is known as the aspect interference problem. This situation arises when the behavior to be introduced by two or more aspects is applied at the same point of interest. In order for the developer to resolve this conflict aspects' interaction and composition must be easy to analyze and manipulate. Under this context, we explore how FVS, a declarative visual language, can address this significant issue by providing a declarative and powerful pointcut model. In addition, FVS allows the possibility of reasoning about violating behavior, that represents a highly valuable information for the developer.

1 Introduction

In the last years, aspect orientation has emerged as an interesting approach to deal with complexity in software artifact description. Aspect oriented technology is rooted in the modularization of crosscutting concerns, which seems an interesting software engineering principle. First introduced as a technique applied in the codification phase, aspect orientation was quickly adapted to perform in other software development phases such as requirement engineering, modeling and design. In particular, applying aspect orientation in specifying requirements in early stages seems pretty natural [7], since requirements are normally expressed in such a way that fits an aspect profile (for example, "every time a message arrives, the server is notified"). This is, aspects manifests in requirements as behavior that is described as being triggered by many other behaviors [8]. Applying aspect oriented philosophy in early stages is widely known as "early aspects"-term first introduced by [1].

Nonetheless, some authors have pinpointed some difficulties with applying aspect orientation in the modeling phase, specially with operational notations inspired in finite state machines or labeled transition systems(e.g., statecharts) [13, 12]. Many aspect oriented approaches boil down into providing syntactic weaving mechanisms, usually with non-clear semantics counterpart [13]. Thus, unlike other well-established modularization mechanisms as procedures, parallel

* This work was partially funded by PAE-PICT-2007-02278:(PAE 37279), PIP 112-200801-00955 and UBACyT X021. V. Braberman is also affiliated to CONICET

composition, or logical conjunction (in declarative approaches) aspect orientation, though attractive in principle, is still a second class citizen, holding just the status of a hacking or dynamic instrumentation mechanism where semantics impact is not neatly characterized.

We believe one of the main reasons for this relies in that aspect oriented modeling techniques somehow inherited the implementation-oriented flavor aspects were first conceived for. In general, the behavior specified by an aspect must be introduced *exactly* before, or after the occurrence of another behavior. This is because pointcuts were originally designed to predicate and reason about methods calls abstractions. Actually, this result in a very rigid pointcut model for specifying requirements. This lack of flexibility causes a significant problem known as *aspect interference problem* [5], which arises when two or more aspects behavior interact with each other. Typically, this situation arises when two or more aspects behavior are to be introduced in the very same point of interest in our system. In these cases, several questions may arise:

– *Order of execution*: which one should be executed first? Why? Does it matter? This is also referred as defining aspects precedence. In some situations the desired final behavior of the augmented system may depend in the aspects' execution order. For example, consider two aspects to be added in a protocol communicating two systems. The first one encrypts the information following a security requirement whereas the second one prepares the data to be sent to fit in the communication channel. It may be the case that the data should be encrypted first and then fragmented to be sent into the communication channel. In this context, if the fragmentation aspects is executed first, then the encryption aspect may not behave as expected, perhaps even exposing the information to possible attackers. Therefore, is essential for the developer to completely understand aspects interaction in order to correctly determine aspects precedence.
– *Behavior Dependency*: Do the aspects depend with each other or the features to be added are orthogonal? In some cases, one aspect may depend on another aspect behavior. This dependency may due to sharing of variables, data, or by methods invocation. If the aspects just play the role of observers, in an read-only fashion, then there will be no dependency between them. This kind of aspects are called "spectative" [13]. However, if the aspects modify data, variable's values or alter somehow the execution flow, then there may exists a dependency between them that must be properly addressed. [13] classifies these aspects as: "regulative" and "invasive". Aspects in the former category can modify the base system as long they do not alter its original execution flow. On the other hand, invasive aspects can change the base system in any way, without restrictions. Thus in principle, they could completely invalidate any property that held previously in the underlying system [13]. For this reason we only focus in this work on regulative aspects.
– *Reasoning in the Augmented System*: Once all the aspects are weaved, what can be said about the augmented system' behavior? Where all the aspects

behavior introduced adequately? As is was said before, the lack of a clear semantics regarding aspects composition difficulties reasoning about the augmented behavior [13]. If several aspects are involved, then the final result may be troublesome to predict since complex aspects interactions can arise.

We think that in order to attack the aspect interference problem is crucial for the developer to deeply understand the implication of an aspect's application, thus the formalism used to specify aspects behavior plays a transcendental role. In addition, this problem is harder to handle in early stages, since there is no concrete implementation to inspect. In this sense, a declarative specification may greatly help since declarativeness seems an attractive and natural approach for capturing early requirements on behavior [19]. That is, the developer specifies *what* behavior is to be added, and not *how* this behavior is introduced, as in many operational notations, such as graphs-notations, finite-state machines, or state charts. Related to this point, aspect mining techniques ([18, 10, 16]) may help to identify aspects interactions and to point out where two or more aspects may have problems with their interaction. However, once the interaction problem is identified, the developer must still face the problem to determine and specify how these aspects must interact and the correct order of execution. In order to do so, is must be easy for the developer to look at aspects specification and easily define the correct precedence. As it was previously said, this is not a trivial task, since the developer may need to understand complex compositions, which sometimes are not clearly specified [13, 11, 15]. For example, in order to determine the correct precedence between aspects, the developer may have to deal with intricate automata, containing several states and transitions. Another interesting input in order to take this decision is to reason about **complementary behavior**. That is, to reason about how things can go wrong and violate the specified behavior, which represents valuable information to the developer. Again, this is bound to a complicate task in the mentioned operational notations. For example, operations to complement the language of an automaton are not trivial and may suffer from exponential state-explosion problems. Finally, another useful information for detecting conflicts between aspects interaction is the possibility of analyzing valid traces of the system. In this way, the developer may early detect these problematic conflicts.

In few words, we would like a declarative formalism to specify aspects that also enables the possibility of reasoning about violating behavior. Both are desirable characteristics that contribute to appropriately analyze and handle aspects interactions. Given this context, in this work we explore Featherweight Visual Scenarios (FVS) [3] as an aspect-oriented modeling language that correctly address the aspect interference problem. FVS, a simple fragment of VTS (Visual Timed Scenarios) [6], is a declarative visual language to define complex event-based requirements and to describe event patterns, which can be regarded as simple, graphical depictions of predicates over traces, constraining expected behavior. The formalism used is scenarios, where scenarios represent event patterns, graphically depicting conditions over traces. One distinguishable feature in FVS is that violating behavior can be automatically generated, enabling the possibility of reasoning about complementary behavior. In FVS each aspect is described as a rule following an antecedent-consequent shape establishing a new condition to be met by the system. FVS holds a very flexible pointcut model [4]. Conditions can be specified not only considering future behavior, but also considering past behavior, or even behavior occurring given a certain scope. In addition, valid traces can be generated for a particular set of FVS rules using a tableaux procedure described in [6]. In a nutshell, we analyze if FVS characteristics are capable enough to completely handle the aspect interference problem.

The rest of the paper is structured as follows. Section 2 introduces FVS features and details how it can be seen as an aspect oriented modeling language. Section 3 shows FVS in action addressing the aspect interference problem. Three different and simple examples are treated and analyzed. Finally, section 4 presents the conclusions of the present work and briefly discuss future work.

2 Featherweight Visual Scenarios

In this section we will informally describe the standing features of FVS. The reader is referred to [2, 4] for a more complete definition of the language including a formal characterization of its semantics. We use a simple running example based on the Telecom application, which is part of the AspectJ distribution. Basically, this application is a simple simulation of a telephony system in which customers make, accept, merge and hang-up both local and long distance calls. FVS is a graphical language based on scenarios consisting of points, which are labeled with the possible events occurring at that point, and arrows connecting them. Two kinds of relationship can be described among points: *precedence* and *forbidden* events. Some FVS scenarios are depicted in figure 1.

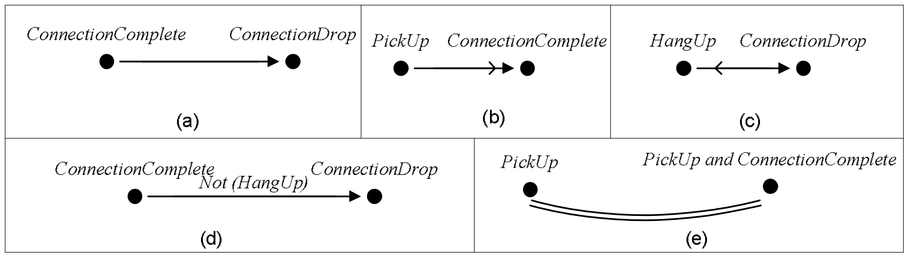

Fig. 1. FVS' Basic Elements

Scenario in figure 1-(a) indicates that a *ConnectionComplete* event precedes a *ConnectionDrop* event. A special notation is used to indicate the immediate next or previous occurrence of an event after another: a second (open) arrow near the destination point. This can be seen in figures 1-b and 1-c. Forbidden events are specified by labeling arrows. We see forbidden behavior in action in figure 1-d: a *ConnectionComplete* event precedes a *ConnectionDrop* event such that *HangpUp* event does not occur between them. To conclude, FVS supports aliasing between points, that can be used to represent that two or more events occurs simultaneously. For example, scenario in rule 1-e shows an occurrence of the *PickUp* event that occur simultaneously with a *ConnectionComplete* event.

FVS Rules We now introduce the concept of rules[1], a core concept in the language. In few words, a rule is divided into two parts: a scenario playing the role of an antecedent and, at least, one scenario playing the role of a consequent. The intuition is that wherever a trace "matches" a given antecedent scenario, then at least it must match one of the consequents. Graphically, the antecedent is shown in black, and consequents in grey. Since a rule can feature more than one consequent, elements which do not belong to the antecedent are numbered to identify the consequent they belong to.

To exemplify FVS rules, we model some requirements of the previously mentioned example. The rule in figure 2-a basically says that every established connection must eventually finish. More formally, it establishes that every *ConnectionComplete* event must be followed by a *ConnectionDrop* event. The rule in figure 2-b reasons consider two possible admitted behavior once the user picks up the phone. Either the connection is established (consequent 1), or there occurs an communication problem (consequent 2). Finally 2-c models conditions needed for a call to be made. If a *ConnectionComplete* event is followed by a *ConnectionDrop* event then it must be the case that the user hang up in between, and also, it picked up the phone before the connection was initiated.

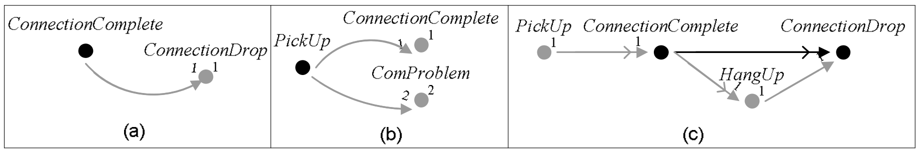

Fig. 2. FVS rules for the Telecom example

Anti-Scenarios An interesting feature in FVS is that anti-scenarios can be automatically generated from rule scenarios. This is a valuable information for the developer since they represent a sketch of how things could go wrong and violate the rule. The complete procedure is detailed in [6], but informally the algorithm computes all the possible situations where the antecedent is found, but none of the consequents is matchable. One anti-scenario for the rule in figure 2-c is shown in figure 3. In this case, the user does not hang up the phone before the connection is dropped.

[1] FVS rules corresponds to the Featherweight version of Conditional Scenarios available in VTS

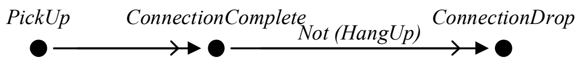

Fig. 3. An anti-scenario in FVS

Note that anti-scenarios are actually ordinary FVS scenarios as the ones shown in figure 1 and they are automatically generated from a given rule. Each of them represent a situation that violates the rule. As said, the anti-scenario in figure 3 is an scenario that violates rule in figure 2-c: the antecedent is matched (a *ConnectionComplete* event is followed by a *ConnectionDrop* event), but the consequent is not satisfied. Although a *PickUp* event occurs prior the occurrence of the *ConnectionComplete* event, there is no occurrence of an *HangUp* event, and both conditions are required to occur by rule 2-c. On the other hand, FVS rules are *composed* of scenarios: one scenario stands for the antecedent of the rule, and one or more scenarios stand for the consequents of the rule. We define scenarios representing a violation of the rule as anti-scenarios.

FVS as an Aspect-Oriented Modeling Language FVS rules fits into the aspect oriented perspective: rules' antecedents play the role of pointcuts, whereas consequents play the role of advices. To illustrate FVS expressivity power as an aspect oriented modeling language we will model the Timing aspect for the Telecom example. This aspect is in charge of computing the duration of the call. To achieve this objective, the aspect will start a timer when the connection is complete and the call is initiated. Similarly, it will stop the timer when the connection is dropped. Finally, it will compute the elapsed time according to the timer. This behavior is depicted in figure 4. Rules in figure 4-a and 4-b appropriately start and stop the timer respectively. Finally, rule in 4-c dictates that the *TimeElapsed* event will always follow the *StopTimer* event.

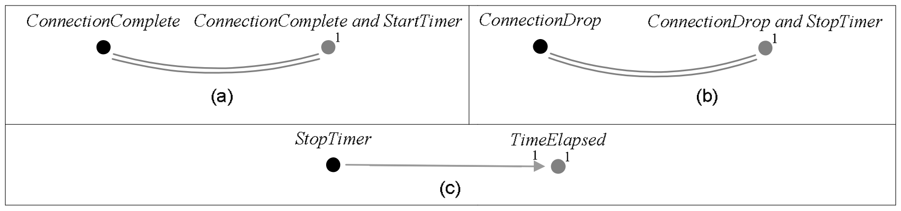

Fig. 4. Timing Aspect in FVS

3 Handling Aspects Interaction in FVS

We believe FVS supports desirable features to address the aspect interference problem. First, it supports a declarative specification that helps to naturally handle aspects interaction. Second, it generates automatically for every rule anti-scenarios, enabling the possibility of reasoning about violating behavior. Third, the developer may inspect system valid traces generated by a tableau process, in order to detect possible aspects interaction. That is, FVS allows localized and partial modeling. We explore these features in the next subsections. Subsection 3.1 consider an extra aspect for the Telecom Example, the Billing aspect, who is responsible of adding billing functionality to the Telecom application. On the other hand, subsection 3.2 deals with an aspects interaction problem through the use of anti-scenarios. In this case, we introduce another example, based on the JukeBox system presented in [10]. Subsection 3.3 is focused on localized modeling, describing the logging vs encryption conflict introduced in [9]. Finally, subsection 3.4 analyzes the mentioned examples.

3.1 Declarative Specifications and the Aspect Interference Problem

In this section we introduce rules for the Billing Aspect, who computes the charge of the call based on its duration. Based on the specification we obtain the rule shown in figure 5, which says that every occurrence of the *CostObtained* event must be preceded by a *TimeElapsed* event. In other words, it cannot be the case that the cost of the call is obtained without its duration being calculated before.

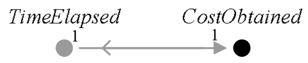

Fig. 5. Billing Timing aspect specification in FVS

As it can be noted, there exists a conflict between the Timing Aspect and the Billing Aspect. For the system to behave as expected, the timing aspect must precede the billing aspect. Otherwise, the billing aspect would not be able to fulfill its objective. However, by simply declaratively modeling these requirements as FVS rules the conflict was naturally solved as seen in the rule in figure 5. In both cases, an explicit precedence relationship is present between the *TimeElapsed* event and the *CostObtained* event.

The same situation in AspectJ or other textual languages is resolved by introducing the **declare precedence** statement, followed by a list containing the aspects' names, which is introduced manually by the developer. This is clearly an error-prone process. According to the AspectJ documentation, precedence is determined based on the aspects' order in the list. Thus, the first aspect in the list has the higher precedence, and the last one has the lowest precedence. What is more, due to AspectJ semantics, this precedence list behavior is not as intuitive as expected. For example, the Telecom example is resolved by adding the following line: ***declares precedence: Billing, Timing***. So, at first sight this might look as an error, because the clause is explicitly saying the Billing aspect is executed first than the Timing aspect. However, precedence is actually influenced by the execution flow. That is, for "before" advices the order behaves as expected: aspects at the beginning of the list are executed first. However, for "after" advices the precedence is completely reversed, because the execution flow is "returning" to the base system. Thus, aspects are executed from the last position to the first one. Since in the Telecom example both aspects contains "after" advices, aspects are executed in the right order after all.

A final remark can be made about flexility in the pointcut model. In the Telecom example, the cost of the call must be calculated exactly after the connection is dropped. However, it might be the case that is no actual requirement in the specification for the cost to be calculated at exactly that point. For example, it could be calculated at the end of the month instead. Despite this, due to the lack of flexibility in the pointcut model the developer is forced to use an "after" advice, leading to premature decisions. In FVS, more flexible options are available [4], or further rules can be added later on.

3.2 Anti-Scenarios and the Aspect Interference Problem

We now consider another example, based on the JukeBox system presented in [10]. Basically, the Jukebox system allows a user to select a song, that then will be reproduced by an appropriate multimedia player. Two aspects are added to consider other requirements for the system: the FavoriteList aspect, who is in charge of saving the last songs selected by the user, and the Counter aspect, who is in charge of counting the number of times that a song is played for statistical purposes. Since both aspects are applied before the song is played, their behaviors are in conflict. Once the conflict is discovered, is up to the developer to resolve it, by indicating the appropriate precedence of the aspects execution. A possible specification for this situation is depicted in figure 6.

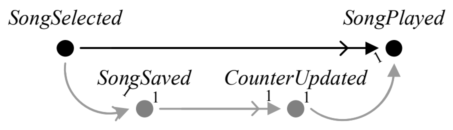

Fig. 6. JukeBox Aspects Conflict modeled in FVS

So as to resolve the conflict between these two aspects the developer may inspect the anti-scenarios automatically generated in FVS (see figure 7). FVS generates four anti-scenarios. Rule in figure 7-a models the case where a song is selected and played, but it is not saved nor the song counter is updated. Rules in figure 7-b and 7-c model two situations where only one of the two events of interest occurs: in the first case (figure 7-b), the song counter is updated, but the song is not saved, whereas in the second one (figure 7-b), the song is saved in the list, but its counter is not updated. Finally, in figure 7-d both events occur, but the song counter is first updated, and then the song is saved. These anti-scenarios represent valuable information for the developer in order to resolve the conflict, since the represent a human-readable description of how things could go wrong.

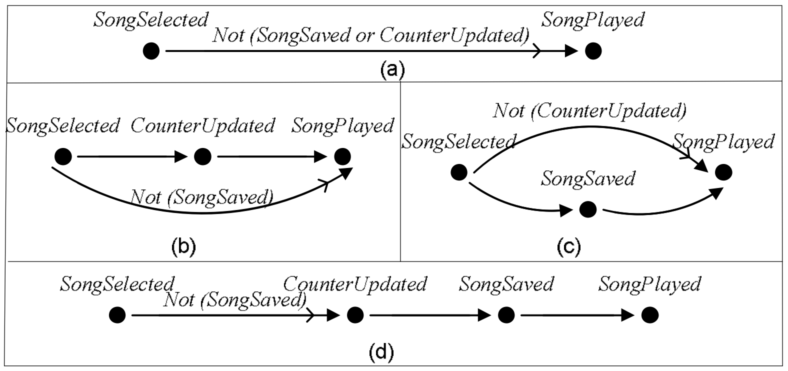

Fig. 7. Anti-scenarios for the JukeBox Conflict

By simply inspection the developer is able to analyze the interaction between conflicting aspects and reach a possible solution. In this example, by analyzing the anti-scenarios and in particular the fourth case, the developer may realize that in fact both events (the song is saved and the counter is updated) can occur in any order without affecting the system's behavior. That is, the requirements demand that the song must be updated and its counter updated, but actually who is executed first is unimportant. Taking this into consideration, the developer may reformulate the original aspects interaction in figure 6 as shown in figure 8.

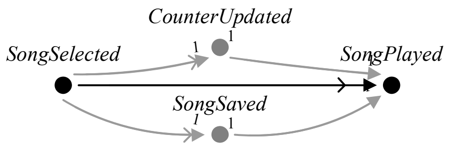

Fig. 8. Aspects interaction reformulated by analyzing anti-scenarios

3.3 Localized Modeling and the Aspect Interference Problem

Systems features in FVS can be independently inspected in FVS for early detection and analysis of aspects interaction. Consider the logging vs encryption conflict introduced in [9]. Once the server is ready, some information is sent to an interested client, but it must be logged and encrypted first for security reasons. This can be modeled in FVS as shown in figure 9-a. Before the data is sent, it is encrypted and logged, as specified in both rules.

Due to the tableaux process described on [6] valid traces can be generated upon a certain set of rules. The developer may inspect them in order to detect possible conflicts between aspects and gain more information about the specified system behavior. A valid trace satisfying both rules in figure 9-a can be given by the following succession of events: {*ServerReady, Encrypted, Logged, DataSent*}. In this case, information is encrypted first, and logged later. Analyzing this behavior, the developer may realize that this trace is actually an invalid one, since the data being logged is encrypted, and therefore useless for security analysis according to the specification. The possibility in FVS of analyzing localized partial execution of the system may lead to positively identify and resolve aspects interaction. In this case, the developer just need to add a new rule describing the proper interaction between both aspects: information is logged first, and then encrypted before being sent (see figure 9-b).

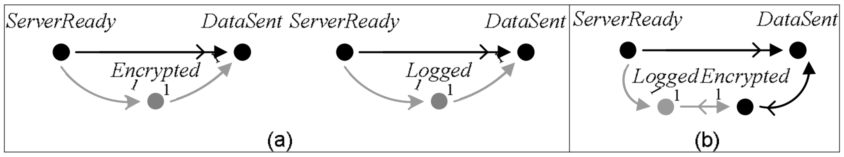

Fig. 9. Logging vs encryption conflict

3.4 Discussion

In the first example, the conflict between the Timing aspect and the Billing aspect is implicitly solved by the declarative specification as FVS rules. In these rules it is explicitly stated the precedence between both aspects, and this information can be obtained by simple looking at them. This information is not as easy achievable in operational notations. Usually precedence is stated by using UML stereotype-like constructors, and the developer is required to understand complex artifacts' composition in order to resolve the conflict.

In the second example, the developer uses violating behavior to understand and resolve the interference problem between both aspects augmenting the JukeBox system behavior. By a simple inspection of the anti-scenarios the developer gains valuable information about both aspects and is able to modify the behavior with the obtained information: the original specification was too restrictive: aspects can be executed in any order. In the third example, the developer analyzes valid traces of the system in order to detect and resolve a conflict between two aspects. FVS can generate valid traces, each one different and non equivalent with the rest. With this information, the developer can validate and express the desired expected behavior: the original specification was too permissive, and a new rule is added to restrict aspects behavior (logging aspects must precede the encryption aspect). These features are crucial in early stages, where no concrete implementation is available.

Related Work In previous works we explored FVS as an aspect oriented modeling language [2, 4], specially showing how the flexibility of our notation can be seen as a very powerful join point model. Continuing this research line, in this work we focus on illustrating how distinguishable FVS features can be applied to deal with the aspect interference problem from three different angles: declarativeness, violating behavior, and system traces analysis.

Other works has been proposed to tackle the aspect interference problem, from different points of view and focusing in different phases of software development [10, 18, 20, 14, 17]. Work like [14] or [18] proposes aspect-oriented tools for detecting conflict between interacting aspects. Both tools are based on syntactic transformation over graphs. As said, we propose a totaly different approach, moving towards a declarative language to model behavior, closer to early descriptions of the systems and the way requirements are expressed [19]. Other approaches like [10, 20] also offer a rich pointcut model based on events. Differently from our view, their use of event patterns (e.g., context free grammars) is basically limited to point-cut determination (while in our case patterns also indicates where "advices" may be featured).Our notion of events is abstract, we aim at a complete description of the systems in terms of rules and we sacrifice operationally of specification to enhance the declarative nature of our language. Finally, [17] presents an interesting tool called MEDIATOR to support conflicts among aspects. This framework is oriented to the implementation phase and it also supports the notion of triggered behavior. It is worth mentioning that, to our best knowledge, none of the previously mentioned approaches is equipped with deductive features for complementariness reasoning.

4 Conclusions and Future Work

In the present work we identify useful features that an aspect-oriented modeling language should support in order to tackle the aspect interference problem. Aspects specification should be easy to handle and understand so that their interaction can be straightforwardly depicted. This allows the possibility that any detected conflict between the involved aspects can be easily and naturally resolved. In this sense, reasoning about violating behavior represents a valuable information for the developer. Finally, the possibility of analyzing system valid traces for a particular set of requirements is also another meaningful information for the developer. By introducing some simple but rich examples, we showed how FVS fulfills these characteristics. Besides providing a declarative and flexible pointcut model, anti-scenarios can be automatically generated in FVS. Regarding future work, we are considering enhancing FVS's expressivity power to enable expressing arbitrary ω-regular languages. We are also working on defining a synthesis algorithm for FVS's rules, enabling the possibility of elaborated automatic analysis.

References

1. J. Araújo, A. Moreira, I. Brito, and A. Rashid. Aspect-oriented requirements with UML. In M. Kandé, O. Aldawud, G. Booch, and B. Harrison, editors, *Workshop on Aspect-Oriented Modeling with UML*, 2002.
2. F. Asteasuain and V. Braberman. Exploring Visual Scenarios as aspect oriented modeling language. In *ASSE 2010*. 39 JAIIO, 2010.
3. F. Asteasuain and V. Braberman. Specificattion patterns can be formal and also easy. In *SEKE*, 2010.
4. F. Asteasuain and V. Braberman. FVS: A declarative aspect oriented modeling language. *EJS - Electronic Journal SADIO*, 10(1):20–37, 2011.
5. L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Analysis of Aspect-Oriented Software (ECOOP 2003)*, 2003.
6. V. Braberman, N. Kicillof, and A. Olivero. A scenario-matching approach to the description and model checking of real-time properties. *IEEE TSE*, 31(12):1028–1041, 2005.
7. R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, A. Jackson, and S. Clarke. Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, AOSDEurope, 2005.
8. S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design. The Theme Approach*. Object Technology Series. Addison-Wesley, Boston, USA, 2005.
9. P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *ECOOP 2006*. Citeseer, 2005.
10. P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *2nd European Interactive Workshop on Aspects in Software (EIWAS05)*. Citeseer, 2005.
11. G. J. Holzmann. The logic of bugs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '02/FSE-10, pages 81–87, New York, NY, USA, 2002. ACM.
12. S. Katz. Diagnosis of harmful aspects using regression verification. In *FOAL*, pages 1–6, 2004.
13. S. Katz. Aspect categories and classes of temporal properties. In *Trans. Aspect-Oriented Softw. Develop*, pages 106–134, 2006.
14. K. Mehner, M. Monga, and G. Taentzer. Interaction analysis in aspect-oriented models. 2006.
15. D. O. Paun and M. Chechik. Events in linear-time properties. In *RE*, pages 123–132, USA, 1999. IEEE.
16. J. Pryor and C. Marcos. Solving conflicts in aspect-oriented applications. *Proceedings of the Fourth ASSE*, 32, 2003.
17. S. Sandra I. Casas, J. J. Baltasar García Perez-Schofield, and C. Claudia A. Marcos. MEDIATOR: an AOP Tool to Support Conflicts among Aspects. *International Journal of Software Engineering and Its Applications (IJSEIA)*, 3(3):33–44, 2009.
18. M. Storzer, R. Sterr, and F. Forster. Detecting precedence-related advice interference. In *ASE*, pages 317–322. IEEE, 2006.
19. A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. *re*, page 0249, 2001.
20. N. Weston, R. Chitchyan, and A. Rashid. Formal semantic conflict detection in aspect-oriented requirements. *Requirements engineering*, 14(4):247–268, 2009.