# Exploring visual scenarios as an aspect-oriented modeling language*

Fernando Asteasuain and Víctor Braberman

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

**Abstract.** Very well known problems such as the fragility problem, the AOP paradox, or the aspect interference problem threaten aspect oriented application in the modeling phase. In this work we explore FVS, a declarative visual language, as an aspect-oriented modeling language. Our language exhibits a very flexible and rich joinpoint model to leverage aspect-oriented application and is suitable for incremental modeling, a highly desirable quality attribute in any modeling language.

## 1 Introduction

In the last years, aspect orientation has emerged as an interesting approach to deal with complexity in software artifact descriptions. Aspect oriented technology is rooted in the modularization of crosscutting concerns which seems an interesting software engineering principle. Aspects are specified as a twofold: a pointcut, which selects **where** the aspect's behavior is to be introduced, and a advice, which details **what** behavior in particular is to be added. Moreover, its application in specifying requirements in early stages seems pretty natural [5], 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").

However, 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) [8, 7]. One of the main difficulties is the lack of flexibility in the joinpoint model. Expressing requirements which predicates about events that had previously happened are not easily (or not even feasible in some cases) modeled. For example, a requirement like "Every alarm is due to a fault", which predicate about past events, is not naturally captured in an aspect oriented specification. This lack of flexibility leads to very well known problems such as the AOP paradox [13] or the pointcut fragility problem [9]. Another significant problem is what the aspect-oriented community defined as the *aspect interference problem* [3], which arises when two or more aspects behavior interact with each other. For example, in the Telecom application which is part of the AspectJ distribution, the aspect who is in charge of keeping track of the duration of a phone call must precede the aspect in charge

of calculating the amount of money that customers are charged, since it needs to know the duration of the phone call. It is crucial for any in aspect-oriented modeling languages to correctly address this problem. A third obstacle is related with incremental modeling, a desirable characteristic for any modeling language. An incremental specification consists of gradually adding new features to a basic system. However, the lack of a clear semantics makes aspect oriented application cumbersome for incremental modeling, especially because reasoning about properties in the augmented system is hard to achieve [8].

Given this context, in this work we explore Featherweight Visual Scenarios (FVS) [2] as an aspect-oriented modeling language. FVS, a simple fragment of VTS (Visual Timed Scenarios) [4], 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. In FVS each aspect is described as a rule following an antecedent-consequent shape establishing a new condition to be met by the system. This is suitable for incremental modeling, since adding a new feature consists of simply adding a new rule to the set of rules to be fulfilled. Another strong point of FVS is due to its flexibility. Conditions can be specified not only considering future behavior, but also considering past behavior, or even behavior occurring given a certain scope. Finally, due to FVS expressivity power aspects interaction are introduced harmlessly. In few words, we propose a declarative language (not founded in modal logics but in scenario-based notations) to model early behavior where features can be incrementally added with an aspect oriented flavor, easing the specification of systems behavior even in early stages. The rest of the paper is structured as follows. Section 2 introduces FVS while section 3 show how it can be used as an aspect oriented modeling language. After some discussion in section 4 the paper concludes mentioning future work and conclusions.

## 2 Featherweight Visual Scenarios

In this section we will informally describe the standing features of FVS. The reader is deferred to [4] for a formal characterization of the language. We use a simple running example (based on the Lighting System presented in [11]) to highlight FVS features. It consists of an embedded software for a vehicle lighting system that controls the interior lights of an automobile. Basically, the system is in charge of turning on the interior lights when a door is opened as well as turning off the interior lights when all the doors are closed, based on the statuses of the doors, door locks and power switch.

FVS is a graphical language based on scenarios. Scenarios consist 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. An arrow between two points indicates precedence of the source with respect to the destination: for instance, in figure 1-(a)

*PowerOn*-event precedes *LightsOn* event. A common feature regarding precedence is reasoning the immediate next or previous occurrence of an event after another. For these cases we use an special representation: a second (open) arrow near the destination point. For example, in figure 1-b the scenario captures the very next *DoorClosed* event following a *DoorOpened* event, and not any other *DoorClosed* event. Conversely, to represent the previous occurrence of a (source) event, there is a symmetrical notation: an open arrow near the source extreme. In figure 1-c the scenario captures just the immediate previous *DoorOpened* event from *DoorClosed* event. The *forbidden* relationship is denoted labeling arrows. That is, events labeling the arrow are interpreted as forbidden events between both points. In figure 1-d *PowerOn* event precedes *DoorOpened* event such that *PowerOff* event does not occur between them.
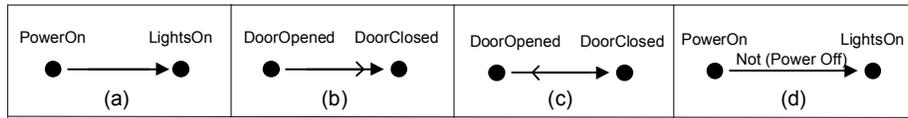


**Fig. 1.** Basic Elements in FVS

**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. In other words, rules take the form of an implication: an antecedent scenario and one or more consequents scenarios. The antecedent is a common substructure of all consequents enabling complex relationship between points in antecedent and consequents: our rules are not limited, like most triggered scenario notions, to feature antecedent as a pre-chart which events should precede consequent events. Thus, rules can state about expected behavior happening in the past or in the middle of a bunch of events. 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 for the previously mentioned example. The rule in figure 2-a basically says that lights must be turned on once the door is opened. More formally, it establishes that every *DoorOpened* event must be followed by a *LightsOn* event. The rule in figure 2-b reasons about past events. The requirement being modeled is: "The door must be unlocked to be opened". The rule specifies that if a *DoorOpened* event occurs, then a *DoorUnlocked* event must had previously occurred such that such that no *DoorLocked* event occurred between them. Finally, rule in figure 2-c specifies two

---

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

possible behaviors for turning lights off: either a door was closed (consequent 1) or the battery run out of energy (consequent 2). Note the power of our trigger notation where the antecedent need not to precede the consequent in time.
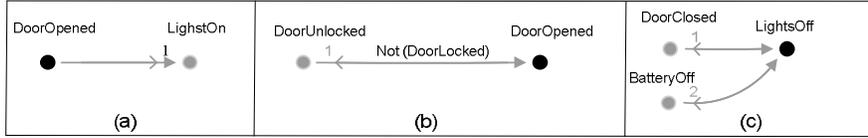


**Fig. 2.** Rule Scenarios in FVS

## 3 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 Interior Lights aspect for the running example, which basically dictates how interior lights must be turned on and off according to the door status. Rule in figure 2-a specifies when lights must be turned on. This functionality can be extended by specifying the opposite behavior: the lights must be turned off when doors are closed (figure 3-a). Yet another important rule can augment the expected behavior: interior lights can not be turned on twice in a row without being turned off in the middle (figure 3-b): between two consecutive occurrences of *LightsOn* event lights must be turned off. That is, once lights are turned on, they can not be turned on again without being turned off first. Note that in this rule, the consequent occurs between the two events representing the antecedent.
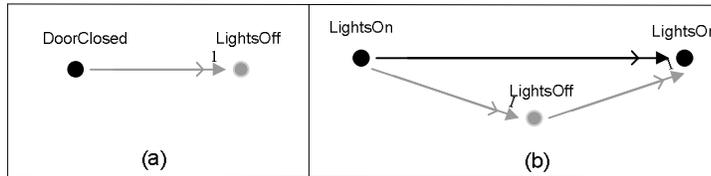


**Fig. 3.** Interior Lights Aspect in FVS

*Adding new features* Suppose now a new requirement arises, which include a battery saver feature that prevents the battery from being discharged. In the case where lights are turned off while the power is off, the battery saver is activated and after a certain amount of time automatically turn off the interiors lights. This new functionality is simply added as a new rule modeling the expected behavior. Rules in figure 4 models the Battery Saver aspect: figure 4-a models the battery saver activation and figure 4-b when lights are turned off.
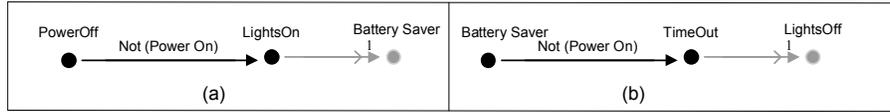
**Fig. 4.** Battery Saver Aspect in FVS

## 4 Discussion

The examples shown in the previous section allows an interesting discussion about FVS's performance as an aspect-oriented modeling language. Firstly, new features can be easily added, thus supporting incremental modeling: new features are simply added by introducing a new rule modeling their behavior. For example, the battery saver functionality was harmlessly introduced in the system. Secondly, FVS holds great flexibility to capture the particular moments of interest where aspects behavior needs to be inserted, resulting in a very rich and powerful pointcut model. Pointcuts can predicate about past behavior, or even behavior occurring in a certain scope. For example, the rule in figure 3-b models an aspect where the advice behavior occurs in between two points that constitutes the pointcut of the aspect (the initial *LightsOn* event and the final *LightsOn* event). This is very hard to achieve (if possible) in pointcut models that predicate on heap abstractions based on method calls. Similarly, rules that predicate about past events (e.g. figure 2-b) are modeled naturally in FVS. Again, this is difficult to achieve in traditional pointcut models. Lastly FVS handles aspects interactions in very neat way. For example, the Battery Saver Aspect needs the prior occurrence of the Interior Lights Aspect, whose is in charge of the *LightsOn* event. By simply modeling the battery saver functionality, aspects' interaction was naturally included in the model. In general, aspects precedence requires an special instruction to be explicitly included by the developer, or event worst, it is decided by the weaving process possibly leading to ambiguous specifications.

*Related Work* Approaches like [1, 11] take an operational view of aspect-oriented modeling weaving inspired on UML diagrams. 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 [10]. Our proposal focus on the notion of events, while most others works are grounded on notion of states or interactions. On the other hand, there are approaches that share the idea of matching (declarative) event patterns on traces [12, 6]. They pursue improving maintainability of applications heavily dealing with protocols. 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). Finally, FVS specifications were compared against other formal languages to express behavior in [2]. This comparison showed that, for the properties considered, FVS specifications were more succinct and easier to validate and modify.

# 5 Conclusions and Future Work

In this work we identify what we believe are important points to be considered in a aspect-oriented modeling language: the need for a rich and flexible joinpoint model, a neat characterization of aspects interaction and the ability of adding features incrementally. In this sense we explore FVS as an aspect-oriented modeling language and show how it fulfills the mentioned characteristics by modeling a simply but interesting example. Regarding future work, we are considering enhancing FVS's expressivity power to enable expressing arbitrary $\omega$-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. Araujo, J. W. J, and D. Kim. Modeling and composing scenario-based requirements with aspects. In *RE*, pages 58–67, 2004.
2. F. Asteasuain and V. Braberman. Specificattion patterns can be formal and also easy. In *SEKE*, 2010.
3. L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Analysis of Aspect-Oriented Software (ECOOP 2003)*, 2003.
4. 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.
5. 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.
6. R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD*, pages 141–150, 2004.
7. S. Katz. Diagnosis of harmful aspects using regression verification. In *FOAL*, pages 1–6, 2004.
8. S. Katz. Aspect categories and classes of temporal properties. In *Trans. Aspect-Oriented Softw. Develop*, pages 106–134, 2006.
9. C. Koppen and M. Storzer. Attacking the fragile pointcut problem. In *EIWAS*, 2004.
10. A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE*, 2001.
11. N. Noda and T. Kishi. An aspect-oriented modeling mechanism based on state diagrams. In *9th International Workshop on AOM*, 2006.
12. R. W. R. and K.Viggers. Implementing protocols via declarative event patterns. In *FSE*, pages 158–169, 2004.
13. T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. *SPLAT*, 2003.