Electronic Journal of SADIO

http://www.sadio.org.ar

vol. 10, no. 1, pp. 20-37 (2011)

FVS: A declarative aspect oriented modeling language

Fernando Asteasuain[1] *Víctor Braberman*[1,2]

[1] Departamento de Computación, FCEyN, Universidad de Buenos Aires, Ciudad Universitaria, Pab. I, (1428) BUENOS AIRES, Argentina.
fasteasuain@dc.uba.ar
[2] CONICET, Consejo Nacional de Investigaciones Científicas y Técnicas. Buenos Aires, Argentina.
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.

Keywords. Requirements Engineering, Aspect Oriented, Behavioral Modeling.

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 [8], 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 [9]. Applying aspect oriented philosophy in early stages is widely known as "early aspects"-term first introduced by [2].

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) [19, 18]. Many aspect oriented approaches boil down into providing syntactic weaving mechanisms, usually with non-clear semantics counterpart [19]. Thus, unlike other well-established modularization mechanisms as procedures, parallel 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.

One of the main difficulties is the lack of flexibility in the joinpoint model. Aspects were originally conceived for implementation and codification development phases, and not for the modeling phase. In general, pointcuts model predicate about method calls abstractions and advices are specified using Turing-complete languages. This implementation-oriented flavor of the aspect constructor make somehow awkward its application while capturing requirements.

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. Requirements that predicate about events happening given a certain scope suffers the very same problem, where very complex joinpoints may be needed to capture correctly the expected behavior. This lack of flexibility leads to very well known problems such as the AOP paradox [30] or the pointcut fragility problem [22].

Another significant problem is what the aspect-oriented community defined as the *aspect interference problem* [5], 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 [32]. The possibility of describing behavior in such way is more than attractive during early stages in software development, since usually requirements are not completely specified. Again, aspect-oriented philosophy seems to be a good candidate, since introducing a new aspect to a base system can be described as introducing a new feature or behavior when certain conditions are met. In this sense, an aspect can augment the systems behavior and at the same time restrict its application. For example, an aspect that encrypts information adds a new term to the system: the concept of encrypted information. But the aspect also dictates exactly when and how the information is encrypted and decrypted, (i.e., restricting the encrypting behavior).

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 [19]. Many techniques have been proposed to tackle this issues: introducing an intermediate layer [28, 20, 15], aspect-oriented interfaces [29, 21], or providing more powerful constructors [24, 12, 13, 16, 27]. However, most of these approaches focus only about augmenting the expressiveness of the pointcut model, neglecting the advice model. What is more, most of this efforts are focused on design and implementation phases, excluding the requirement phase.

1.1 Declarative modeling

Declarative modeling seems an attractive and natural approach for capturing early requirements on behavior [31]. In this respect, we believe there is a need for defining a new **declarative** language, capable of dealing with an **incremental** description of behavior. Taking into consideration declarative approaches, we believe that a graphical notation like scenarios will be more suitable than a temporal logic-based language, since the formal description and validation of properties involving logic formulas is a daunting task, even for trained people [11].

Given this context, in this work we introduce Featherweight Visual Scenarios (FVS) [4] as an aspect-oriented modeling language. 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.

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 expressive power aspects interaction are introduced harmlessly. For example, a possible rule can describe a requirement like the following: "If the user is admitted, then she must have previously entered correctly its password".

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 whose syntax and semantics are properly described next in section 3. Section 4 shows how FVS can be viewed as an aspect oriented modeling language. Next, section 5 analyzes the results of this view and describes how FVS addresses some very well known problems in the aspect oriented community. Finally, 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 referred to section 3 for a formal characterization of the language. We use a simple running example (based on the Lighting System presented in [25]) 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. Finally, two distinguished points are introduced to denote the beginning and the end of the trace: a big full circle for *begin*, and two concentric circles for *end*.

The former is illustrated in figure 1-e. This rule captures the first occurrence of the *PowerOn* event since the beginning of the trace. Analogously, rule in figure 1-f shows the use for the point representing the end of the trace. It captures the last occurrence of the *LightsOn* event in the trace.

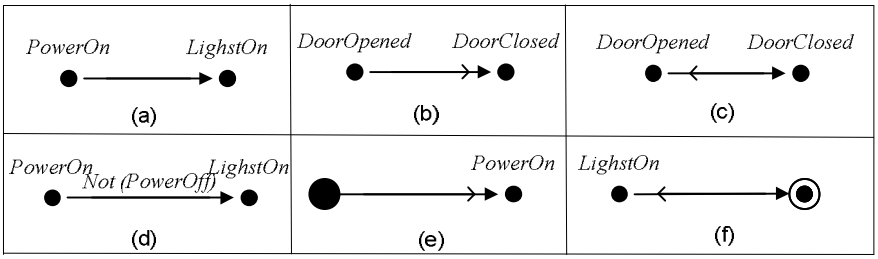

Figure 1: Basic Elements in FVS

2.1 FVS Rules

We now introduce the concept of rules[3], 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 of the previously mentioned example. The rule in figure 2 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 3 reasons about past events. The requirement being modeled is: "The door must be unlocked to be opened". In other words, the rule dictates that if a *DoorOpened* event occurs, then in the past the door was unlocked (the event *DoorUnlocked* occurred) and it remained in that state until it

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

Figure 2: An FVS rule describing interior lights behavior

was opened (the event *DoorLocked* event did not occur in between).

Figure 3: FVS rules describe the expected behavior of the system

Finally, rule in figure 4 specifies two 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.

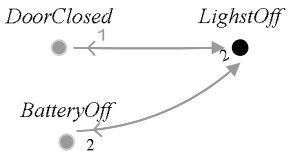

Figure 4: Two possible contexts for switching lights off

3 FVS Syntax and Semantics

We now formally define FVS syntax and semantics to provide a more complete definition of the language. The reader that is not interested in the formality of the language may skip this section and is referred to the next section (section 4): FVS as an Aspect-Oriented Modeling Language.

We introduce FVS syntax and semantics in the following way. First we introduce the formal definition of FVS scenarios. Second, we define a key operation between scenarios: *morphisms*, which allows the formal definitions of FVS rules. Finally, we define the formal semantics of FVS, by defining the notion of traces and rules satisfiability.

3.1 FVS Syntax

Definition 3.1 (FVS Scenario) *An FVS* scenario *is a tuple* $\langle \Sigma, P, \ell, \equiv, \not\equiv, <, \gamma \rangle$*, where:*
S1: Σ *is a finite set of propositional variables standing for types of events;*
S2: P *is a finite set of points;*
S3: $\ell : P \rightarrow \mathcal{PL}(\Sigma)$*, is a function that labels each point with a set of events, where* $\mathcal{PL}$ *is the set of propositional formulas that can be obtained from a variable(events) set* Σ*;*
S4: $\equiv \ \subseteq P \times P$ *is an equivalence relation (to "alias" points);*
S5: $\not\equiv \ \subseteq P \times P$ *is an asymmetric relation among points ("separation"of points);*
S7: $\gamma : (\not\equiv \cup <) \rightarrow \mathcal{PL}(\Sigma)$ *assigns to each pair of points, related by precedence or separation, a formula which constrains the set of events occurrences that may occur between the pair. Function* γ *satisfies the following condition.* $\gamma(\mathsf{p}, \mathsf{q}) \Rightarrow \gamma(\mathsf{p}, \mathsf{w}) \vee \ell(\mathsf{w}) \vee \gamma(\mathsf{w}, \mathsf{q}), \forall \mathsf{p} < \mathsf{w} < \mathsf{q} \in P.$

For a better comprehension of this section we provide further examples for the running example. Scenario in figure 5-a illustrates the occurrence of a simple sequence of events: once the car is started (*PowerOn* event) a door is opened and interior lights are turned on. Finally, a door is closed and interior lights are consequently turned off. Similarly, scenario in figure 5-b illustrates a situation where the car is started and switched off twice, and during this period, interior lights are never turned on.

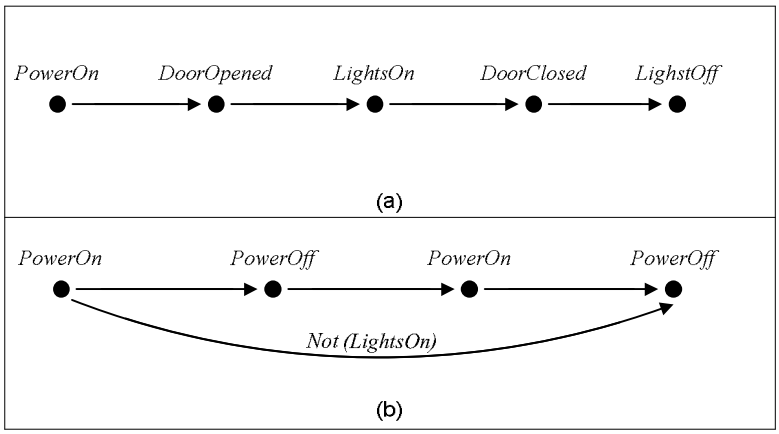

Figure 5: Further examples of FVS scenarios

We now formally define *morphisms* between scenarios. Intuitively, we would like to obtain a matching between scenarios ,i.e., a **mapping** between their points exhibiting how an scenario "specializes" another one.

Definition 3.2 (Morphism) *Given two scenarios $\mathcal{S}_1$, $\mathcal{S}_2$ (assuming a common universe of event propositions), and f a total function between P_1 and P_2 we say that f is a morphism from $\mathcal{S}_1$ to $\mathcal{S}_2$ (denoted $f : \mathcal{S}_1 \to \mathcal{S}_2$) iff*
M1: *$\ell_2(a) \Rightarrow \ell_1(\mathsf{p})$ is a tautology for all $\mathsf{p} \in P_1$ and all $a \in P_2$ such that $a \equiv_2 f(p)$;*
M2: *$\gamma_2(f(\mathsf{p}), f(\mathsf{q})) \Rightarrow \gamma_1(\mathsf{p}, \mathsf{q})$ is a tautology for all $\mathsf{p}, \mathsf{q} \in P_1$;*
M3: *$\mathsf{p} \equiv_1 \mathsf{q}$ then $f(\mathsf{p}) \equiv_2 f(\mathsf{q})$ for all $\mathsf{p}, \mathsf{q} \in P_1$;*
M4: *$\mathsf{p} \not\equiv_1 \mathsf{q}$ then $f(\mathsf{p}) \not\equiv_2 f(\mathsf{q})$ for all $\mathsf{p}, \mathsf{q} \in P_1$;*
M5: *$\mathsf{p} <_1 \mathsf{q}$ then $f(\mathsf{p}) <_2 f(\mathsf{q})$ for all $\mathsf{p}, \mathsf{q} \in P_1$.*

We say that $\mathcal{S}_2$ features more constraints than $\mathcal{S}_1$ when there exists a morphism $m : \mathcal{S}_1 \to \mathcal{S}_2$. This relation between two scenarios establishes that $\mathcal{S}_1$ is embedded into $\mathcal{S}_2$ if the latter features more constrains (this is analogous to a logical subsumption). Conversely, we say, in this case, that $\mathcal{S}_2$ specializes $\mathcal{S}_1$.

Figure 6 illustrates a morphism example (shown in dotted arrows). The scenario in the top of the figure (scenario $\mathcal{S}_2$) shows a sequence of events from the Lighting example considering events from the car power, and door and light status. On the other hand the scenario in the bottom of the figure (scenario $\mathcal{S}_1$) shows a more simple sequence of events: door are opened and closed without considering interior lights status. Thus, $\mathcal{S}_2$ features more constrains, since it considers the interior lights' status. In particular, considering the given morphism's definition is satisfied that $DoorOpened \wedge LightsOn \Rightarrow DoorOpened$ and that $DoorClosed \wedge LightsOff \Rightarrow DoorClosed$ are tautologies.

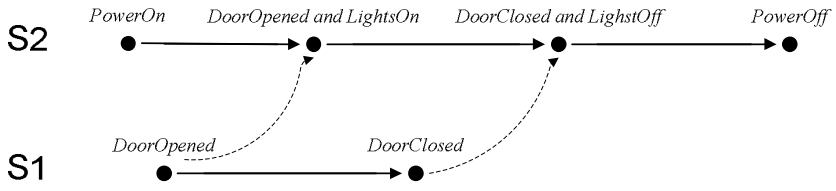

Figure 6: A morphism example

3.1.1 FVS rules

FVS rules model the expected behavior of the system, enabling a very rich, flexible and powerful mechanism to predicate and reason about systems' behavior. As it was said before, a rule structure is divided into two parts: a scenario playing the role of an antecedent and, at least, one scenario playing the role of a consequent. Whenever the antecedent is matched (i.e., a morphism f exists), then f should be extensible to show a matching of a consequent scenario (i.e. at least one of the consequents is matched too). The formal definition is given below.

Definition 3.3 (FVS Rule) *Given a scenario $\mathcal{S}_0$ (*antecedent*) and an indexed set of scenarios and morphisms from the antecedent $f_1 : \mathcal{S}_0 \to \mathcal{S}_1$,*

$f_2 : \mathcal{S}_0 \to \mathcal{S}_2$, ..., $f_k : \mathcal{S}_0 \to \mathcal{S}_k$ *(*consequents*), we call* $\mathrm{R} = \langle \mathcal{S}_0, \{f_i\}_{i=1...k} \rangle$ *an FVS Rule.*

As an example, consider the following rules in figure 7 modeling a portion of the expected behavior of the running example introduced before. The rule in figure 7-a says that once the car is started, the engine will be eventually switched off. Similarly, rule in figure 7-b basically says that an whenever a *PowerOff* event occurs then there must had occurred an event *PowerOn* in the past. That is, every *PowerOff* event must be preceded by a *PowerOn* event.

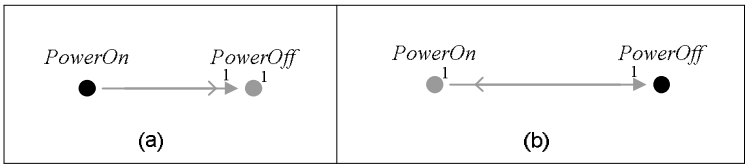

Figure 7: FVS rules examples

In the next section FVS semantics is fully described.

3.2 FVS Semantics

Semantics can be formalized using the notion of morphisms. The following definition establishes when a certain scenario $\mathcal{S}$ fulfills an FVS rule R:

Definition 3.4 (FVS Rules' Semantics) *An scenario* $\mathcal{S}$ *satisfies an FVS rule* R ($\mathcal{S} \models R$) *iff for every morphism* $m : \mathcal{S}_0 \to \mathcal{S}$ *there exists* $m_i : \mathcal{S}_i \to \mathcal{S}$, **for some** $i \in \{1..k\}$, *such that* $m = m_i \circ f_i$.

Two more definitions are needed to completely describe FVS semantics, which are described next. These definitions are focused in establishing the set of valid traces of a system. In order to do so, traces must be properly defined as well as their relationship with scenarios and rules satisfiability.

3.2.1 Trace-based Semantics

As said, traces model the abstract outcome of an event-based system. To keep our framework homogenous traces are understood as particular scenarios in the following way. Precedence in trace scenarios are total orders and ℓ function explicitly specifies the presence or absence of each possible event in each point of the trace, returning a *minterm* (a conjunctive clause where event propositions appears only once, either complemented or uncomplemented) over the set of available events.

Definition 3.5 (Traces) *A trace scenario* $\mathcal{S}_\sigma$ *is an scenario* $\langle \Sigma_\sigma, P_\sigma, \ell_\sigma, \equiv_\sigma, \not\equiv_\sigma, <_\sigma, \gamma_\sigma \rangle$ *where:*

T1: P_σ, $<_\sigma$ *is a total order;*
T2: $\ell_\sigma(\mathsf{p})$ *returns a minterm over* Σ_σ *for all* $\mathsf{p} \in P_\sigma$;
T3: $\gamma_\sigma(\mathsf{p},\mathsf{q})$ *= false if there is no* w *such that* $\mathsf{p} <_\sigma \mathsf{w} <_\sigma \mathsf{q}$*;*
T4: $\mathsf{p} \equiv_\sigma \mathsf{q}$ *if and only if* $\mathsf{p} = \mathsf{q}$ *,for all* $\mathsf{p},\mathsf{q} \in P_\sigma$.

Given this definition, we need a further operation to relate traces and scenarios saying when a general scenario can be projected into a trace. This notion is the *trace morphism*. In this way, we can later define when a trace satisfy a rule (or a set of rules).

Definition 3.6 (Trace morphism) *Given the trace scenario* $\mathcal{S}_\sigma$ *and an scenario* $\mathcal{S}$*, (assuming a common universe of event propositions and labels), and* g *a total function between* P *and* P_σ *we say that* g *is a projection morphism from* $\mathcal{S}$ *to* $\mathcal{S}_\sigma$ *(denoted* $g : \mathcal{S} \to \mathcal{S}_\sigma$*) iff*
M1: $\Sigma\sigma \subseteq \Sigma$
M2: $\ell_\sigma(g(\mathsf{p})) \Rightarrow \exists\ \upsilon_1,\upsilon_2\ \ldots \upsilon_n\ \ell(\mathsf{p})$ *is a tautology for all* $\mathsf{p} \in P$ *where* $\Sigma_{aux} = \{\upsilon_1,\upsilon_2\ \ldots \upsilon_n\}$
M3: $\gamma_\sigma(g(\mathsf{p}), g(\mathsf{q})) \Rightarrow \exists\ \upsilon_1,\upsilon_2\ \ldots \upsilon_n\ \gamma(\mathsf{p},\mathsf{q})$ *is a tautology for all* $\mathsf{p},\mathsf{q} \in P$ *where* $\Sigma_{aux} = \{\upsilon_1,\upsilon_2\ \ldots \upsilon_n\}$*;*
M4: $\mathsf{p} \equiv \mathsf{q}$ *then* $g(\mathsf{p}) = g(\mathsf{q})$ *for all* $\mathsf{p},\mathsf{q} \in P$*;*
M5: $\mathsf{p} \not\equiv \mathsf{q}$ *then* $g(\mathsf{p}) \not\equiv_\sigma g(\mathsf{q})$ *for all* $\mathsf{p},\mathsf{q} \in P$*;*
M6: $\mathsf{p} < \mathsf{q}$ *then* $g(\mathsf{p}) <_\sigma g(\mathsf{q})$ *for all* $\mathsf{p},\mathsf{q} \in P$.

This definition is very similar to the morphism operation previously defined, but introducing the necessary changes in the morphism's function requirements to properly deal with traces. Finally, the following definition provides the semantics of our language. The semantics of a set of rules R is the set of all traces that satisfy R. Formally:

Definition 3.7 (Trace-semantics of a FVS rule set) *A trace scenario* $\mathcal{S}_\sigma$*, satisfies a set of rules* R *iff there exists an scenario* S *such that:* $\forall\ r \in \mathrm{R}\ S \models r$ *and* $\exists\ g$*, a trace morphism* $g : \mathcal{S} \to \mathcal{S}_\sigma$.

In other words, a trace will satisfy a set of rules if there exists an scenario that can be projected into the trace and that satisfy all the rules in the set.

4 FVS as an Aspect-Oriented Modeling Language

As it was said before, aspects are described as a twofold: a pointcut, which selects **where** the aspect's behavior is to be introduced, and an advice, which details **what** behavior in particular is to be added. Requirements are commonly expressed in a aspect-oriented way, this is, expressing which things must happen in the system when a given situation arise, i.e., behavior that is triggered by other behaviors. For example, the following sentence illustrates a requirement for the Lighting system: "When any door is opened, interior lights must

be turned on". Using an aspect-oriented notation, this requirement may be modeled as something like this:

- **pointcut** (where): *Any Door is opened.*
- **advice** (what): *Turn on Interior Lights.*

FVS rules fits into the aspect oriented perspective: rules' antecedents play the role of pointcuts, whereas consequents play the role of advices. This mapping is shown in table 1 .

Table 1: Aspect oriented concepts translated into FVS concepts

Aspect-Orientation	FVS
Pointcut	Antecedent of a rule
Advice	Consequent(s) of a rule
Aspect	Rule

In order to illustrate this mapping, consider the previously introduced rule in figure 2, which naturally models this requirement fitting appropriately the aspect-oriented terminology. The antecedent is given by an occurrence of a *DoorOpened* event, that corresponds to the pointcut specification. Similarly, the consequent is given by the occurrence of the *LightsOn* event, which corresponds to the advice specification.

In what follows we complete show how FVS can model aspects considering the Lighting example.

4.1 Lighting system in FVS

The rule given in figure 2 specifies when lights must be turned on, which constitutes a crucial requirement of the system. This functionality can be extended by specifying the opposite behavior: the lights must be turned off when doors are closed (figure 8-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. This behavior is depicted in figure 8-b, which basically establishes that 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. This exemplifies the power and flexibility of FVS joinpoint model, where the behavior introduced in the advice actually occurs between the events constituting the pointcut of the aspect.

4.2 Incremental modeling: 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

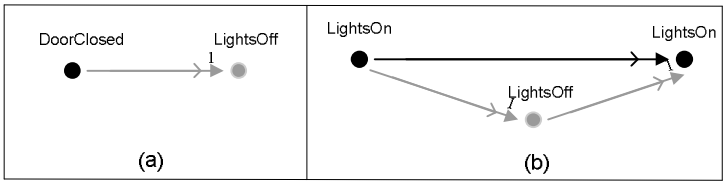

Figure 8: Interior Lights Aspect in FVS

turned on while the power is off, the battery saver is activated and after a certain amount of time it automatically turns off the interiors lights. This new functionality is simply added as new rules modeling the expected behavior.

Rules in figure 9 and 10 model the Battery Saver aspect. For one side, figure 9 models the battery saver activation: given the occurrence of the *PowerOn* event followed by an *LightsOn* event and no occurrence of *PowerOff* event in between (i.e, the lights are turned on while the power is off), then *BatterSaverOn* event must occur, representing the activation of the batter saver.

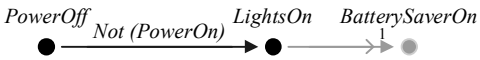

Figure 9: Battery Saver Aspect in FVS-Battery Saver Activation

On the other side, figure 10 shows the battery saver in action once activated: after a certain amount of time, which is given by the occurrence of the *TimeOut* event, interior lights must be turned off. That is, given the occurrence of the *BatterySaverOn* event followed by an *TimeOut* event such that there is no occurrence of the *PowerOff* event in between then *LightsOff* event must also occur. In other words, interior lights had been turned on while the power was off too long and they must be turned off to prevent the battery from being discharged.

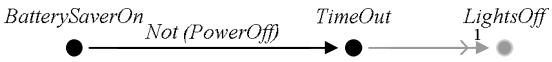

Figure 10: Battery Saver behavior in action

5 Discussion

The examples shown in the previous section allows an interesting discussion about FVS's performance as an aspect-oriented modeling language. As it was previously stated, we mentioned three obstacles threatening aspect-oriented application in early stages: the lack of flexibility in the joinpoint model, *aspect interference problem*, and the difficulty for an incremental modeling approach. We now discuss and analyze how FVS might address these known problems.

5.1 JoinPoint Model Flexibility

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 8-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. figures 3 and 4) are modeled naturally in FVS. More specifically, in figure 3 the behavior denoted in the consequent of the rule (*DoorUnlocked* event) occurs prior to the behavior denoted in the antecedent (*DoorOpened* event). Analogously, both consequents in rule 4 precede the antecedent in time. Again, this is difficult to achieve in traditional pointcut models.

This enhanced flexibility eases aspects application in early stages, since requirements that predicate about past events, or events occurring in a certain scope can be directly and naturally modeled. In more rigid join point models, these requirements might have to be rephrased, or they result in very complex pointcuts, which are hard to understand and evolve.

5.2 Aspect Interference

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.

In addition, the graphical nature of our language plays an important role in handling aspect interference. The behavior of each aspect is graphically modeled, which helps the specifier in detecting and handling aspects' interactions.

5.3 Incremental Modeling

As it was said before, incremental modeling is a highly desirable feature for any modeling language. In FVS, 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's specification. In the same way, new features can be gradually added to an existing specification (set of rules). Thus, the declarative flavor of the FVS language leverages aspects' application in early stages by supporting incremental modeling.

6 Related Work

Regarded as an aspect oriented modeling approach, our notation is based on a symmetric view of aspect orientation [14]. In a symmetric view aspects behavior is considered and treated as any other functionality in the system. On the contrary, in an asymmetric view exists a separation between aspects and other functionality. There exists a "basic" system, usually denoting functional requirements, where aspects behavior is aggregated. The obtained system after the weaving process, combining basic functionality plus aspects behavior is called the "augmented" system.

Approaches like [25, 7, 23, 26, 17] take an asymmetric view of aspect oriented modeling weaving aspects into notation like UML sequence diagram, message sequence charts, etc. Other like [3] are symmetric but also aspect application is operationally defined via a weaving mechanism. 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 [31]. 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 asymmetric aspect oriented programming approaches that share the idea of matching (declarative) event patterns on traces [33, 10, 1]. 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). 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, as a behavioral modeling language, FVS was compared against other formal languages in [4]. In particular, FVS specifications were compared against automata-based notations, natural language-based notations, and temporal logic descriptions taking as a case of study the specification patterns proposed by [11]. This comparison showed that, for the properties considered, FVS specifications were more succinct and easier to validate and modify.

7 Conclusions and Future Work

In this work identify several obstacles that somehow prevents aspect-oriented orientation from being a solid and attractive candidate to model behavior in early stages. These obstacles causes several known problems in the aspect oriented community such as the AOP paradox [30], the pointcut fragility problem [22], and the *aspect interference problem* [5].

We believe that in order to address these obstacles an aspect oriented modeling language should hold the following characteristics: a rich, expressive and flexible joinpoint model, a neat characterization of aspects interaction and the ability of adding features incrementally. In this sense we explored FVS as an aspect-oriented modeling language and showed how a simple but yet interesting example was modeled in FVS in an aspect-oriented way. Finally, we presented a discussion to point out how FVS fulfills the mentioned characteristics.

7.1 Future Work

Regarding future work, we would like to continue exploring FVS as an aspect oriented modeling language. This involves using FVS in more complex examples and cases of study such as protocols specifications and expressing communication between processes. In addition, we believe that FVS can greatly help to detect and resolve situations where two or more aspects behavior are in conflict, since aspects behavior is graphically denoted. Thus, our next immediate step is to study this interesting possibility.

We are also considering enhancing FVS's expressive power to enable expressing arbitrary ω-regular languages. Finally, we are working on defining a synthesis algorithm for FVS's rules, enabling the possibility of elaborated automatic analysis.

8 Acknowledgments

This work was partially funded by PAE-PICT-2007-02278:(PAE 37279), PIP 112-200801-00955, UBACyT X021 and STIC-AmSud project TAPIOCA.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. D. Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA 05*, pages 345–364, 2005.

[2] 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.

[3] J. Araujo, J. Whittle, and D. Kim. Modeling and composing scenario-based requirements with aspects. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 58–67. IEEE, 2005.

[4] F. Asteasuain and V. Braberman. Specificattion patterns can be formal and also easy. In *The 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2010.

[5] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Analysis of Aspect-Oriented Software - European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[6] V. Braberman, N. Kicillof, and A. Olivero. A scenario-matching approach to the description and model checking of real-time properties. *IEEE Transactions on software Engineering*, pages 1028–1041, 2005.

[7] W. Cazzola and S.Pini. Join point patterns: A high-level join point selection mechanism. In *MoDELS Workshops*, 2006.

[8] 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.

[9] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design. The Theme Approach.* Object Technology Series. Addison-Wesley, Boston, USA, 2005.

[10] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM, 2004.

[11] M. Dwyer, M. Avrunin, and M. Corbett. Patterns in property specifications for finite-state verification. In *21st international conference on Software engineering (ICSE)*, pages 411–420, 1999.

[12] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. *Programming Languages and Systems*, pages 366–381, 2004.

[13] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM, 2003.

[14] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. In *Technical report, IBM - RC22685 (W0212-147)*, December 2002.

[15] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264, 2009.

[16] K. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *Proceedings of the 5th international Symposium on Principles and Practice of Programming in Java*, pages 63–72. ACM, 2007.

[17] L. H. J. Klien and J. Jezequel. Semantic-based weaving of scenarios. In *AOSD*, 2005.

[18] S. Katz. Diagnosis of harmful aspects using regression verification. In *Foundations of Aspect-Oriented Languages (FOAL) Workshop*, pages 1–6, 2004.

[19] S. Katz. Aspect categories and classes of temporal properties. *Transactions on aspect-oriented software development I*, pages 106–134, 2006.

[20] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. *ECOOP 2006–Object-Oriented Programming*, pages 501–525, 2006.

[21] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. *ECOOP 2005-Object-Oriented Programming*, pages 195–213, 2005.

[22] C. Koppen and M. Storzer. PCDiff: Attacking the fragile pointcut problem. In *First European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[23] M. Mahoney, A. Bader, O. Aldawud, and T. Elrad. Using aspects to abstract and modularize statechars. In *AOM Workshop In Conjunction with UML '04*, 2004.

[24] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. *Programming Languages and Systems*, pages 105–121, 2003.

[25] N. Noda and T. Kishi. An aspect-oriented modeling mechanism based on state diagrams. In *9th International Workshop on Aspect Oriented Modeling (AOM)*, 2006.

[26] A. B. O. Aldawud and T. Elrad. Weaving with statecharts. In *Workshop on Aspect-Oriented Modeling (held with AOSD-2002)*, 2002.

[27] H. Rajan and G. Leavens. Ptolemy: A language with quantified, typed events. *ECOOP 2008–Object-Oriented Programming*, pages 155–179, 2008.

[28] A. C. Rubn Altman and N. Kicillof. On the need for SetPoints. In *First European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[29] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with xpis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009.

[30] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. *SPLAT*, 2003.

[31] A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2002.

[32] M. Veanes and W. Schulte. Protocol Modeling with Model Program Composition. *LECTURE NOTES IN COMPUTER SCIENCE*, 5048:324, 2008.

[33] R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 159–169. ACM, 2004.