Specification patterns can be formal and still easy

Fernando Asteasuain
FCEyN-University of Buenos Aires
Email: fasteasuain@dc.uba.ar

Víctor Braberman
FCEyN-University of Buenos Aires
Email: vbraber@dc.uba.ar

***Abstract*—Property specification is still one of the most challenging tasks for transference of software verification technology like model checking. The use of patterns has been proposed in order to hide the complicated handling of formal languages from the developer. However, this goal is not entirely satisfied. When validating the pattern the developer may have to deal with the pattern expressed in some particular formalism. For this reason, we identify three desirable quality attributes for the underlying specification language: succinctness, ease of validation and modifiability. We show that typical formalisms such as temporal logics or automata fail at some extent to support these features. In this work we propose FVS, a graphical scenario-based language, as a possible alternative to specify behavioral properties. We illustrate FVS' features by describing one of the most commonly used pattern, the Response Pattern, and several variants of it. Other known patterns such as the Precedence pattern and the Constrained Chain pattern are also discussed. We also thoroughly compare FVS against other used approaches.**

I. Introduction

Property specification is still one of the most challenging tasks for transference of software verification technology like model checking. Users of these techniques must still face the challenge of expressing properties in the language or formalism used in the specification tool. Using natural language to express requirements may appear as an alternative to formal approaches, but in general leads to ambiguous and imprecise specifications, threatening the advantages of an automated verification process. Two of the most common formal approaches are temporal logics such as Linear Temporal Logic (LTL) and operational notations such as finite-state machines. In these approaches final users are required to have a considerable expertise on the formalism to accurately express the requirement they want to express ([10], [12], [9], [21], [6], [18], [2]) and this clearly constitutes not a minor obstacle.

The usage of specification patterns has been proposed as an interesting alternative to overcome this problem [9], [8]. The main purpose of a pattern is to capture recurring solutions to a particular type of problem. In [9], patterns are described considering two aspects. On the one hand, the *intent* of the pattern is described, that is, the structure of the defined behavior. This is usually expressed using a disciplined natural language (DNL) notation [21]. For example, the intent for the *Response* pattern is denoted as "One or more occurrences of **action** result in one or more occurrences of **response**". On the other hand, each pattern has a scope, which is the extend of the program execution over which the pattern must hold. For example, we can say that the pattern must hold between the occurrence of two given events. Although patterns offer a friendlier way to express typical requirements, in order to determine if the pattern is accurately expressing the desired property, the developer usually must deal with not only the DNL description of the pattern, but its translation into an underlying formalism such as LTL [12], [21]. This situation also arises when deciding the right pattern to be employed, or even more important, when validating the usage of the pattern. Therefore, using patterns is not enough to entirely hide the subtleties of the underlying formalism from the user. This suggests that the specification language should be easy to use, and sufficiently expressive to enable skilled and non-skilled users to use it appropriately [16], [12]. More specifically, we define three quality attributes desirable for the formalism: *succinctness*, *ease of validation* and *modifiability*. The first one requires that the specification of a pattern expressed in the formalism should be as succinct as the DNL description of the pattern. Ease of validation estimates how simple it is to determine if the specification of a pattern actually expresses the desired property. This attribute can be further subdivided into two sub-attributes: *comparability* and *complementariness*. The resulting specification of the patterns should be easy to compare and distinguish. For example, when comparing two patterns it is natural to try to understand which one is more restrictive. Complementariness refers to the ability of reasoning about how the property is violated, which provides meaningful information to the developer. The formalism should provide an easy way to reason about complementary behavior. Finally, modifiability is the ability to manipulate objects expressed in the formalism, so that the pattern can be adapted to subtle modifications in the application context.

We believe that the provided mappings of the patterns to LTL detailed in [9], [8] fail at some extent to support these features. The resulting LTL formulae may result in complicated artifacts, which are difficult to compare without deductive manipulation. Reasoning about complementary behavior is also troublesome since it requires sophisticated formulae manipulation. If the underlying formalism is an automata-based one instead, the analysis is similar. In order to accurately compare two automata language inclusion needs to be tested. Similarly, operations to complement the language of an automaton are not trivial and may suffer from exponential state-explosion problems. Modifiability also follows an analogous reasoning. Intricate operations may be needed to modify a LTL formula or an automaton to adapt to different situations.

Given this context we propose a graphical language based

on scenarios which aims to improve and ease the property specification process. The language, called FVS (Featherweight Visual Scenarios) is a simple fragment of VTS (Visual Timed Scenarios) [5], a visual language to define complex event-based requirements. We show that FVS, although simple, is powerful enough to properly describe specification patterns. For one side, its visual aspect will help the user concentrate in the properties themselves instead of dealing with the subtleties of their formalization. The scenarios are built using just a few and simple elements, therefore obtaining compact and minimal specifications. Semantic and logical relationships between the elements involved can be extracted directly from the scenarios, improving reasoning about patterns. The language supports the automatic construction of anti-scenarios, which help the user in the property specification by looking at behavior that leads to the violation of the property. What's more, patterns specification is very flexible and can be easily adapted to different application contexts. Concretely, we analyze FVS as a specification language by describing some of the patterns introduced by [9] with an special focus on one of the most used pattern, namely the Response Pattern, and several variants of it. The Constrained Chain Pattern and the Precedence pattern are also modeled in FVS. We compare FVS against the original patterns specification in [9] and also against the Propel language [21], an approach that define one of the most known extensions to patterns specification. Propel uses two notations: an extended finite-state automaton representation and a DNL representation, based on a restricted subset of natural language. The comparison is based on the previously defined concepts: succinctness, ease of validation and modifiability.

The rest of the paper is structured as follows. After describing some initial background, FVS is explained in section II. Section III shows the specification of the Response Pattern (including several variants of it), the Constrained Chain pattern and the Precedence pattern in FVS. Analysis and comparison with other formalisms are presented in section IV. The paper concludes mentioning future work and conclusions.

A. Background

Using patterns for property specification was first introduced by [9]. According to the authors, patterns fall into two main categories based on their semantics: *Occurrence*, where patterns require events to occur or not to occur and *Order*, where patterns constrain the order of events. The Absence pattern and the Universality pattern are examples of the first category, whereas Precedence and Response are examples of the second category. The complete list of the patterns is available online [8] where also mappings to different formalisms are supplied. In this work we will focus only on their mapping to LTL, being one of the most used formalisms for specification. The LTL formulae discussed in this work refer to the LTL mapping specified in [8]. The other formalism discussed here is the Propel Language [21]. Propel presents a very exhaustive extension to some of these patterns, which covers additional issues regarding patterns behavior. FVS is thoroughly compared to both approaches.

II. Feather weight Visual Scenarios

In this section we will informally describe the standing features of FVS. The reader is referred to [5] for a formal characterization of the language. 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. An arrow between two points indicates precedence of the source with respect to the destination: for instance, in figure 1-(a) *A*-event precedes *B*-event. We use an abbreviation for a frequent sub-pattern: a certain point represents the next occurrence of an event after another. The abbreviation is a second (open) arrow near the destination point. For example, in figure 1-b the scenario captures the very next *B*-event following an *A*-event, and not any other *B*-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 immediately previous *A*-event from *B*-event. Events labeling the arrow are interpreted as forbidden events between both points. In figure 1-d *A*-event precedes *B*-event such that C-event does not occur between them. The abbreviations presented before can also be expressed using labeled arrows. Scenario in figure 1-e shows an equivalent version of the scenario in figure 1-b. 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* (shown in figure 1-f).

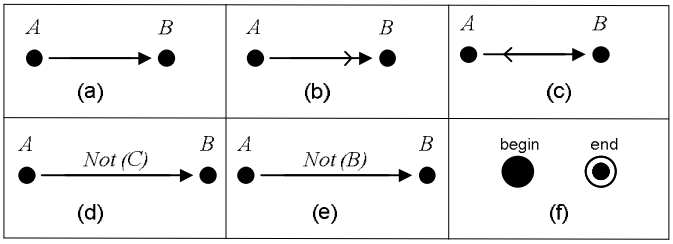

Fig. 1. Basic Elements in FVS

A. FVS Rules

We now introduce the concept of Rule Scenario (RS) or simply a rule[1], a core concept in the language. Roughly speaking, 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 whenever a trace "matches" a given antecedent scenario, then it must match at least one of the consequents. In other words, rules take the form of an implication: an antecedent scenario and one or more consequent scenarios. The antecedent is a common substructure of all consequents, enabling complex relationship between points in antecedent and consequents: our rules are not limited,

[1]Rule Scenarios represent the FVS's version of Conditional Scenarios available in VTS [5]

like most triggered scenario notions, to feature antecedent as a pre-chart where events should precede consequent events. Thus, rules can state 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 scenario are numbered to identify the consequent they belong to. An example is shown in figure 2. The interpretation of the rule is that, whenever an *Access request* event is followed by an *Access granted* event (without a logoff in between), one of two other event sequences must be observed too. One of them (consequent1) requires that, after the access has been requested, a valid password is entered. The other one (consequent 2) allows for the case where a valid password has been entered before access to the resource is requested. Observe the power of our trigger notation, where the antecedent need not precede the consequents in time.

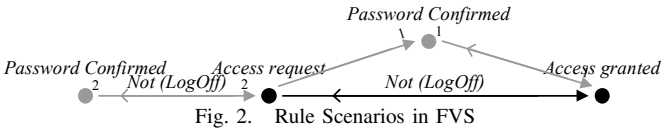

Fig. 2. Rule Scenarios in FVS

B. Anti-Scenarios

An interesting feature in FVS is that anti-scenarios can be automatically generated from rule scenarios. This is valuable information for the developer since it represents a sketch of how things could go wrong and violate the rule. The complete procedure is detailed in [5], but informally the algorithm computes all possible situations where the antecedent is found, but none of the consequents is matchable. One anti-scenario for the rule in figure 2 is shown in figure 10. In this case, a valid password was not entered since the beginning of the trace and still access is granted.

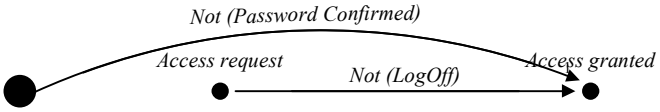

Fig. 3. An anti-scenario in FVS

III. Pattern Specification in FVS

In this section we illustrate how FVS may describe some of the specification patterns introduced by [9]: the Response Pattern, the Constrained Chain pattern and the Precedence pattern. One of the most widely used patterns is the *Response* pattern, in which the occurrence of one event (referred as the *Action* event), leads to an occurrence of another event, the *Response* event. This illustrates a cause-and-effect relationship between the events involved. Examples of the Response pattern may be requirements such as "Every client request is acknowledged by the server" or "every time the doors are opened the lights are turned on". The rule depicted in figure 4 reflects

Fig. 4. The basic Response Pattern

this behavior. As pointed out in [21], a closer examination of this pattern's behavior reveals there are significant questions about the pattern that need to be answered and may lead to variants of it. For example, can the *Action* event occur more than once before the *Response* event occurs? Taking this fact into consideration, [21] proposes an extension to the patterns' specification, defining six possible options to obtain a more accurate and complete definition of the pattern: **Pre-arity**, which determines whether the *Action* event may occur one time or many times before the *Response* event does, **Post-arity**, which determines whether the *Response* event may occur one time or many times after the *Action* event does, **Immediacy**, which determines whether or not other intervening events may occur between the *Action* event and the *Response* event, **Precedence**, which determines whether or not the *Response* event is allowed to occur before the first occurrence of the *Action* event, **Nullity**, which determines whether or not the *Action* event must ever occur and finally, **Repeatability**, which determines whether or not occurrences of the *Action* event after an occurrence of a *Response* event are required to be followed by a *Response* event. In what follows we model all six cases in FVS.

1) Pre-arity: the rule in figure 4 models the situation where the *Action* event can occur several times before the *Response* event (by default in FVS, any arbitrary *Action* event may become a valid antecedent to trigger the rule). To limit the behavior such that *Action* event can not be repeated before *Response* event, a restriction is added to the rule. The condition prohibits the occurrence of another *Action* event until the occurrence of the *Response* event. This is shown in figure 5-a.

2) Post-arity: the rule depicted in figure 4 also models the situation where the *Response* event can be repeated. If *Response* event cannot be repeated, then between any two consecutive *Response* events there must occur an *Action* event. In this way, traces containing two or more consecutive *Response* events after an *Action* event will be excluded. Figure 5-b illustrates this behavior.

3) Immediacy: forbidden events between *Action* and *Response* are simply added to the pattern by including the forbidden events properly. This is illustrated in figure 5-c.

4) Precedence: this is achieved by requiring that the occurrence of a *Response* event must always be preceded by an *Action* event. The rule in figure 5-d reflects this behavior.

5) Nullity: the initial scenario for the Response pattern (figure 4) covers the case where the *Action* event is not required to occur. Demanding the occurrence of the *Action* event implies that the *Action* event must occur after the beginning of the trace. This is shown in figure 5-e.

6) Repeatability: the initial scenario models a repeatable behavior, since it requires a *Response* event for every *Action*

event. To model the case where the pattern must be fulfilled only for the first occurrence of the *Action* event, a restriction is added so that the antecedent contemplates only the first, and not each, occurrence of the *Action* event. The antecedent in figure 5-f demands that there is no previous occurrence of the *Action* event from the beginning of the trace. Clearly, only the first *Action* occurrence will satisfy this condition.

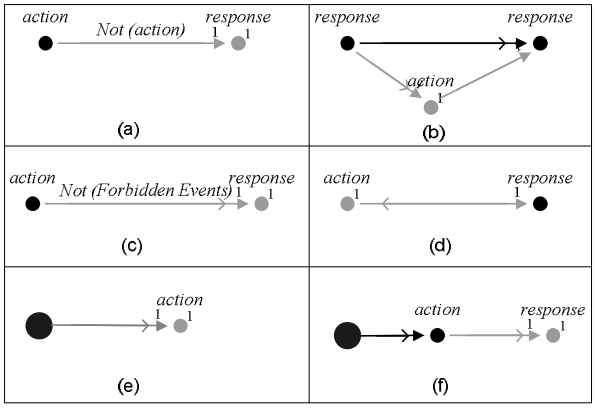

Fig. 5. Six rules covering the extended behavior

The rules presented in figure 5 model each aspect individually. Naturally, two or more aspects can be combined by simply including their corresponding rule.

A. Modeling Scopes

The previous section describes what the work in [9] defines as the intent of the pattern, that is, the structure of the defined behavior. This notion can be completed by defining a scope for the pattern, establishing the extent of program execution over which the pattern must hold. The available scopes defined in [9] are: **Before P** (the pattern must hold before the first occurrence of an event *P*), **After Q** (the pattern must hold only after the first occurrence of an event *Q*), **Between P and Q** (the pattern must hold after the occurrence of *P* but before the occurrence of *Q*), **After Q until P** (similar to the previous one, but *P* is not required to occur) or **Global**, which denotes the whole execution.

Scopes are introduced in FVS in the same way as any other restriction of behavior. Thus, Global scope is implicity modeled by introducing no scope restrictions (all the rules shown up to here assumed Global scope). In what follows we model the rest of the scopes for the basic Response pattern[2].

1) Before P - After Q: The rule in figure 6-a corresponds to the Before P scope. The occurrence of the property must precede the first occurrence of *P*. This is, if an Action event occurs before the first *P* occurrence, then a *Response* event must also occur before *P*. Similarly, the scenario in figure 6-b shows the After Q scope. That is, if an *Action* event occur after the first *Q* occurrence, then a *Response* event must also occur afterwards.

[2]Note that extra considerations such as pre-arity or post-arity can be added to the rules as shown in the previous section.

2) After Q until P - Between P and Q: Figure 6-c shows the After Q until P scope. In this case, an *Action* event occurring after *Q* must be followed by a *Response* event, but the *Response* event must occur before *P*. Note that the rule can be satisfied without the occurrence of *P*. Figure 6-d shows the between P and Q scope. In this case, both delimiters must occur, namely P and Q.

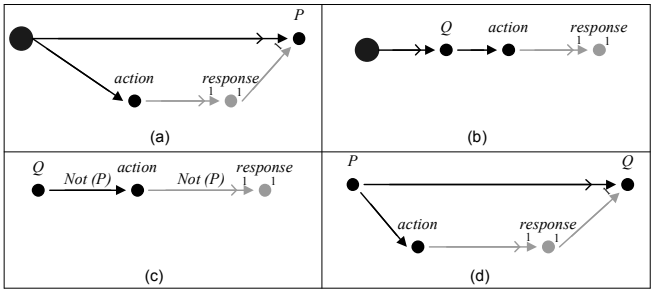

Fig. 6. Different scopes for the Response Pattern

B. Response Chain Pattern

We now consider a generalization of the Response pattern: the *Response-Chain* pattern. In this pattern a sequence of events $P_1, P_2, \ldots, P_n$ must always be followed by a sequence of events $Q_1, Q_2, \ldots, Q_n$. For simplicity and space reasons, we only consider one case of this pattern, where one stimuli must be followed by two responses, considering Global (figure 7-a) and Between P and Q scopes (figure 7-b).

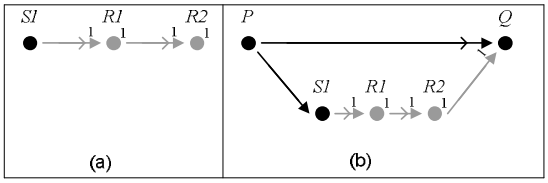

Fig. 7. One Stimuli-Two Responses Pattern

Note that the only difference with the rules describing the basic Response pattern (one stimuli - one response), shown in figures 4 (for Global scope) and 6-d (for Between scope), is just the inclusion of the second response, keeping the pattern specification simple and succinct.

C. Constrained Chain Pattern

This pattern introduces a variant for the Response-Chain pattern. In this case, the pattern restricts user specified events from occurring between pairs of events in the chain sequences. As we have already shown, restricting behavior is added very simply in our graphical language, as a label between the events involved. The example shown in this section is an extension for the One Stimuli-Two Responses Pattern, where a *W* event must no occur for the property to hold. The scopes modeled are Global (figure 8-a) and Between P and Q (figure 8-b).

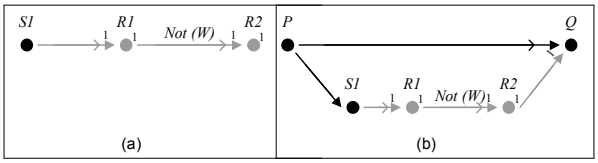

Fig. 8. Constrained Chain Pattern in FVS

D. Precedence Pattern

To conclude this section, we introduce another pattern: the Precedence pattern. The intent of this pattern is to describe relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second [8]. In this case, the occurrence of a *S* event must precede the occurrence of a *R* event. The scopes modeled are Global(figure 9-a) and Between P and Q (figure 9-b).

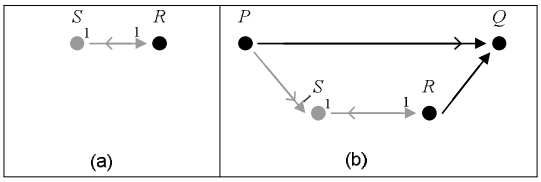

Fig. 9. Precedence Pattern in FVS

IV. Analysis and Comparison

We now compare FVS against the mentioned approaches considering the three quality attributes defined initially.

1) Succinctness: to compare FVS against the resulting LTL formulae in [8] and Propel's notation we propose the following alternative to somehow measure the complexity of the objects expressed in the formalisms. For the LTL formulae in [8], we measure their depth of nesting (DN) and the number of operators involved (O). For Propel's automata notation we measure the number of states (St) and number of transitions (T) of the automata described in [21] and for Propel's DNL templates we measure the numbers of statements (S) needed to express the property as detailed in [21]. For FVS rules we measure the number of points (P) and the number of restrictions (R) specified, including precedence and forbidden events. We compare the basic Response pattern with the following characterization (expressed in Propel's notation):

- Core Phrase (with Pre-arity, Immediacy and Post-arity options): One or more occurrences of *Action* eventually result in one or more occurrences of *Response*.
- Nullity Phrase: *Action* may occur zero times.
- Precedence Phrase: *Response* may occur before the first *Action* occurs.
- Repetition Phrase: The behavior is repeatable.

Regarding scopes, we take into account only Global and Between scopes. Tables I and II exhibit the obtained results for the Succinctness attribute. Table I is focused on the basic Response Pattern whereas table II considers the Response Chain pattern and the Constrained Chain pattern with one stimuli and two responses, and the Precedence pattern. In this case, FVS is only compared against LTL formulae: the first two are not available in Propel [6] whereas the Precedence Pattern is not completely described in [21].

TABLE I
Complexity comparison for the basic Response pattern

Formalism	Global	Between
LTL	DN=3,O=4	DN=4,O=12
Propel-Automata	St=3,T=6	St=5,T=12
Propel-DNL	S=7	S=10
FVS	P=2,R=1	P=4,R=4

TABLE II
Complexity comparison for the Response Chain Pattern, the Constrained Chain Pattern and the Precedence Pattern

	LTL		FVS	
	Global	Between	Global	Between
R-Chain	DN=2,O=6	DN=5,O=14	P=3,R=2	P=5,R=5
C-Chain	DN=3,O=9	DN=6,O=18	P=3,R=3	P=5,R=6
Precedence	DN=3,O=10	DN=5,O=13	P=2,R=1	P=4,R=4

One way of analyzing succinctness is to examine how the final objects grow when scope restrictions are added. As it is shown in tables I and II, the LTL formulae as given in [8] grow significantly when a more intricate scope is involved. Propel's automata representation also become more complicated, especially due to the growth of the number of transitions. Conversely, scopes in FVS are introduced seamlessly, just as any other restriction in the rules, without affecting succinctness. Propel's DNL notation handles scopes adequately. However, Propel suffers from some limitations when modeling scopes: delimiters in the scopes must be distinct, and there must be no intersection between events defining the intent and the scope of the pattern. These limitations are not present in FVS. Another interesting analysis comes up considering the Response Chain pattern with one stimuli and two responses. The only difference with the basic Response pattern is the inclusion of a second response. However, this simple modification causes a meaningful growth in the resulting LTL formulae, seriously threatening the scalability principle. On the other hand, FVS rules maintain a suitable complexity scaling appropriately. LTL formulae may be expressed more naturally or more succinctly employing modalities such as past or "from now on" operators [14], [15], but this require non trivial formulae manipulation, or even exponential blow-ups during the process [17].

2) Ease of Validation: in order to validate, understand and compare different patterns, the specification of the patterns expressed in the formalism must be easy to handle, manipulate and compare. These characteristics constitute the *Comparability* sub-attribute. This goal is hard to achieve when dealing with complicated LTL formulae or when comparing automata with several states and transitions. Formally, this would require testing language inclusion for automata or employing deductive simplification mechanisms for LTL formulae. The

situation is different in FVS' specifications. For example, the rule for the Response Chain pattern is the natural extension to the rule for the basic Response-pattern and this relationship can be visually depicted without extra manipulation. To mention another example, recall the Constrained Chain pattern in figure 8. The difference between the constrained version and the unconstrained version in figure 7 is clear when comparing both scenarios: the inclusion of the restriction through labeled arrows.

More elaborated analysis can also be gathered visually. As an example, recall the Response Chain pattern example considering Between scope (figure 7-b) and the Constrained Chain pattern considering the same scope (figure 8-b). In both rules antecedents are equivalent, but the consequent in figure 8-b is "stronger" than the consequent in figure 7-b, since it features more constrains. Thus, the rule depicted in figure 8-b is an specialization of the rule described in figure 7-b. The specialization relationship is a notion is similar to logical subsumption [4]. This also holds for the rules describing both patterns with Global scope (figures 8-a and 7-a). Now, when comparing rules in figure 7-a and 7-b it can be seen that although the consequent in figure 7-b is stronger than the consequent in figure 7-a, it is also the case that its antecedent is stronger too. Therefore in this case there is no specialization relationship. On the other hand, this kind of analysis is difficult to achieve when using an automaton notation or an LTL formula without proper manipulation. For example, the LTL version for the rules described in figures 7-a and 7-b presented in [8] are: $(\Box S1 \rightarrow \Diamond(R1 \wedge X \Diamond R2))$, for figure 7-a, and $\Box((P \wedge \Diamond Q) \rightarrow (S1 \rightarrow (\neg Q U (R1 \wedge \neg Q \wedge X(\neg Q U R2)))) U Q)$, for figure 7-b. By just looking at these formulae it is hard to recognize and understand the semantic relationship between them.

The other attribute composing the Ease of Validation attribute is *Complementariness*. FVS supports this feature by automatically generating anti-scenarios. Figure 10 illustrates two anti-scenarios for the Response pattern and Between Scope (in figure 6-d): figure 10-a models the case where the *Response* event did not occur in the trace after an *Action* event whereas in figure 10-b, the *Response* event did occur, but after the occurrence of *P*. Complementary behavior can be achieved by negating a formula or complementing an automaton, but this may involve non trivial operations.

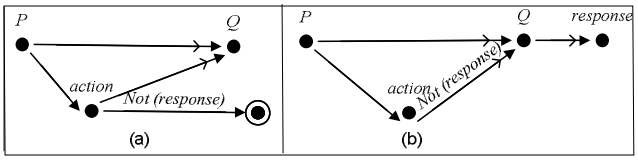

Fig. 10. Anti-scenarios for the Response pattern with Between scope

3) Modifiability: One interesting thing that can be observed in the Response Chain pattern is that the sequence of events involved must follow a strict order. In the example used here, with one stimuli *S1* and two responses *R1* and *R2*, response event $R1$ must precede response event $R2$. For some situations we would like to relax this condition, and to allow responses to occur in any order. This is easily achievable in FVS, since the only difference with the regular case is that no precedence restriction is present between both *Response* events. The rule in figure 11 shows this version considering Global scope.

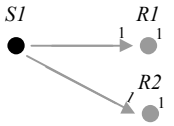

Fig. 11. A relaxed version of the One Stimuli-Two Responses Pattern

Finally, we conclude this section introducing another useful variant of the Response Chain pattern, where responses may occur before or after delimiter *Q*. In this case, the interpretation of the pattern is the following: an occurrence of stimuli *S1* between *P* and *Q* must always be followed by responses *R1* and *R2*. This version is suitable, for example, for modeling *race conditions*, a typical situation in multithreaded or concurrent systems. The rule in figure 12 reflects this behavior.

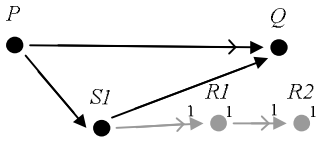

Fig. 12. A Variant of the One Stimuli-Two Responses Pattern

These examples show how patterns in FVS can be easily modified to fit in different situations or contexts. These variants of the Response Chain pattern are neither available in Propel nor in the LTL formulae available at [8]. With proper manipulation the developer could get an equivalent automaton or LTL formula, but again, this is bound to a complicated task.

V. Related Work

TimeEdit [20] and GIL (Graphical Interval Logic) [7] are two graphical specification languages based on timeline diagrams that do not feature partial ordering of events. TimeEdit is particularly focused on capturing complex chain events [2], while FVS stands for a more general approach. TimeEdit features a restricted notion of triggered scenarios using *required* events (events that are required to occur if all previous events have occurred). This limitation makes properties about past events, or events occurring in a certain scope, harder to specify and understand. GIL provides search operators to locate end points of intervals, similar to *next* and *previous* in FVS. However, it *previous* operator can not be applied freely as in FVS: interval recognition starts always forward a generic (or the first) point in the enclosing interval. Thus, easily expressible situation in FVS like freshness, correlation constrains or asserting the existence of a past in general can not be stated in GIL. Finally, specification of complex properties involving

several events in GIL requires nesting or stacking operators, threatening succinctness, ease of validation, and modifiability quality attributes. Other worth-mentioning approach is PSC (Property Sequence Chart) [2], which is inspired in UML 2.0 Interaction Sequence Diagrams. PCS's notation is also validated modeling property specification patterns. As said by the authors, it might be difficult to directly express properties in this language, and some automated assistance tool may still be need to help the developer [3]. Denoting complex constrains between events may require textual annotations. In addition, properties in PCS are described as anti-scenarios (e.g [1]) and not as conditional or triggered scenarios.

Other visual formalisms based on Message Sequence Charts such as [19], [22], [11] have been proposed for scenario-based specifications. We share with them the idea of using partial orders to describe scenarios. However, our work differs in several aspects. Our language is meant to express properties to be checked against a model or an implementation under analysis; we do not focus on creating an executable modeling language for different phases of the development process. FVS's trigger notation is distinguishable too: in our approach, the antecedent pattern is not required to predicate on a prefix of the behavior. Our consequents can refer to events occurring before the trigger or interleaved with its events.

Finally, to our best knowledge, none of the previously mentioned approaches is equipped with deductive features for comparability or complementariness reasoning.

VI. Conclusions and Future Work

Property specification is one of the most challenging task for the transference of software verification techniques. In this sense, the use of patterns has been proposed in order to hide from the developer the complicated handling of formal languages. However, when validating whether the pattern correctly expresses the desired property, the developer might face the translated version of the pattern into some specification's formalism. In this respect we identify three desirable quality attributes for the underlying formalism: *succinctness*, *ease of validation*, and *modifiability*. We show that typicaly used formalisms such as temporal logics or automata fail at some extent to support these characteristics. We propose FVS as a possible alternative to specify behavioral properties and we assess its performance by comparing it with two known approaches using one of the most commonly used pattern, the Response Pattern, and several variants of it. Other patterns such as the Constrained Chain pattern and the Precedence pattern are also considered. 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. Finally, since VTS can express real time properties, we would like to explore real time specification patterns [13], an extenstion to the patterns introduced by [9] considering timing requirements.

Acknowledgments

This work was partially funded by PICT 32440, PAE-PICT-2007-02278:(PAE 37279), PIP 112-200801-00955, UBACyT X021 and STIC-AmSud project TAPIOCA. Both authors are also affiliated to CONICET.

References

[1] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *Proceedings of the 26th International Conference on Software Engineering*, page 177. IEEE Computer Society, 2004.

[2] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14(3):293–340, 2007.

[3] M. Autili and P. Pelliccione. Towards a graphical tool for refining user to system requirements. *Electronic Notes in Theoretical Computer Science*, 211:147–157, 2008.

[4] V. Braberman, D. Garbervestky, N. Kicillof, D. Monteverde, and A. Olivero. Speeding Up Model Checking of Timed-Models by Combining Scenario Specialization and Live Component Analysis. In *FORMATS*, page 72. Springer, 2009.

[5] 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.

[6] R. Cobleigh, G. Avrunin, and L. Clarke. User guidance for creating precise and accessible property specifications. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, page 218. ACM, 2006.

[7] L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith, and Y. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):131–165, 1994.

[8] M. Dwyer, G. Avrunin, and J. Corbett. "Specification Patterns Web Site". In *http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml*.

[9] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE*, volume 99, 1999.

[10] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, page 266. ACM, 2003.

[11] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched lscs. In *MASCOTS '02*, pages 193–202. IEEE Computer Society.

[12] G. Holzmann. The logic of bugs. *ACM SIGSOFT Software Engineering Notes*, 27(6):87, 2002.

[13] S. Konrad and B. Cheng. Real-time specification patterns. In *Proceedings of the 27th ICSE*, pages 372–381. ACM, 2005.

[14] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *Proceedings- Symposium on Logic in Computer Science. pp. 383-392. 2002*, 2002.

[15] N. Markey. Temporal logic with past is exponentially more succinct. *EATCS Bull*, 79:122–128, 2003.

[16] D. Paun and M. Chechik. Events in linear-time properties. In *Proceedings of 4th International Conference on Requirements Engineering*. Citeseer, 1999.

[17] M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti. Practical model checking of LTL with past. In *ATVA03: 1st Workshop on Automated Technology for Verification and Analysis*. Citeseer, 2003.

[18] R. W. R. and K.Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on FSE(FSE-12)*, pages 158–169, 2004.

[19] B. Sengupta and R. Cleaveland. Triggered message sequence charts. In *SIGSOFT FSE*, pages 167–176, 2002.

[20] M. Smith, G. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Proceedings of the 5th IEEE International symposium on Requirements Engineering*, pages 14–22. Citeseer, 2001.

[21] R. Smith, G. Avrunin, L. Clarke, and L. Osterweil. Propel: An approach supporting property elucidation. In *ICSE*, volume 24, pages 11–21, 2002.

[22] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *Proc. of FSE '02*, pages 109–118. ACM Press, 2002.