International Journal of Software Engineering
and Knowledge Engineering
Vol. 25, No. 4 (2015) 669–700
© World Scientific Publishing Company
DOI: 10.1142/S0218194015500060

Specification Patterns: Formal and Easy*

Fernando Asteasuain[†] and Víctor Braberman[‡]
Dpto. Computación, FCEyN, UBA
Pabellón I, Ciudad Universitaria
(C1428EGA), Buenos Aires, Argentina
[†]*fasteasuain@dc.uba.ar*
[‡]*vbraber@dc.uba.ar*

Received 2 March 2011
Revised 3 September 2014
Accepted 5 September 2014

Property specification is still one of the most challenging tasks for transference of software verification technology. 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 desired property the developer may have to deal with the pattern representation in some particular formalism. For this reason, we identify four desirable quality attributes for the underlying specification language: succinctness, comparability, complementariness, and modifiability. We show that typical formalisms such as temporal logics or automata fail at some extent to support these features. Given this context we introduce Featherweight Visual Scenarios (FVS), a declarative and graphical language based on scenarios, as a possible alternative to specify behavioral properties. We illustrate FVS applicability by modeling *all* the specification patterns and we thoroughly compare FVS to other known approaches, showing that FVS specifications are better suited for validation tasks. In addition, we augment pattern specification by introducing the concept of violating behavior. Finally we characterize the type of properties that can be written in FVS and we formally introduce its syntax and semantics.

Keywords: Behavioral modeling; specification patterns; requirements engineering.

1. Introduction

Property specification is still one of the most challenging tasks for transference of software verification technology like model checking [7] and model-based testing [9, 34]. 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

*This work was partially funded by ANPCYT PICT 1774/11 and 0724/12, UBACYT W0813, UBACYT 20020130100384BA and MEALS 295261. Víctor Braberman is also affiliated to CONICET.

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. One alternative is the use of specification languages that resemble known and familiar programming languages, like JML [20] or Spec Explorer [35], used typically for pre and post contract specification. Nonetheless, when the intention is to predicate about the expected behavior over traces of reactive systems in a declarative way the mainstream options are formal formal approaches like Linear Temporal Logic (LTL) or operational notations such as finite-state machines. In these approaches users are required to have a considerable expertise on the formalism to accurately express the requirement they want to express (e.g. [14, 17, 13, 32, 8, 29, 3]) and this clearly constitutes not a minor obstacle.

The usage of specification patterns has been proposed as an interesting alternative to overcome this problem [12, 13]. The main purpose of a pattern is to capture recurring solutions to a particular type of problem. In [13], 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 [32]. 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, the user still needs to validate the property: "*Is this the right requirement*?". What is more, the user needs to be able to compare properties. For example, in order to select between two candidates properties the user might need to answer typical questions like: "*Which one constitutes a stronger formulation of requirements*?, *Which one imposes more restrictions*?, *How are these properties related and why*?". Another important input in order to validate a property comes from reasoning about violating behavior. That is, to reason about the generic scenarios of requirement's violation to gain insight into the sort of behavior that is being ruled out.

However, to perform all these validations tasks based solely on the DNL description of the patterns may be hard to achieve and many times the pattern translation into an formalism must be analyzed instead [17, 32]. Besides, sometimes a property needs to be slightly modified to suit actual system requirements, and this, again, suggests manipulations of the translated versions. Therefore, using patterns is not enough to entirely hide the subtleties of the underlying formalism from the user. This suggests the need of a formal specification language easy to use, and sufficiently expressive to enable skilled and non-skilled users to use it appropriately (e.g. [17, 26]). More specifically we propose four quality attributes desirable for the formalism to accurately handle validation tasks: *succinctness*, *comparability*, *complementariness*, and *modifiability*. 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) [2] is a simple fragment of VTS (Visual Timed Scenarios) [6], a visual language to define complex event-based requirements.

1.1. *Declarative description of properties: Existing difficulties*

As mentioned before, we propose the following quality attributes as desirable characteristics for a given formalism to accurately handle specification tasks: *succinctness*, *comparability*, *complementariness*, and *modifiability*. Succinctness, defined in [36], refers to how short a formula can be found to express a given property, and is considered one of the foremost quantitative measures when characterizing the expressive power of a formal language, and also a natural measure for comparing the strength of different formalisms [15]. In this sense, succinctness requires that the specification of a pattern expressed in the formalism should be as succinct as the DNL description of the pattern.

The resulting specification of the patterns should be easy to compare and distinguish. This objective constitutes the comparability attribute. For example, when comparing two related patterns it is natural to try to understand which one is more restrictive. More concretely, we define comparability as the ability to consider two related patterns specifications and understand the differences and similarities between them, and determine if there exists an embedding between both specifications.

Complementariness refers to the ability of reasoning about how the property is violated, which provides meaningful information to the specifier. The formalism should provide an easy way to reason about violating behavior. Specifically, we define complementariness as the ability of a language to generate general scenarios, expressed in that very same language, leading to a violation of a given property.

Finally, we define modifiability as the ability to manipulate objects expressed in the formalism, so that a specification can be adapted to subtle modifications in the application context, following the spirit of the changeability concept introduced by Parnas [25]. Small changes in the application context should lead to minor and localized modifications in the specification. This implies that the differences between the original and the modified specification should be small and localized.

We believe that the provided mappings of the patterns to LTL detailed in [12, 13] 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 and tool support. Reasoning about violating behavior is also troublesome since it requires sophisticated formulae manipulation. Further, since the user is dealing with complex formulae it is not always easy to gain insight about the obtained result from a tool. For example, it is hard to analyze the possible scenarios leading to a property violation directly from the negation of the property given by a tool. Similarly, the "no" or "yes" answer to an implication query between two LTL formulae might not be completely satisfactory, since it is demanding for the user to understand and debug why the answer is such. In addition, modifiability involves formulae manipulation. Appropriate manipulation of a formula with several arguments and operators, and/or a high depth of nesting is usually a difficult task and there is no tool support to help the specifier in the process.

If the underlying formalism is instead an automata-based one, 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 an automaton to adapt to different situations.

The expressive power of the formalism is also an interesting topic to point out. Complexity and expressivity of the formalism must be adequately balanced: the formalism must be expressive enough to denote all the properties of interest, keeping specifications simple and understandable for human comprehension. Similarly, an unnecessary increase in the expressive power of the formalism may lead to more complex specifications. For example, as is shown later on this work, it is not necessary for a formalism to have the same expressive power as full LTL in order to model all the specification patterns.

1.2. *Previous work and new contributions*

In previous work [2] we presented FVS as a property specification language. In that work we illustrated FVS features by describing three specification patterns: the Response pattern, the Precedence pattern and the Constrained Chain pattern. For each pattern we considered only two possible scopes, Global and Between. In contrast to our preliminary results, which were based on three patterns with two scopes [2], in this paper we now present a complete specification for all the patterns and each possible scope. That is, from the initial comparison with three patterns and only two scopes, we now present eleven specification patterns, considering the five scopes for each one. This allows for a richer and more exhaustive comparison against known approaches. In this way, results are not only valid for a subset of the specification patterns, but for the complete set of specifications patterns. In this work we demonstrate that those initial results can be generalized considering all the specifications patterns. In addition, in this paper we include a simple yet interesting example showing our approach in action through all the property specification process.

Also we show in this work how FVS is able to combine the properties proposed in the extended pattern specification in [32]. Moreover, on the top of this extension, we now introduce the notion of violating behavior for each pattern. That is, violating behavior is added to each pattern as a valuable information for the specifier. In our previous work [2] we claimed that formality is a key aspect regarding property specification. We now take this notion one step further presenting FVS syntax and semantics and we also characterize the type of properties that can be written in FVS, which constitutes an essential input to assess FVS expressive power. Finally, in this work we do not only present specification patterns but we also provide examples for pattern instantiation in concrete cases to highlight FVS applicability.

The rest of this paper is structured as follows. Background section (Sec. 2) describes the specification patterns mentioned in this paper and informally explains FVS, whereas its syntax and semantics are formally introduced in Sec. 3. Section 4 shows the FVS specification for every pattern described in [13], including each possible scope. Analysis and comparison with other formalisms are presented in Sec. 5. Finally, Sec. 6 mentions related work, and Sec. 7 presents future work and conclusions.

2. Background

In this section we briefly introduce specifications patterns and informally describe the main features of FVS. A formal characterization of the language is presented later in Sec. 3.

2.1. *Specification patterns*

Using patterns for property specification was first introduced by [13]. 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 first category includes four patterns, namely the Absence pattern, the Universality pattern, the Existence pattern and the Bounded Existence pattern whereas the former category defines other five patterns: the Precedence pattern, the Response pattern, the Precedence Chain pattern, the Response Chain pattern and the Constrained Chain pattern. All the patterns descriptions are available online [12] where also mappings to different formalisms are supplied. Specification patterns constitutes a highly valuable and noticeably representative case of study: a study presented in [13] collects 555 specifications from at least 35 different sources and it was found that 92% of them matched one of the specification patterns.

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 [12]. The other formalisms discussed here are those used in the Propel Language [32]: an extended finite-state automaton representation and a template representation, based on a restrict subset of natural language. Propel presents a very exhaustive extension to some of these patterns, which covers additional issues regarding patterns behavior. FVS is thoroughly compared to all of these approaches. All the specification patterns discussed in this work are those presented in [13], which are available online in [12]. We also discuss an extended version of these patterns described in [32].

2.2. *FVS: Feather weight visual scenarios*

FVS is a graphical language based on scenarios. Scenarios are partial order of events, consisting 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 Fig. 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 Fig. 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 Fig. 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 Fig. 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 Fig. 1(e) shows an equivalent version of the scenario in Fig. 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*. Both points are shown in Fig. 1(f).

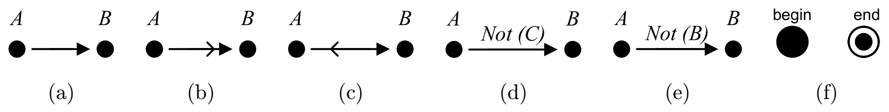


Fig. 1. Basic elements in FVS.

2.2.1. *FVS rules*

We now introduce the concept of Rule Scenario (RS) or simply a rule,[a] 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 also 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, 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

[a]Rule Scenarios represent the FVS's version of Conditional Scenarios available in VTS [6].

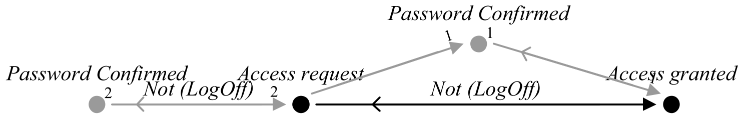


Fig. 2. Rule scenarios in FVS.

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 Fig. 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 (consequent 1) 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. Note the power of our trigger notation, where the antecedent need not precede the consequents in time.

2.2.2. *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 [6], 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 Fig. 2 is shown in Fig. 3. In this case, a valid password was not entered since the beginning of the trace and still access is granted.

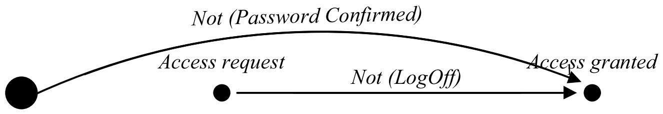


Fig. 3. An anti-scenario in FVS.

2.3. *On FVS expressive power*

We now briefly compare the expressive power of FVS and LTL with the aim of assessing the following question: *What kind of properties can be written in FVS*? Any property described in FVS can be written in LTL. Given any FVS rule η, we can obtain an anti-scenario $\neg\eta$, standing for the complementary behavior of η. The representation of $\neg\eta$ is just a plain scenario, a partial order of events considering precedence and occurrence restrictions. These type of properties can be expressed in LTL by linearizing and using the nesting of operator *Until*.

On the other hand, FVS as presented here is strictly less expressive than LTL: FVS can not express what is commonly known as strong fairness or compassion [22]. *Compassion* requires that if certain conditions are true infinitely often then certain other conditions must also hold infinitely often. In LTL *compassion* can be written as $\Box\Diamond p \rightarrow \Box\Diamond q$, where p and q are pure past formulas [21].

However, not being able to denote compassion does not impose a practical limitation to FVS for the defined purposes of this paper. More concretely, compassion is mostly used for synthesizing a system from a declarative specification [11, 27]. On the other hand, the kind of properties that we are interested in are focused on describing the expected behavior of an existing system, like the properties involved in the specification patterns. As it is shown later in this work, all of the specification patterns proposed by [13] can be modeled in FVS. In addition, another study in [23] indicates that these kind of properties are the most common properties in concurrent systems.

3. FVS Syntax and Semantics

We now formally define FVS syntax and semantics to provide a rigorous definition of the language. The reader that is not interested in the formality of the language may skip this section and is referred to Sec. 4: Pattern Specification in FVS.

We introduce FVS syntax and semantics as follows. First we present 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 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);
S6: $< \subseteq (P \uplus \{\mathbf{0}\} \times P \uplus \{\infty\}) \backslash \{\langle \mathbf{0}, \infty \rangle\}$ is a precedence relation between points ($\mathbf{0}$ and ∞ represent the beginning and the end of execution, respectively);
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

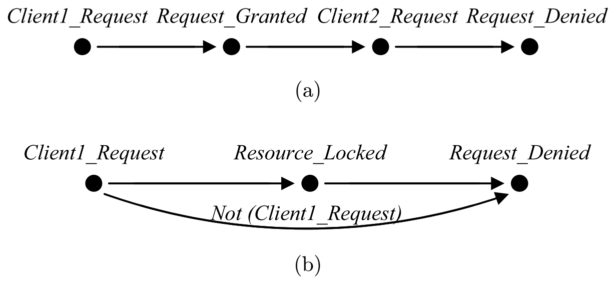


Fig. 4. Examples of FVS scenarios in a client-server architecture.

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 examples inspired in a small and simple protocol, where a server handles clients' requests trying to gain access to some resources. Scenario in Fig. 4(a) illustrates the occurrence of two events *Client1Request* and *Client2Request* standing for two requests from clients followed by their corresponding responses from the server. Scenario in Fig. 4(b) illustrates the occurrence of a client request, followed by an event *ResourceLocked* and a *ResponseDenied event.* In addition, the client does not raise another request until the answer is provided.

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 $f : \mathcal{S}_1 \rightarrow \mathcal{S}_2$) 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 $m : \mathcal{S}_1 \rightarrow \mathcal{S}_2$. 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 5 illustrates a morphism example (shown in dotted arrows). The scenario in the top of the figure (scenario $\mathcal{S}_2$) shows a sequence of requests and responses but also taking into account events reflecting the resource's status (locked or unlocked). On the other hand the scenario in the bottom of the figure (scenario $\mathcal{S}_1$) shows only a

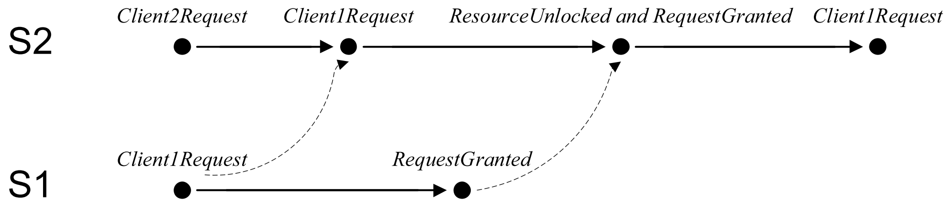


Fig. 5. A morphism example.

client's request followed by a server's response. It can be noted that scenario $\mathcal{S}_2$ features more constrains, since it considers the resource's status. In particular, considering the given morphism's definition is satisfied that $\mathit{Client1Request} \Rightarrow \mathit{Client1Request}$ and that $\mathit{ResourceUnlocked} \wedge \mathit{RequestGranted} \Rightarrow \mathit{RequestGranted}$ are tautologies.

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 $f_1 : \mathcal{S}_0 \to \mathcal{S}_1$, $f_2 : \mathcal{S}_0 \to \mathcal{S}_2, \ldots, f_k : \mathcal{S}_0 \to \mathcal{S}_k$ (*consequents*), we call $R = \langle \mathcal{S}_0, \{(S_i, f_i)\}_{i=1\ldots k}\rangle$ an FVS Rule.

As an example, consider the following rules in Fig. 6 modeling a portion of the expected behavior of the protocol introduced before. The rule in Fig. 6(a) says that every request will be eventually granted. Similarly, rule in Fig. 6(b) basically says

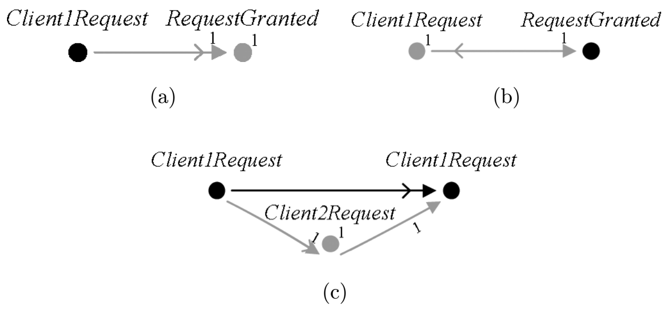


Fig. 6. FVS rules examples instantiating specification patterns.

that whenever an event *RequestGranted* occurs then there has occurred an event *ClientRequest* in the past. That is, every *RequestGranted* event must be preceded by a client's request. Finally, rule in Fig. 6(c) say that is not possible for *Client1* to raise two consecutive requests without the occurrence of a request from *Client2*. That is, between two consecutive requests from *Client1* there must exist a request from *Client2*, indicating a request's alternation between clients. These rules represent possible instantiations of some specification patterns: the Response pattern (Fig. 6(a)), the Precedence pattern (Fig. 6(b)) and the Existence pattern (Fig. 6(c)) [13]. These patterns are completely specified in FVS in Sec. 4. 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 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 a 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 $g : \mathcal{S} \rightarrow \mathcal{S}_\sigma$) 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: $\forall\, \mathrm{r} \in R\; \mathrm{S} \models \mathrm{r}$ and $\exists\, g$, a trace morphism $g : \mathcal{S} \rightarrow \mathcal{S}_\sigma$.

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. Pattern Specification in FVS

In this section we illustrate how FVS describe the specification patterns described in [13]. Initially we specify the one of the simplest and most used patterns, the Response pattern, introducing in Sec. 4.1 the notion of scopes. Next, Secs. 4.2 and 4.3 presents all the *occurrence* and *order* patterns respectively. Lastly, Sec. 4.4 deals with the extended pattern specification introduced by [32].

In the Response pattern 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 Fig. 7 reflects this behavior.

4.1. *Modeling scopes*

The previous section describes what the work in [13] defines as the intent of the pattern, that is, the structure of the defined behavior. This notion can be completed

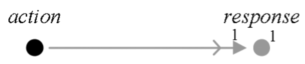


Fig. 7. The basic response pattern.

by defining a scope for the pattern, establishing the extent of program execution over which the pattern must hold. The available scopes defined in [13] are:

- **Global:** the pattern denotes the whole execution.
- **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.
- **After Q until P:** the pattern must hold after the occurrence of Q but before the occurrence of P. It is not necessary the occurrence of ending delimiter P for the pattern to hold.
- **Between P and Q:** Like the previous scope, the pattern must hold after the occurrence of P but before the occurrence of Q. However, in this case, both delimiters must occur for the pattern to hold.

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). We now briefly present in Fig. 8 the Response pattern, considering each of the other possible scopes.

After this introduction to FVS features modeling the *Response* pattern, we now complete patterns specification by modeling the rest of them in FVS. Note that in what follows we model all the specification patterns proposed by [13]. In addition, we describe each pattern considering the five available scopes.

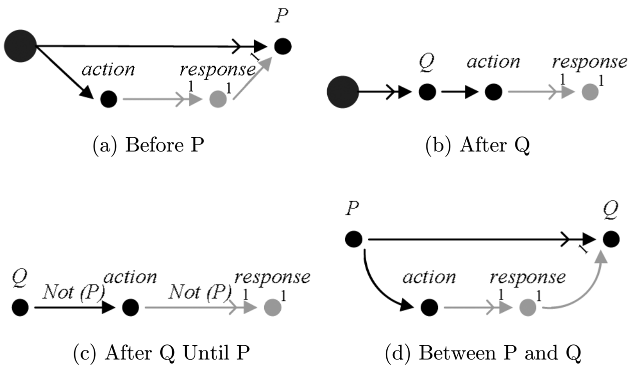


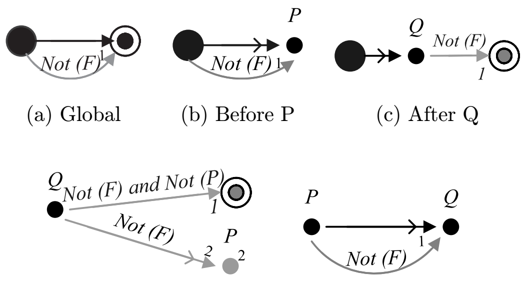
Fig. 8. Different scopes for the response pattern.

4.2. *Occurrence patterns*

The following subsections consider each of the four patterns in this category: Absence, Universality, Existence and Bounded Existence.

4.2.1. *The absence pattern*

The intent of this pattern is to describe a portion of a system's execution that is free of certain events. One of the most common examples of the Absence pattern is mutual exclusion between processes. In this context, the scope of the pattern would be a segment of the execution in which some process is in its critical section (i.e. between an enter section event and a leave section event) and the forbidden event would be the event that some other process enters its critical section. The next FVS rules in Fig. 9 describe this pattern considering each possible scope. In these rules, we consider the event F as event to be forbidden in the system's execution. That is, the valid traces of the system's execution are only those where the absence of the event F holds. For example, the rule for the global scope simply states that between the beginning and the end of the execution, event F does never occur. This behavior is then restricted according to the different scopes's requirements. Note that the rule describing the *Until* scope has two consequents denoting two possible admitted behavior. Since the scope can be satisfied with o without the occurrence of delimiter P, we consider two cases: one where P occurs and the other one where P does not occur. In the given rule, either consequent one or consequent two must occur: after the occurrence of the delimiter Q either both delimiter P and event F do not occur until the end of execution (consequent 1) or delimiter P actually occurs but event F does not occur in that segment (consequent 2).



(a) Global (b) Before P (c) After Q

(d) After Q Until P (e) Between P and Q

Fig. 9. The absence pattern covering different scopes.

4.2.2. *The universality pattern*

This pattern is suitable for those cases where a event occurs throughout a certain scope. Actually, this pattern describes more appropriately state-based systems than event-based systems. In state-bases systems, this pattern is used to express that a certain property must be true in each state included in the scope. To model this pattern considering events rather than states, we include an event standing for the case where the property is no longer true. Under this vision, this pattern can be reformulated as demanding the absence of such event. Therefore, this pattern can be

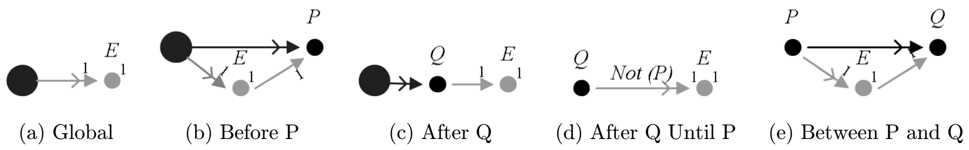


Fig. 10. The existence pattern modeled as FVS rules.

modeled in event-based systems using the *Absence* pattern. This reformulation behavior is based on the *Universality* pattern definition in [13], where it says that universality of an event can be viewed as the absence of its negation. Defining an event F as the event signalling the moment where the property of interest is no longer true, the rules in Fig. 9 completely describe this pattern.

4.2.3. *The existence pattern*

Related to both the Universality pattern and the Absence pattern, the existence patterns aims at describing a portion of a system's execution that contains an instance of certain events. The rules in Fig. 10 considers the case where an event E must indeed occur given a certain scope. These rules simply demand at least one occurrence of event E in each possible scope.

4.2.4. *The bounded existence pattern*

This patterns allows to depict a portion of a system's execution that contains at most a specified number k of instances of a designated event. According to the specified number k, this pattern is also known as the k-Bounded Existence pattern. For example, this pattern can be used to state that a client can be given access to a resource at most twice while another client is also waiting for that resource. For this pattern we consider the case where $k = 2$, i.e. the 2-Bounded Existence pattern, and event E stands for the "bounded" event.

Figure 11 illustrates this pattern considering all possible scopes. The rule for global scope says that given the first occurrence of the E event then it is the case that E occurs only more time (consequent number one), or simply E does not occur again until the end of execution (consequent number two). The behavior analysis is similar for the other rules describing other scopes. The *After Q until P* scope seems at first

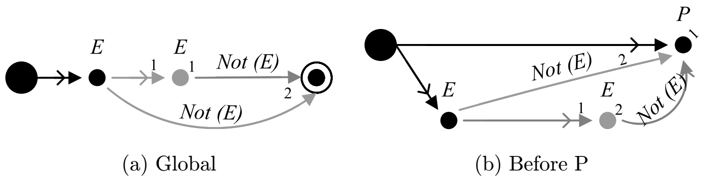
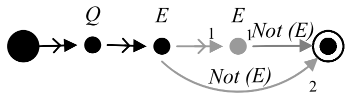
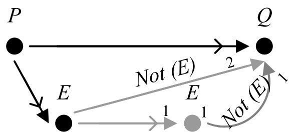


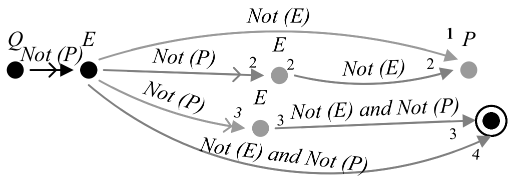
Fig. 11. The 2-bounded existence pattern.



(c) After Q



(d) Between P and Q



(e) After Q Until P

Fig. 11. (*Continued*)

sight a bit more complicated, since four consequents are involved. However, the mentioned consequents cover the basic behavior (after the initial E, either E occurs just one more time or simply does not occur at all until the end of the scope), but considering in this case not one but two possible delimiters for the scope: the occurrence of P (consequents 1 and 2), or the end of execution, that is, P does not occur (consequents 3 and 4).

4.3. ***Order patterns***

The order patterns predicate about relative order in which multiple events occur during system execution. The five patterns belonging to this category are: the Precedence pattern, the Response pattern, the Precedence Chain pattern, the Response Chain pattern and the Constrained Chain pattern. In next, we describe each one as FVS rules.

4.3.1. *The response pattern*

The response pattern was already fully described in this work in Figs. 7 and 8.

4.3.2. *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 [12]. In this case, the occurrence of a stimuli S event must precede the occurrence of a response R event. Figure 12 depicts this pattern considering all possible scopes.

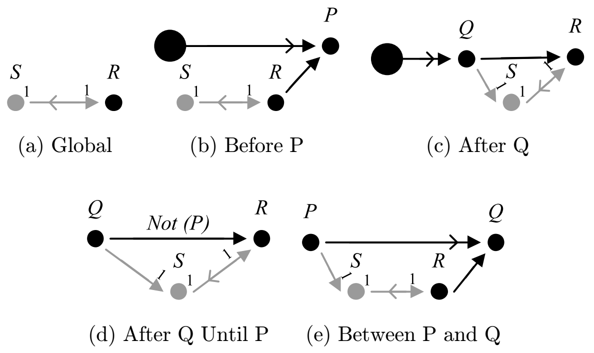


Fig. 12. Precedence pattern in FVS.

4.3.3. *The precedence chain pattern*

The Precedence pattern as well as the Response pattern can be generalized obtaining the Precedence Chain pattern and the Response Chain pattern. The generalized versions considers one or more responses following one or more stimulus as well. The precedence Chain pattern in particularly is used to describe a relationship between two sequences of events in which the occurrence of a sequence $R_1, R_2, \ldots, R_n$ within the scope must be preceded by an occurrence of a sequence $S_1, S_2, \ldots, S_n$ within the same scope. Following the pattern description at [12] we model just two cases for this pattern. In Fig. 13 one response R follows two stimulus $S1$ and $S2$ whereas in Fig. 14 two responses $R1$ and $R2$ follow stimuli S.

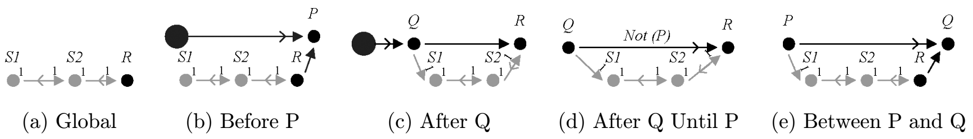


Fig. 13. Precedence chain pattern in FVS with two stimulus and one response.

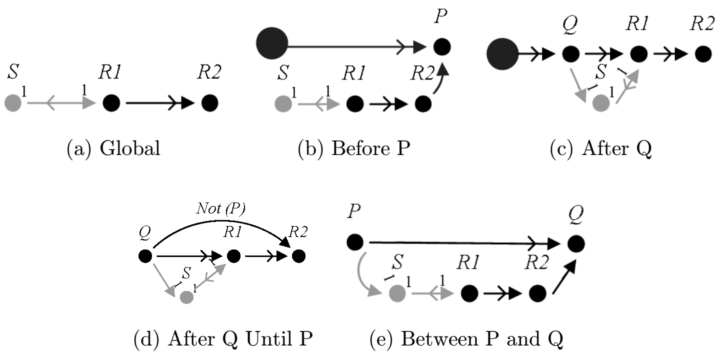


Fig. 14. Precedence chain pattern in FVS with one stimuli and two responses.

4.3.4. *The response chain pattern*

We now consider the generalization of the Response pattern: the *Response-Chain* pattern. In this pattern a sequence of events $S_1, S_2, \dots, S_n$ must always be followed by a sequence of events $R_1, R_2, \dots, R_n$. We first consider in Fig. 15 the case where one stimuli S must be followed by two responses R_1 and R_2, considering each scope.

Note that the only difference with the rules describing the basic Response pattern (one stimuli - one response), shown in Figs. 7 and 8, is just the inclusion of the second response, keeping the pattern specification simple and succinct. In order to provide a more complete description of this pattern, we now consider a second case, where two stimulus are followed by one response. All the rules for this variation are shown in Fig. 16.

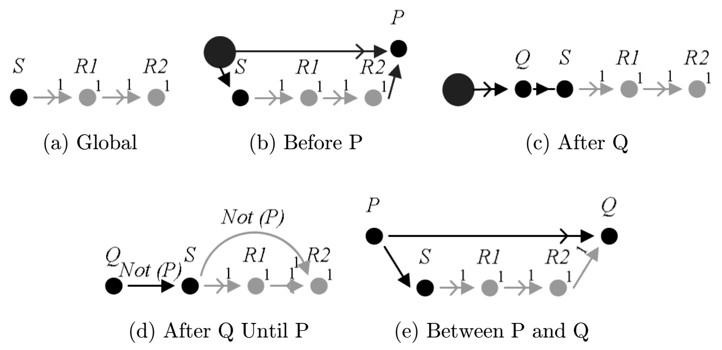


Fig. 15. Response chain pattern with one stimuli and two responses.

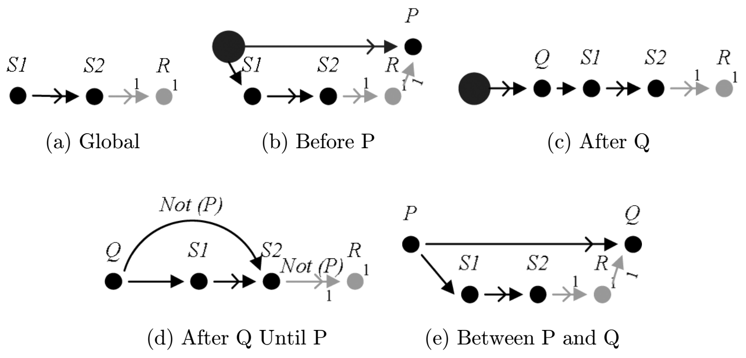


Fig. 16. Response chain pattern with two stimulus and one response.

4.3.5. *Constrained chain pattern*

This pattern introduces a variant for the Response-Chain pattern and the Precedence Chain pattern. In this case, this pattern restricts user specified events from

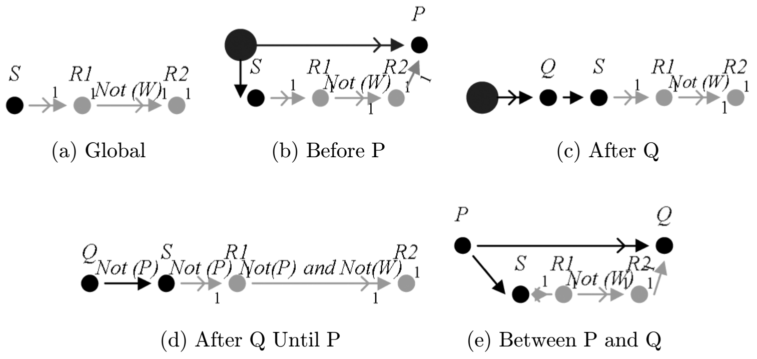


Fig. 17. Constrained chain pattern in FVS.

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 Fig. 17 is an extension for the Response Chain pattern with one stimuli and two responses, where a W event must no occur in the response chain sequence for the property to hold.

4.4. *Extended pattern specification*

Work in [32] proposes an extension to the patterns' specification, including properties such as: pre-arity, post-arity, immediacy, precedence, nullity and repeatability. In this sense, in [2] we showed how FVS was able to model this extended pattern definition, reflecting the flexibility of our notation. Now we illustrate how some of these evolved properties can be easily combined: pre-arity, post-arity, immediacy and nullity properties will be incrementally added to an initial specification. For this, consider the previously seen concrete specification for the Response Pattern in Fig. 6 (a). By combining pre-arity and post-arity, we can establish new restrictions specifying that events *ClientRequest* and *RequestGranted* must only occur in a interleaved fashion. This is achieved by the following rules: once a request occurs, no other request can arise until it is granted (pre-arity, in Fig. 18(a)). Similarly, in order to fulfill with the required interleaving, between two consecutive *RequestGranted* events, a *ClientRequest* must also occur (post-arity, in Fig. 18(b)).

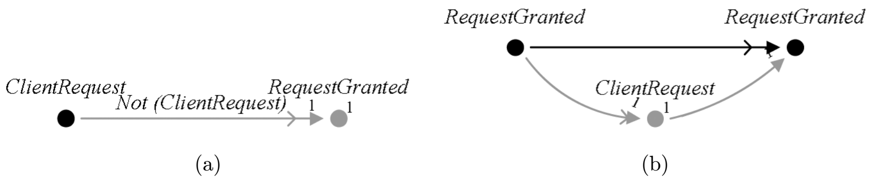


Fig. 18. Combining pre and post arity in the response pattern.

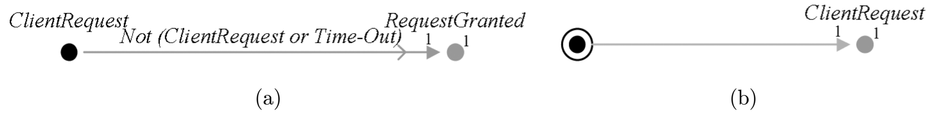


Fig. 19. Adding rules for immediacy and nullity properties.

On the top of this specification, we can easily add new restrictions. By using the immediacy property we can introduce a new requirement: between the request is dispatched and eventually granted, no Time-Out event is allowed to occur, as shown in rule 19-a. Note that this rule also implies the pre-arity property described early. The nullity property forces the developer to reason about the occurrence of the *ClientRequest* event: is a trace where *ClientRequest* event does never occur a valid trace of the system? Is that one the expected behavior of the system? If at least one request must occur, a new rule is added (see Fig. 19(b)).

The set of rules in Figs. 18(b), 19(a) and 19(b) model a possible instantiation of the extended definition of the Response pattern. As said, we enhance the initial specification by considering pre-arity, post-arity, immediacy and nullity properties. In our example, the occurrence of the *ClientRequest* leads to the occurrence of the *RequestGranted* event. In addition, *ClientRequest* event can not occur more times until the occurrence of the *RequestGranted* event, the *RequestGranted* can only occur one time after the occurrence of the *ClientRequest* event, no Time-Out event must occur in between, and finally, at least one request is required to occur. Only those traces satisfying these requirements will be considered valid.

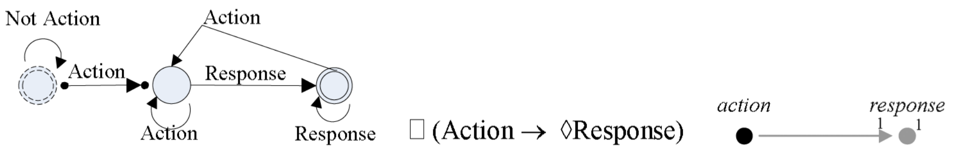
5. Analysis and Comparison

We now compare FVS against the mentioned approaches considering the quality attributes previously mentioned: succinctness, comparability, complementariness and modifiability. Additionally, Sec. 5.2 shows our approach and other notations in action developing an example focused in the property specification process.

5.1. *Quality attributes comparison*

To ease the comprehension of this section we present the specification of the Basic Response pattern in all the formalisms studied in this work. Using a DNL description such as the one proposed in Propel [32] we obtain the following characterization:

- 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.



(a) Automata Notation (b) LTL Notation (c) FVS Notation

Fig. 20. The basic response pattern in three different formalisms.

The following figure (Fig. 20) shows the very same pattern in Propel's automata notation, in LTL, and finally, as an FVS rule.

5.1.1. *Succinctness*

One way of analyzing succinctness is to examine how the specifications grow when scope restrictions are added, and more complex behavior is described. To compare FVS against the resulting LTL formulae in [12] 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 [12], we measure their depth of nesting (N) 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 [32] and for Propel's DNL templates we measure the numbers of statements (S) needed to express the property as detailed in [32]. For FVS rules we measure the number of points (P) and the number of restrictions (R) specified, including precedence and forbidden events.

As a first case of study, we compare only the basic Response pattern following the characterization previously introduced (see Fig. 20). Table 1 exhibits the obtained results for the Succinctness attribute, where all the approaches are compared. Automata and DNL entries in the second and third row respectively corresponds to the available notations in the Propel language.

As it is shown in Table 1 the LTL formulae as given in [12] 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:

Table 1. Complexity comparison for the basic response pattern.

Formalism	Global	Before	After	Between	Until
LTL	N = 1, O = 3	N = 2, O = 5	N = 3, O = 5	N = 4, O = 12	N = 4, O = 10
Automata	St = 3, T = 6	St = 4, T = 8	St = 4, T = 9	St = 5, T = 12	St = 5, T = 12
DNL	S = 7	S = 8	S = 8	S = 10	S = 10
FVS	P = 2, R = 1	P = 4, R = 4	P = 4, R = 3	P = 4, R = 4	P = 3, R = 4

Table 2. LTL formulae succinctness for occurrence patterns.

Pattern	Global	Before	After	Between	Until
Absence	N = 1, O = 2	N = 1, O = 4	N = 2, O = 4	N = 2, O = 8	N = 2, O = 6
Existence	N = 1, O = 1	N = 1, O = 4	N = 2, O = 6	N = 3, O = 8	N = 3, O = 8
2-Bounded	N = 4, O = 8	N = 8, O = 22	N = 6, O = 13	N = 10, O = 24	N = 9, O = 24
Universality	N = 1, O = 1	N = 1, O = 3	N = 2, O = 3	N = 2, O = 7	N = 2, O = 5

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.

We now introduce a second comparison considering all the patterns in the *Occurrence* category, comparing FVS and the LTL formulae given in [12]. Propel's notation are not included since complete patterns specification are not present in [32]. Table 2 shows the growth in the LTL formulae whereas Table 3 shows the growth in the FVS rules. The 2-Bounded entry in the fourth row of both tables refers to the Two-Bounded Existence pattern introduced before.

By analyzing Tables 2 and 3 we can see again an important growth of the LTL formulae, specially when scopes as *Between* or *Until* are specified. Undoubtedly, the Two-Bounded Existence pattern results in the most complex one. The given LTL formulae for this pattern are extremely complex not only for *Between* or *Until* scopes, but also for supposedly more simpler scopes as the *Before* scope, where 8 depth nesting formula and 22 operators formula is needed to specify its behavior. What is more, this growth can be easily incremented by specifying a number k greater than two (for example, the 3-bounded Existence pattern). On the other hand, the FVS rules also grow in complexity, but in a significant smaller growth-rate than the LTL formulae studied, easing the specification for a k-Bounded Existence pattern with $k > 2$.

Finally, we complete the succinctness comparison by introducing the results for the patterns in the *Order* Category. Again, a first table (Table 4) shows the growth in the LTL formulae, while a second table (Table 5) deals with FVS rules. In both tables P-Chain (x,y) and R-Chain (x,y) stands for the Precedence Chain pattern and Response Chain pattern with x stimuli and y responses. Similarly, the C-Chain entry corresponds to the Constrained Chain pattern introduced before. Again, FVS specifications grow in an smaller rate, enhancing scalability and keeping specifications simple and understandable.

Table 3. FVS succinctness for occurrence patterns.

Pattern	Global	Before	After	Between	Until
Absence	P = 2, R = 1	P = 4, R = 4	P = 4, R = 4	P = 4, R = 4	P = 3, R = 4
Existence	P = 2, R = 1	P = 3, R = 3	P = 3, R = 2	P = 3, R = 3	P = 2, R = 2
2-Bounded	P = 4, R = 6	P = 4, R = 7	P = 5, R = 7	P = 4, R = 7	P = 6, R = 12
Universality	P = 2, R = 1	P = 2, R = 1	P = 3, R = 2	P = 2, R = 1	P = 3, R = 3

Table 4. LTL formulae succinctness for order patterns.

Pattern	Global	Before	After	Between	Until
Response	N = 1, O = 3	N = 3, O = 8	N = 2, O = 5	N = 4, O = 12	N = 4, O = 10
Precedence	N = 0, O = 2	N = 2, O = 5	N = 2, O = 7	N = 3, O = 9	N = 3, O = 7
P-Chain(2,1)	N = 3, O = 10	N = 4, O = 11	N = 4, O = 16	N = 3, O = 13	N = 6, O = 13
P-Chain(1,2)	N = 2, O = 7	N = 2, O = 13	N = 3, O = 13	N = 3, O = 15	N = 4, O = 19
R-Chain(2,1)	N = 3, O = 9	N = 3, O = 13	N = 4, O = 12	N = 4, O = 15	N = 5, O = 24
R-Chain(1,2)	N = 2, O = 6	N = 4, O = 12	N = 3, O = 7	N = 5, O = 14	N = 4, O = 19
C-Chain(2,1)	N = 3, O = 9	N = 4, O = 16	N = 4, O = 10	N = 6, O = 18	N = 5, O = 25

Table 5. FVS succinctness for order patterns.

Pattern	Global	Before	After	Between	Until
Response	P = 2, R = 1	P = 4, R = 4	P = 4, R = 3	P = 4, R = 4	P = 3, R = 4
Precedence	P = 2, R = 1	P = 4, R = 4	P = 4, R = 4	P = 4, R = 4	P = 4, R = 4
P-Chain(2,1)	P = 3, R = 2	P = 5, R = 5	P = 5, R = 5	P = 5, R = 5	P = 4, R = 5
P-Chain(1,2)	P = 3, R = 2	P = 5, R = 5	P = 5, R = 5	P = 5, R = 5	P = 4, R = 6
R-Chain(2,1)	P = 3, R = 2	P = 5, R = 5	P = 5, R = 4	P = 5, R = 5	P = 4, R = 6
R-Chain(1,2)	P = 3, R = 2	P = 5, R = 5	P = 5, R = 4	P = 5, R = 5	P = 4, R = 6
C-Chain(2,1)	P = 3, R = 3	P = 5, R = 6	P = 5, R = 5	P = 5, R = 6	P = 4, R = 7

A useful analysis comes up considering the different variants of Response Chain and the Precedence Chain patterns. By just including one more stimuli or a response to the pattern causes a significant growth in the resulting LTL formulae, seriously threatening the scalability principle for any manual analysis at least. This behavior is more notable when considering *Between* or *Until* scopes. On the contrary, 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 [19, 24], but this require non trivial formulae manipulation, or even got exposed to exponential blow-ups during the process [28].

5.1.2. *Comparability*

As mentioned previously, we define comparability as the ability to consider two related patterns specifications and understand differences and similarities between them, and determine if there exists an embedding between both specifications. 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 Fig. 17. The difference between the constrained version and the unconstrained version in Fig. 15 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 (Fig. 15(b)) and the Constrained Chain pattern considering the same scope (Fig. 17(b)). In both rules antecedents are equivalent, but the consequent in Fig. 17(b) is "stronger" than the consequent in Fig. 15(b), since it features more constrains. Thus, the rule depicted in Fig. 17(b) is an specialization of the rule described in Fig. 15(b). The specialization relationship is a notion is similar to logical subsumption [5]. This also holds for the rules describing both patterns with Global scope (Figs. 17(a) and 15(a)). Now, when comparing rules in Figs. 15(a) and 15(b) it can be seen that although the consequent in Fig. 15(b) is stronger than the consequent in Fig. 15 (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 Figs. 15(a) and 15(b) presented in [12] are: $(\Box S1 \rightarrow \Diamond(R1 \wedge X\Diamond R2))$, for Fig. 15-a, and $\Box((P \wedge \Diamond Q) \rightarrow (S1 \rightarrow (\neg QU(R1 \wedge \neg Q \wedge X(\neg QUR2))))UQ)$, for Fig. 15(b). By just looking at these formulae it is hard to recognize and understand the semantic relationship between them and if not relationship holds what the underlying reason is.

Similar conclusions can be drawn analyzing the set of rules modeling a pattern under all the available scopes. Consider for example, the precedence pattern in Fig. 12. The main structure of the pattern, namely the occurrence of a stimuli S event must precede the occurrence of a response R event, is embedded in all the scopes. That is, even though different scopes are introduced, the structure of the pattern remains visible in every case. This fact clearly helps to fully understand the differences scopes helping the specifier to decide the one that fits better for the system being analyzed. In other specifications as the LTL formulas in [12] the main structure of the pattern may not be as easy to figure out because formulae became more complex, as seen before in this work.

5.1.3. *Complementariness*

This attribute refers to the ability of a language to generate general scenarios expressed in that very same language leading to a violation of a given property. FVS supports this feature by automatically generating anti-scenarios. Figure 21 illustrates two anti-scenarios for the Response pattern and Between scope (in Fig. 8(d)): Fig. 21(a) models the case where the *Response* event did not occur in the trace after an *Action* event whereas in Fig. 21(b), the *Response* event did occur, but after the occurrence of P.

We now provide further examples of violating behavior for the rest of the presented patterns, considering *Global* scope. Figure 22 presents violating behavior for the *Absence* pattern, the *Existence* pattern, the *two-Bounded Existence* pattern and

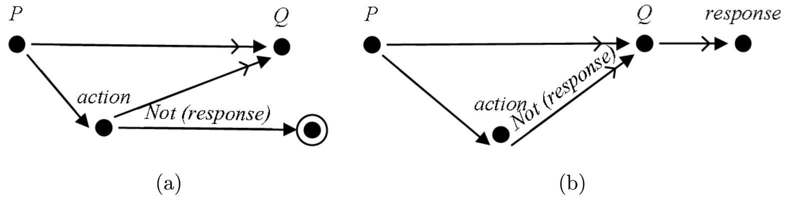


Fig. 21. Anti-scenarios for the response pattern with between scope.

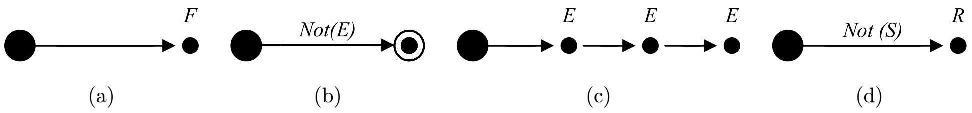


Fig. 22. Violating behavior for several specification patterns.

the *Precedence* pattern. Due to the dual relationship between the *Absence* pattern and the *Universality* pattern we provide only anti-scenarios for the former pattern. The anti-scenario for the *Absence* pattern is a very simple one. It shows the occurrence of the event F (see Fig. 22(a)). Similarly, an anti-scenario for the *Existence* pattern is an scenario where the event E does not occur throughout the entire execution (see Fig. 22(b)). For the 2-*Bounded Existence* pattern, an scenario representing a violation of the rule is one where there exists at least three occurrences of the E event (see Fig. 22(c)). Finally, for the Precedence pattern an anti-scenario consists of the occurrence of the response R such that stimuli S did not occur previously. This anti-scenario is depicted in Fig. 22(d).

For the Precedence Chain with two stimulus and one response, there exists four anti-scenarios. In the first one, the response occurred, but none of the stimulus (Fig. 23(a)). The following two cases stand for those cases where at least one stimuli does not occur, one for $S1$ and the other for $S2$ (Figs. 23(b) and 23(c)). The last case is where both stimulus occur, but in the wrong order. This is shown in Fig. 23(d).

Analogously, four anti-scenarios are presented in Fig. 24 for the Response Chain pattern with one stimuli and two responses, representing situations where the pattern is violated: none responses occur (24(a)), at least one response do not occur (24 (b) and 24(c)), or both responses occur, but in the wrong order (24(d)). Finally, anti-scenarios for the Constrained Chain pattern are presented in Fig. 25, where a W event must not occur trough out the response chain. In this case, a fifth anti-scenario

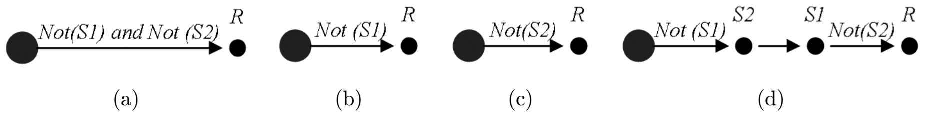


Fig. 23. Violating behavior for the precedence chain pattern.

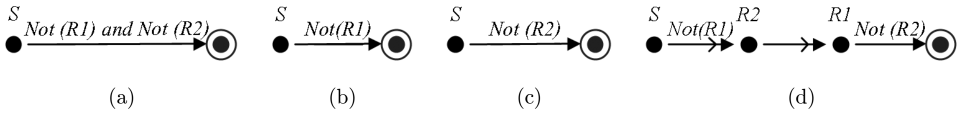


Fig. 24. Anti-scenarios for the response chain pattern.

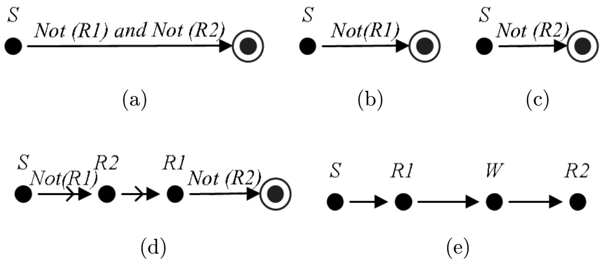


Fig. 25. Anti-scenarios for the constrained chain pattern.

arises in Fig. 25(e), standing for the extra case where both responses occur in the right order, but W event does also occur between both responses.

Violating behavior can be achieved by negating a formula or complementing an automaton, but this may involve non trivial operations. What's more, it may be difficult to reason about the obtained result. The main reason for this is that the result given by a tool when complementing an LTL formula or an automaton is another LTL formula or automaton, and it is difficult to understand directly from the formula or the automata what are the possible scenarios satisfying the negation of the formula and therefore violating the property.

5.1.4. *Modifiability*

This attribute is focused on analyzing the impact of changes in specifications. This implies that small changes in the application context should lead to minor and localized modifications in the specification. For example, 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 Fig. 26(a) shows this version. As a second example, we now consider 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*

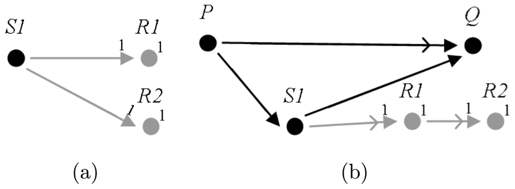


Fig. 26. Two variants of the one stimuli-two responses pattern.

conditions, a typical situation in multi-threaded or concurrent systems. The rule in Fig. 26(b) reflects this behavior.

Finally, we model a variation of the Constrained Chain pattern. In the original specification, a W event must not occur throughout the response chain sequence. In this variation, the restriction is extended, and W event must not occur not only through the response chain sequence but since the occurrence of the stimuli. The rule in Fig. 27 depicts this pattern variation considering each possible scope. Note that the only difference with the original pattern in Fig. 17 is simply the inclusion of the restriction *Not(W)* in the precedence arrow connecting the stimuli $S1$ with the first response $R1$.

These examples show how the specifications of the patterns in FVS can be easily modified to fit in different situations or contexts. With proper manipulation the developer could get an equivalent automaton or LTL formula, but again, this is bound to a complicated task.

5.2. *Approach in action*

To see our approach and other notations in action handling validation and description tasks, we consider a very simple yet interesting example presented in [17], where typical specification difficulties when using formal languages and specification

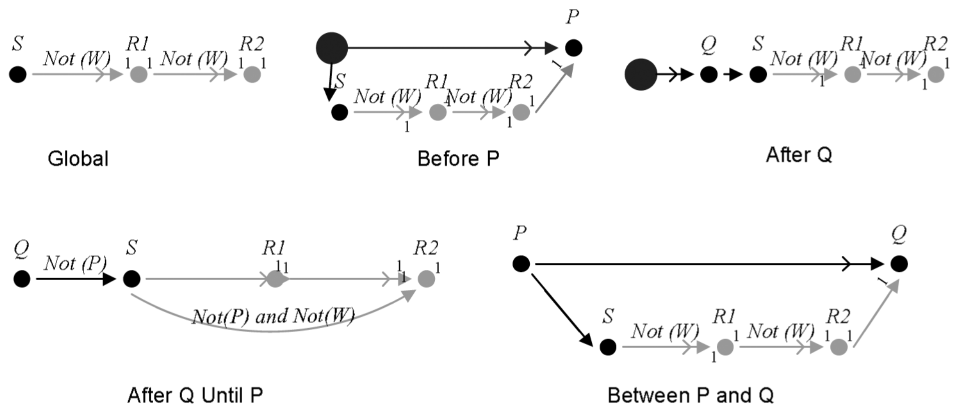
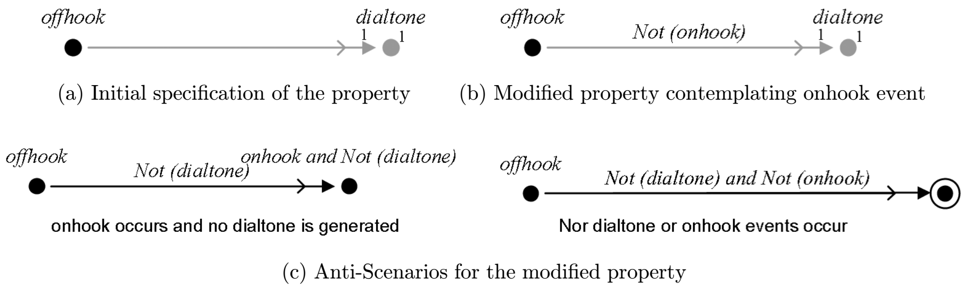


Fig. 27. A variant of the constrained chain pattern.

patterns are described. In particular, we are going to analyze the example focusing on the Response pattern. Authors in [17] also consider the Absence pattern, but similar problems are found.

The main goal is to specify, in a formal language, the following property of a telephone system: *When the subscriber picks up the phone, dial-tone is always generated.* The specifier may detect this as an *action-response* property, and decides to use in consequence the Response pattern. As said before, the LTL formula proposed in [12] for this pattern is: $\Box(action \rightarrow \Diamond response)$. To instantiate this pattern the specifier replaces the general *action* and *response* events for actual events in the system. In this case, those would be the *offhook* and *dialtone* events. Therefore, the concrete LTL formula obtained is $\Box(offhook \rightarrow \Diamond dialtone)$. However, as noted in [17] this specification does not reflect the expected behavior. For example, the following situation is not ruled out. If the subscriber fails to hear a dial-tone, he may return the phone *onhook* and try one or more times, until *dialtone* is generated, which is clearly not the intent. To solve this problem the authors in [17] first tried to modify the LTL formula, but then realized it was easier to specify the negation of the property rather than the property itself, and then prove it to be unfeasible. From this point on, the use of specification patterns is dropped and the property is fixed by proper LTL formulae manipulation. The new proposed formula is the following: $\Diamond(offhook \rightarrow (\neg dialtone \text{ U } onhook))$. Nonetheless, this LTL formula is not yet correct and more fixes are needed, as explained in [17]. The final LTL formula obtained is the following: $\Diamond\ (offhook \wedge \text{X}\ ((\neg dialtone \wedge \neg onhook) \text{ U } (\neg dialtone \text{ U } onhook))$.

Now we describe the same process but using FVS as the specification language. The instantiated Response pattern is illustrated in Fig. 28(a). In order to solve the problem that arises when the *dialtone* is generated not immediately but after several attempts, a simple restriction is added to the scenario: the *onhook* event must not occur between *offhook* and *dialtone* events, as shown in Fig. 28(b). Going one step further, the specifier can generate anti-scenarios for the rule in 28(b) for validation purposes, which are shown in Fig. 28(c). The first one shows an error when the subscriber returns the phone *onhook* and fails to hear a dialtone, behavior that matches the previously mentioned final LTL formula given in [17]. The second anti-


offhook
dialtone
1
1
offhook
Not (onhook)
dialtone
1
1
offhook
Not (dialtone)
onhook and Not (dialtone)
onhook occurs and no dialtone is generated
offhook
Not (dialtone) and Not (onhook)
Nor dialtone or onhook events occur
(a) Initial specification of the property (b) Modified property contemplating onhook event

(c) Anti-Scenarios for the modified property

Fig. 28. Specifying and validating the telephone system property in FVS.

scenario in Fig. 28(c) denotes a situation where dialtone is never generated, but in addition the subscriber never returns the phone *onhook*. It is worth noticing that this last situation is not contemplated in the final LTL formula proposed in [17], which fails to detect this behavior as an error.

Discussion. Specifying the property based on the use of specification patterns, LTL formulae and automata manipulation definitely exhibits some faults. In order to detect the error when *dialtone* is generated by repetition, the specifier must be familiarized with LTL. Specification patterns help the specifier to obtain the LTL formula, but in order to validate it the specifier must have a thorough understanding of the proposed LTL formula. Once an error is detected, the LTL formula must be adjusted to reflect the intended behavior. That is, specification patterns can no longer help the specifier and the problem is solved in [17] by manipulating the LTL formula. This iterative-fixing procedure may be hard and error-prone, even if the specifier is trained in temporal logics. As an example, authors in [17] are forced to model the negation of the property instead of the property itself for complexity reasons. An analogous analysis would be gathered if an automata based notation was used instead.

A different scene occurs when employing FVS as the specification language. The detected error is easily fixed by adding a simple restriction. In addition, the specifier can visually depict how the original rule is embedded in the modified one, exposing the benefits of the *comparability* attribute. The only difference between both rules is the forbidden event labeling the arrow. This is a small and localized change which corresponds to the change in the requirements expressed in natural language. Moreover, the anti-scenarios generated for the modified rule in FVS allow to discover an error situation which is not captured in the final specification for the property given in [17]. Finally, an interesting remark to be made is that with FVS the specifier maintains the specification modeling the *positive* version of the system requirement, instead of moving into reasoning about the negation of the property.

6. Related Work

TimeEdit [31] and GIL (Graphical Interval Logic) [10] 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 [3], 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) [3], 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 [4]. 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 [30, 33, 16] 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. To our best knowledge, none of the previously mentioned approaches is equipped with deductive features for comparability or complementariness reasoning. Finally, specification patterns are extended in [18] considering timing requirements. Since VTS [6] can express real time properties, it is reasonable to expect that these real-time patterns can be adequately and naturally expressed on it.

7. Conclusions and Future Work

In this work we propose FVS as a possible alternative to specify behavioral properties and we assess its performance by comparing it with three known formalisms analyzing all of the specification patterns and scopes proposed by [13], obtaining that FVS is apparently better suited for validations tasks. We enhance pattern specification by not only including the extension proposed in [32], but also depicting violating behavior for each pattern. In addition, we demonstrate how different features extending pattern specifications can be naturally combined in FVS. We show that FVS, although less expressive than LTL, is capable of specifying all of the specification patterns. Finally, we present the formal syntax and semantics of our language, and we provide further examples by instantiating some of the patterns in concrete cases.

The obtained results may have some threats to validity. Based on the analysis and conclusions of several works like [3, 8, 13, 14, 17, 29, 32] we pointed out that formalisms used to describe the expected behavior of reactive systems in a declarative way fail at some extent to support validation tasks. We propose using four quality attributes (succinctness, comparability, complementariness and modifiability) as one possible way to establish how well a given formalism handle validation tasks. We do not validate that fulfilling these attributes helps to meet the goal of easing properties' description and validation. This would require an empirical user study experiment which is out of the scope of this paper. In addition, we do not claim that the quality attributes we propose constitute a complete set of desirable features. However, some authors do propose these activities as usual and useful validation tasks [10, 17, 31, 32]. On the other hand we demonstrated that FVS can model all the specification patterns, including all the scopes. What is more, we showed that FVS adequately supports validation tasks based on the proposed quality attributes.

Regarding future work, we are considering enhancing FVS's expressive power to enable expressing arbitrary ω-regular languages. This can be achieved by introducing *auxiliary events* which allows the user to predicate about properties in a higher level of abstraction. We are also working on defining a synthesis algorithm for FVS's rules, enabling the possibility of elaborated automatic analysis. Furthermore, we are planning to improve FVS features by supporting validation of a set of requirements.

References

1. A. Alfonso, V. Braberman, N. Kicillof and A. Olivero, Visual timed event scenarios, in *6th ICSE'04* (2004), pp. 168–177.
2. F. Asteasuain and V. Braberman, Specification patterns can be formal and also easy, in *Software Engineering and Knowledge Engineering (SEKE)* 2010, pp. 430–436.
3. M. Autili, P. Inverardi and P. Pelliccione, Graphical scenarios for specifying temporal properties: An automated approach, *ASE* **14**(3) (2007) 293–340.
4. M. Autili and P. Pelliccione, Towards a graphical tool for refining user to system requirements, in *ENTCS*, Vol. 211, 2008, pp. 147–157.
5. 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* (Springer, 2009).
6. V. Braberman, N. Kicillof and A. Olivero, A scenario-matching approach to the description and model checking of real-time properties, *IEEE TSE* **31**(12) (2005) 1028–1041.
7. E. Clarke, O. Grumberg and D. Peled, *Model Checking* (MIT Press, 1999).
8. R. Cobleigh, G. Avrunin and L. Clarke, User guidance for creating precise and accessible property specifications, in *Proc. of 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2006, p. 218.
9. S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton and B. Horowitz, Model-based testing in practice, in *ICSE*, 1999, pp. 285–294.
10. L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith and Y. Ramakrishna, A graphical interval logic for specifying concurrent systems, *ACM Trans. Software Engineering and Methodology* **3**(2) (1994) 131–165.
11. N. D'Ippolito, V. Braberman, N. Piterman and S. Uchitel, Synthesis of live behaviour models, in *Proc. of 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.
12. M. Dwyer, G. Avrunin and J. Corbett, Specification Patterns Web Site, in *http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml*.
13. M. Dwyer, G. Avrunin and J. Corbett, Patterns in property specifications for finite-state verification, in *Proc. of 21st International Conference on Software Engineering (ICSE)*, Vol. 99, 1999.

14. D. Giannakopoulou and J. Magee, Fluent model checking for event-based systems, in *Proc. of 9th European Software Engineering Conference*, 2003, p. 266.
15. M. Grohe and N. Schweikardt, The succinctness of first-order logic on linear orders, in *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, 2004, pp. 438–447.
16. D. Harel and R. Marelly, Playing with time: On the specification and execution of time-enriched LSCS, in *MASCOTS '02*, pp. 193–202.
17. G. Holzmann, The logic of bugs, *ACM Software Engineering Notes* **27**(6) (2002) 81–87.
18. S. Konrad and B. Cheng, Real-time specification patterns, in *Proc. of 27th ICSE*, 2005, pp. 372–381.
19. F. Laroussinie, N. Markey and P. Schnoebelen, Temporal logic with forgettable past, in *Proceedings of Symposium on Logic in Computer Science*, 2002, pp. 383–392.
20. G. Leavens, A. Baker and C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, *ACM SIGSOFT Software Engineering Notes* **31**(3) (2006) 1–38.
21. Z. Manna and A. Pnueli, A hierarchy of temporal properties, in *Proc. of Ninth ACM PODC*, 1987, pp. 205–205.
22. Z. Manna and A. Pnueli, Completing the temporal picture, in *Automata, Languages and Programming*, 1989, pp. 534–558.
23. Z. Manna and A. Pnueli, *A Temporal Proof Methodology for Reactive Systems* (Springer-Verlag, 1995).
24. N. Markey, Temporal logic with past is exponentially more succinct, *EATCS Bull.* **79** (2003) 122–128.
25. D. Parnas, On the criteria to be used in decomposing systems into modules, *Commun. ACM* **15**(12) (1972) 1053–1058.
26. D. Paun and M. Chechik, Events in linear-time properties, in *Proc. of 4th International Conference on Requirements Engineering*, 1999.
27. N. Piterman, A. Pnueli and Y. Sa'ar, Synthesis of reactive (1) designs, in Lecture Notes in Computer Science, Vol. 3855, 2006, pp. 364–380.
28. M. Pradella, P. San Pietro, P. Spoletini and A. Morzenti, Practical model checking of LTL with past, in *ATVA03*, 2003.
29. R. W. R. and K. Viggers, Implementing protocols via declarative event patterns, in *ACM Sigsoft International Symposium on FSE*, 2004, pp. 158–169.
30. B. Sengupta and R. Cleaveland, Triggered message sequence charts, in *SIGSOFT FSE*, 2002, pp. 167–176.
31. M. Smith, G. Holzmann and K. Etessami, Events and constraints: A graphical editor for capturing logic requirements of programs, in *Proc. of 5th IEEE RE*, 2001, pp. 14–22.
32. R. Smith, G. Avrunin, L. Clarke and L. Osterweil, Propel: An approach supporting property elucidation, in *ICSE*, Vol. 24, 2002, pp. 11–21.
33. S. Uchitel, J. Kramer and J. Magee, Negative scenarios for implied scenario elicitation, in *Proc. of FSE*, 2002, pp. 109–118.
34. M. Utting and B. Legeard, *Practical Model-based Testing: A tools approach* (Morgan Kaufmann, 2007).
35. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann and L. Nachmanson, Model-based testing of object-oriented reactive systems with spec explorer, in *Formal Methods and Testing*, 2008, pp. 39–76.
36. T. Wilke, CTL+ is exponentially more succinct than CTL, in *Foundations of Software Technology and Theoretical Computer Science*, 1999, pp. 110–121.