Exploring Architectural Model Checking with Declarative Specifications ⋆

Fernando Asteasuain[1,3] and Francisco Tarulla[2]

[1] Universidad Nacional de Avellaneda, Dpto Tecnología y Administración, Avellaneda, Argentina. fasteasuain@undav.edu.ar
[2] Universidad de Buenos Aires, Dpto de Computación, Buenos Aires, Argentina. ftarulla@dc.uba.ar
[3] UAI-CAETI, Buenos Aires, Argentina.

Abstract. In this work we explore the FVS language in the context of architectural behavior model checking. FVS holds desirable characteristics for this particular domain. Its flexible notation enables the possibility of performing behavioral exploration when denoting the properties to be satisfied. In addition, FVS expressive power capable of denoting ω-regular properties is useful to denote behavior in a higher level of abstraction. These are two key activities when specifying and validating a system architecture. Given that FVS specifications can be translated into Büchi automata they can be used as input in a validation tool like model checkers. In this sense, we conducted industrial relevant case studies to apply our approach in concrete examples.

Keywords: Software Architecture, Model checking, Declarative Specifications

1 Introduction

One of the most crucial concepts inherent to any Software Engineering activity is the usage of models and abstraction to be able to reason, explore and specify the expected behavior of the system to be built. In this sense, the design and analysis of Software Architectures constitutes a challenging and significant corner stone to achieve these objectives [8].

When specifying architectural behavior it is essential being able to explore and reason about different alternatives since most of the requirements are still yet to be discovered and defined. This is particulary true when trying to formally validate the expected behavior by performing architectural model checking. A model checker is given as input a model abstracting the system to be developed and a set of properties that the model should satisfy. The topic of formal languages to describe architectural behavior has been tackled by a plethora of approaches such as [13, 22, 17, 19]. However, it has been pinpointed by the community that writing the expected behavior in the form of properties is still one of

⋆ This work was partially funded by UNDAVCYT 2014, PAE-PICT-2007-02278:(PAE 37279), PIP 112-200801-00955 and UBACyT X021, UAI-CAETI

the main challenges to be addressed [2, 20, 1, 3]. Some of the most relevant issues included in this challenge are: the usage of formal languages such as temporal logics which may be hard to adopt, the usage of operational notations such as Automata-based notations or ADL's (Architecture Description Languages) whose constructors and structure resembles source code that might lead to premature implementations decisions and a certain lack of flexibility and expressive power in the notations used.

Given this context we explore in this work the Feather Weight Visual Scenarios (FVS) language as a declarative language to denote and validate architectural behavior. FVS is a declarative language based on graphical scenarios and features a flexible and expressive notation with clear and solid language semantics. FVS expressivity is a distinguished characteristic among declarative approaches since it is able to denote ω-regular properties. This makes FVS, for example, more expressive than LTL (Linear Temporal Logic). This fact enables the user to predicate and express behavior in a higher lever of abstraction which is a key factor in the architectural domain. FVS's specifications can be translated into Büchi automata enabling the possibility of realizing architectural model checking. We explore this alternative by analyzing industrial relevant case studies.

1.1 Previous work and new contributions

In [5, 6] FVS was used in the software architectures domain, denoting variability among the specification of product family architectures and expressing the behavior of architectural connectors. We now build on the top of those works introducing the following new aspects:

* Architectural behavior specifications include not only connectors, but also components and other relevant architectural interactions.
* We take FVS's specifications one step further by using them as input in a known model checker: LTSA (Labeled Transition System Analyzer [14]). In this sense, we present a tool denominated GTxFVS implementing FVS features and enabling the interaction with the LTSA model checker.
* We exploit FVS expressive power to denote ω-regular properties to reason in a higher level of abstraction. In particular, we use this FVS's power to denote behavior imposed by the layered architectural style.
* We introduce relevant case studies exhibiting FVS performance within the context of architectural model checking.

The rest of this work is organized as follows. Section 2 introduces FVS's main features. Section 3 describes the analyzed case studies. Section 4 presents some lessons learned when developing the case studies and summarizes conclusions of the work. Finally, Section 5 briefly discuss related work and raises some points regarding future work.

2 Feather weight Visual Scenarios

In this section we will informally describe the standing features of FVS [3, 4]. The reader is referred to [4] for a formal characterization of the language. FVS is a

graphical language based on scenarios. Scenarios are partial order of events, consisting of points, which are labeled with a logic formula expressing 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. Events labeling an arrow are interpreted as forbidden events between both points. In figure 1-c A-event precedes B-event such that C-event does not occur between them. Finally, FVS features aliasing between points. Scenario in 1-d indicates that a point labeled with A is also labeled with $A \wedge B$. It is worth noticing that A-event is repeated on the labeling of the second point just because of FVS formal syntaxis.

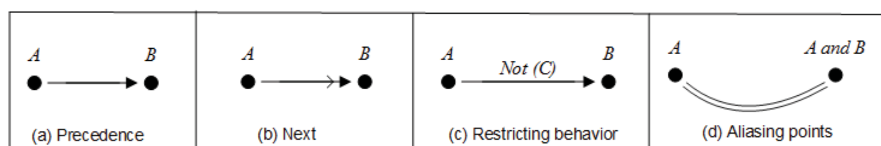

Fig. 1. Basic Elements in FVS

We now introduce the concept of FVS rules, 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. 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 rule describes the circumstances under which writing in a pipe is valid. For every occurrence of a write event, then it must be the case that either the pipe did not reach its maximum capacity since it was ready to perform (Consequent 1) or the pipe did reach its capacity, but another component performed a read over the pipe (making the pipe available again) afterwards and the pipe capacity did not reach again its maximum (Consequent 2).

Ghosts Events and Translation into Büchi automata FVS is expressive enough to denote ω-regular properties [4]. This is due to the introduction of

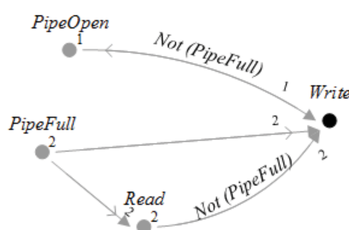

Fig. 2. An FVS rule example

abstraction, which is incorporated in our notation by introducing a new type of events. By using these events, the specifier can abstract behavior and reason about events that are not present in the system traces, but actually represent a higher level of abstraction. We call these special events as "ghost" events, in contrast with "actual" events, the set of events present in the system's specification. In order to verify that a certain trace of the system (which only contains actual events) satisfies a rule containing ghosts events there is a internal procedure based on morphisms that discards ghost events based on a classic process of existential elimination [4]. Finally, there is a tableau procedure that translates FVS rules into Büchi automata [4]. In this way FVS specifications can play the role of input properties to be analyzed by a model checker.

FVS tool implementation: GTxFVS The tool GTxFVS basically implements the FVS language. It is based on the Meteor platform[4] , an open source platform for web, mobile, and desktop applications. GTxFVS also implements the tableau algorithm which translates FVS rules into Büchi automata and it allows the interaction with the LTSA model checker so that architectural model checking can be performed.

3 Case Study

In this section we describe the case studies illustrating our approach. Section 3.1 shows the formal validation of a known Publish/Subscribe based application whereas Section 3.2 exhibits FVS expressive power verifying properties in a layered system.

3.1 Verifying SIENA's architectural behavior

We applied our approach within the context of a known Publish/Subscribe event notification service called SIENA [11] developed at University of Colorado. This system was analyzed in [9, 10] were a model representing a minimal schema of SIENA is given. In what follows we describe and verify the behavior of the

[4] (https://www.meteor.com/)

SIENA system based on this latter work. Two mains components highlight in the Siena's architecture specification: *clients* and *event-service*. The event-service is composed by a number of servers which offers clients the publish/subscribe interface. Clients are both publishers and subscribers. Subscribers express their interest in events by supplying a *filter*. We described, specified and verified three main properties of the SIENA model. It is worth to point out that these properties are presented in [10, 9] as essential to the behavior of the system. The mentioned properties are:

– Property 1: If a component *C0* subscribes a filter expressing interest in component *C1* publications' and *C1* publishes an event then C0 receives the corresponding notifications unless C0 unsubscribes its filter.
– Property 2: Servers must process events in the same order they were received.
– Property 3: Events received by a component which are generated by the same source maintain the publication order.

The FVS rule in Figure 3 specifies the behavior required by *Property 1*. The following events are involved: Event *ACT-FI* (component *C0*'s filter is active in the event-service structure), *PUB-C1* (component *C1* publishes an event), *NOT-C0* (component *C0* receives an event notification) and *UNS-C0*(component *C0* unsubscribes the filter). The rule simply states that whenever *C1* publishes an event given that *C0*'s filter is active, then *C0* receives its notification. The condition in the rule scenarios (*Not UNS-C0*) checks that component *C0* did not unsubscribed the filter.

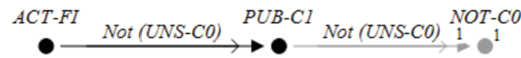

Fig. 3. All the publications must be received by the subscribers

Rule in Figure 4 shows the behavior denoted in *Property 2*. For this particular case we exhibit the rule considering only messages for subscribing and activating filters: the filters must be activated accordingly to the time their subscription was received. Similar rules are defined for other kind of messages. The rule says that when two filter subscriptions are received in a server (*SUB-Filter0* first and *SUB-Filter1* afterwards) and the second one was activated (*ACT-Filter1*) then it must be the case that the first filter was activated before (*ACT-Filter1*).

Finally, rule in Figure 5 illustrates the behavior imposed by *Property 3*. This is a particular case of the previous property (*Property 2* in Figure 4). In this rules the following events are involved: *SUB-C0* (component *C0* subscribes a filter), *ACT-FC0* (the filter for component *C0* is activated), *PUBC1-E1* (component *C1* publishes event E1), *PUB-C1E2* (component C1 publishes event *E2*), *NOT-C0E1* (component *C0* receives the notification of event *E1*), *NOT-C0E2* (component *C0* receives the notification of event E1) and UNS-C0(component *C0* unsubscribes its filter). The rule demands that whenever two events are

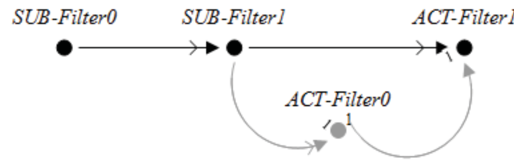

Fig. 4. Messages orders must be preserved

published by the same source then if the interested subscriber received the notification of the latter event and did not realize a unsubscription, then the subscriber received the notification of the former event before the occurrence of the notification of the latter.

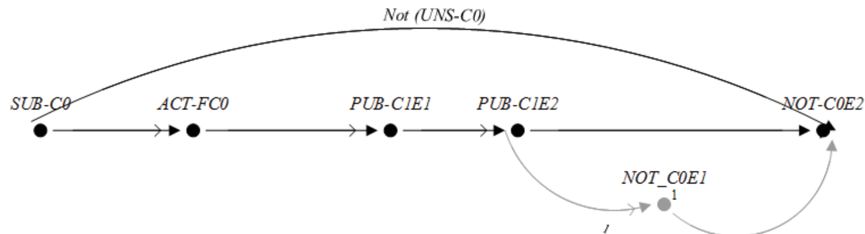

Fig. 5. Events generated by the same source respect the publication order

It is worth mentioning that all the rules were satisfied when employing the LTSA model checker to validate their behavior.

3.2 Verifying architectural conformance within Layered-based Systems

Inspired by the case of study introduced in [15] we show next how using *FVS's ghosts events* we can specify rules that verify that a system is satisfying the restrictions imposed by the layered pattern [8]. While services invocations are regular events present in the system's traces, the notion of layers lives in a higher level of abstraction. Given this context we introduce *ghost events* to capture the notion of layers. Events named *Layer-1*,*Layer-2*, ...,*Layer-N* represents the layers of the system. Once these events are defined, the user can predicate about architectural behavior based on these events. For example, the user can introduce rules that verify that services are only invoked from the immediate lower layer. The FVS rules in Figure 6 tackle the ghost layered events definition. For simplicity reasons, we show the definitions for a general schema with three services: *S-1*, *S-2* and *S-3* and three layers:*Layer-1*,*Layer-2* and *Layer-3*, where

VI

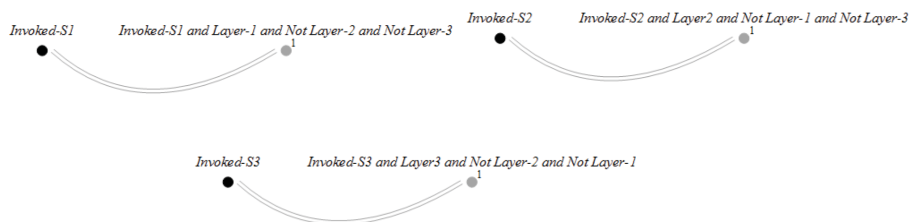

Fig. 6. Layered Ghost Events Definition

service *S-i* belongs to the layer *Layer-i*. In few words, each service invocation is associated with the corresponding ghost event.

The FVS rules in Figure 7 reflect the behavior restrictions imposed by the layered pattern: each service can only be invoked by services in the immediate lower level. These rules are based only in the ghost layered events.

Fig. 7. Rules reflecting the behavior of a layered pattern

We applied this approach to a layered system based on the example introduced in [15]. Roughly speaking, the system contains four layers that behave in the following way: the Web client and the application client call the session beans, the session beans invoke the entity beans, and the entity beans access the database tables on the back end. We defined the corresponding ghosts events and specified layered-restrictions with FVS rules (similar to the ones in Figures 6 and 7 instantiating the general schema to the concrete system), we obtained a model of the system and checked whether its satisfied the given properties. Similar to the results exhibited in [15] we found violations within the layered structure, since sessions beans interacted directly with the database layer.

4 Lessons Learned and Conclusions

Based of the results obtained in the case studies we believe that the FVS language is suitable for specifying and validating architectural behavior. The operational flavour given by the translation of FVS rules into automata made possible the integration with model checkers, which in turn, enabled the possibility of realizing architectural model checking.

The flexility and expressiveness of the language do have an important impact when modeling and exploring architectural behavior. The properties specified for the *SIENA* system were expressed in [10, 9] using temporal logic formulas although their approach uses another notation (a graphical language called

Property Sequence Charts). In addition, when specifying the formulas some auxiliary predicates were defined in order to simplify the specification. This might indicate that the comprehension of the formula may be a challenging task. This could become even more difficult if the property needs to be modified to adapt to a different context or compared against other possible formula describing an alternative solution. These activities are crucial since exploration and reasoning about behavior is fundamental when defining architectural behavior and interactions between the components of the system.

Consider, for example, that a more relaxed version of the *SIENA* system is considered, where messages can be processed by servers in any order, and not necessarily respecting the time they were received. The architect would then need to modify the requirements described in Section 3.1. For the FVS specification described by rules in Figure 4 only one modification is needed: the elimination of the precedence relationship between events *ACT-FILTER0* and *ACT-FILTER1*, which states that *ACT-FILTER0* must precede *ACT-FILTER1*. If this relationship is removed, then the rule simply state that both filters should be activated, but it does not impose any order of occurrence. The new version of the rule is shown in Figure 8. In the case of a temporal logic specification, the architect of the system might be faced against a complex formula whose modification is far from being trivial. The same analysis can be obtained when employing another notations such as structured ADL's or automata-based notations.

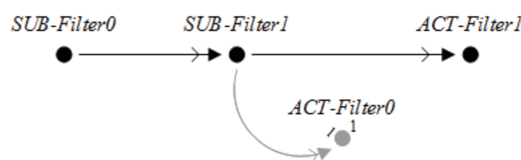

Fig. 8. A new rule modeling an architectural change in the system

The case study shown in Section 3.2 exhibits the benefits given by the expressive power of FVS. The use of ghost events makes a FVS a powerful language since ω-regular properties can be expressed, which is a distinguishable feature among declarative notations. In this case, ghost events are introduced so that the architect of the system can predicate in a higher level of abstraction and indicate interactions restrictions imposed by the layered architectural style.

Although the positive observations described in the previous paragraphs some threats to validity should be mentioned. First of all, a well designed experiment is needed to conclude with more precision about the flexibility and expressive power of our notation compared with other notations. FVS was compared against oher notations regarding this issue in [4, 3] in the context of expressing properties describing early behavior, showing that is more suitable for expressing behavior and its validation. However, a new experiment is needed to validate this aspects in the architectural domain. This experiment is beyond the scope of this paper

and is addressed as future work. The development of the case studies is another threat to be considered. In all the cases we replicated experiments shown in the literature, perhaps easing the specification and analyses process. Therefore, new case studies from scratch are needed to further validate the obtained results. This clearly represent a challenging issue regarding feature work.

5 Related and Future Work

The first step regarding future work is to validate the results presented in this work with case studies with industrial relevance from scratch. This is a logical continuation of the experiments shown in this paper. We also would like to interact with architectural conformance between a system's implementation with respect to its architecture specification. This would involve to combine our approach with others techniques focused on extracting, either statically or dynamically, the architecture from the source code [21, 2]. Finally, we are interested in performing an experiment to measure and compare flexibility and expressiveness of FVS with other notations. A good starting point could be considering the comparison of formal architectural approaches made in [23].

Related work can be divided into approaches denoting graphical languages and approaches employing ADLs (Architectural Description Languages) based on operational or source-code flavoured notations. Among the first group, probably the most representative approach is Property Sequence Chart (PSC) [7]. PSC is a graphical language inspired in UML 2.0 Interaction Sequence Diagrams that has been applied in the software architecture domain [18, 10, 9]. In PSC, denoting complex constraints between events may require textual annotations. In addition, properties in PCS are described as anti-scenarios and not as conditional or triggered scenarios. PSC is less expressive than FVS since it can only describe a subset of LTL, whereas FVS is more expressive than LTL. Work in [17] employs a graphical notation based on UML together with a textual notation focusing on a model driven approach. Our approach is focused on expressing behavioral properties to be model checked. FVS features only graphical scenarios and a more rich and flexible triggering mechanism, since for example, the antecedent need not to precede the consequents in time. Work in [22] also employs an hybrid notation combining UML like scenarios with additional textual description based on OCL constrains. This approach is mostly focused on modeling behavior rather than its specification and validation.

Other approaches rely on different ADLs (Architectural Description Languages) [16, 19, 13] based on operational, textual or source-code flavoured notations. We believe the graphical and declarative notation of FVS might make it more suitable for early behavior exploration [20]. Since FVS scenarios can be translated into Büchi automata [4] it would be interesting to investigate if FVS can be combined with others ADLs such as the mentioned ones or others.

References

1. S. T. Albin. *The art of software architecture: design methods and techniques*, volume 9. John Wiley & Sons, 2003.
2. J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *ICSE 2002*, pages 187–197. IEEE, 2002.
3. F. Asteasuain and V. Braberman. Specification patterns: formal and easy. *IJSEKE*, 25(04):669–700, 2015.
4. F. Asteasuain and V. Braberman. Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, pages Vol 22,239–274, 2017.
5. F. Asteasuain and F. Tarulla. Modelado de comportamiento de conectores de software a travs de lenguajes declarativos. In *CONAIISI*, 2016.
6. F. Asteasuain and L. P. Vultaggio. Declarative and flexible modeling of software product line architectures. *IEEE Latin America Transactions*, 14(2):885–892, 2016.
7. M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *SCESM '06*, pages 21–28. ACM, 2006.
8. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.
9. M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *ICSE 2004*.
10. M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of architectural patterns. In *EWSA*, pages 10–24. Springer, 2004.
11. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *TOCS*, 19(3):332–383, 2001.
12. P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002.
13. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In *WS-FM*, volume 4184, pages 193–213. Springer, 2006.
14. J. Magee and J. Kramer. *State models and java programs*. Wiley, 1999.
15. P. Merson. Using aspect-oriented programming to enforce architecture, software engineering institute. Technical report, CMU/SEI-2007-TN-019, 2007.
16. F. Oquendo, J. Leite, and T. Batista. Specifying architecture behavior with sysadl. In *WICSA 2016*, pages 140–145. IEEE, 2016.
17. F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. Archware: Architecting evolvable software. In *EWSA 2004*.
18. P. Pelliccione, P. Inverardi, and H. Muccini. Charmy: A framework for designing and verifying architectural specifications. *IEEE TSE*, 35(3):325–346, 2009.
19. T. K. Satyananda, D. Lee, and S. Kang. Formal verification of consistency between feature model and software architecture in software product line. In *ICSEA 2007*.
20. A. Van Lamsweerde. From system goals to software architecture. *Formal Methods for Software Architectures*, pages 25–43, 2003.
21. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems. In *ICSE*, aosfpp 470-479, 2004.
22. U. Zdun and P. Avgeriou. Modeling architectural patterns using architectural primitives. In *ACM SIGPLAN Notices*, volume 40, pages 133–146. ACM, 2005.
23. P. Zhang, H. Muccini, and B. Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723–744, 2010.