Compositional Verification of Events and Observers (Summary)

Cynthia Disenfeld and Shmuel Katz
Department of Computer Science
Technion - Israel Institute of Technology
{cdisenfe,katz}@cs.technion.ac.il

ABSTRACT

By distinguishing between events and aspects, it is possible to separate the problem of identifying when an aspect should be applied, from what it must do. Observers (aspects that do not affect the state of the base system) are already part of aspect-oriented programming and language support is emerging for events that gather information and announce occurrence. The goal of compositional verification of events and observers is to prove that they are correct so that their guarantees may be used by other events or aspects. Moreover, a compositional verification model allows applying formal verification techniques in smaller models, and also building a library of events, in which for any base system that satisfies certain assumptions, the event detection will satisfy its guarantees. In this work compositional verification of events and observers will be defined to aid in the design of a framework that allows users to verify events, providing as well flexibility in the input language of the specification.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, Model checking*

General Terms

Languages, Verification

Keywords

Events, Observer Aspects, Verification, Composition

1. INTRODUCTION

The goals of this work are to (1) precisely identify the components involved in the verification of events, (2) provide a methodological way to specify and verify events compositionally and (3) outline a framework design in which the input to the verification process that the user must provide is clearly defined, as are the steps performed automatically to verify events. The full version of this summary is available from the authors.

Aspect oriented programming (AOP) [13] allows expressing crosscutting concerns to the application in a modular way. AspectJ [12] defines a set of possible *joinpoints* - states where an aspect should be applied. For each aspect, *pointcuts* define where the response should be applied, and *advices* define what must be done.

However, AspectJ does not provide an optimal notation for a variety of problems. Most pointcuts in AspectJ can only see the present state in the execution and the current call stack. This does not give enough flexibility to be able to aggregate the history of events that have occurred. The second problem is the difficulty to share information between events: pointcuts only expose information on the target class, the arguments and the current aspect being executed. The third problem is that pointcuts are defined by means of events in the code, and sometimes we may be interested in expressing matching joinpoints in a more abstract way, for instance by defining events that occur as a result of the composition of other events.

[1, 17, 4] deal with the first problem by using a restriction of the language of aspects to regular expressions, or treating sequences of events but still the composition of lower level events and independence between the joinpoint and the response are not treated.

Douence *et al.* [6] present a solution for these problems by allowing to share variables between *crosscuts* (pointcuts), preserving the history of execution and defining composition between aspects. However the crosscuts are still tightly related to the *inserts* (advices), and this restricts reusability.

The separation of events has been presented already in the event-based approaches of [8, 14], independently of aspects.

[15, 11, 5] have identified the need of defining event aspects, spectators or observers that gather information but do not change the base system, although those aspects are allowed to print values.

Bockisch *et al.* [3] introduce a solution to these problems by syntactically distinguishing between *events* and aspects. Event declarations may accumulate information and do not affect the underlying system in any way, including printing values. They indicate when a certain concern should be woven and provide collected information of the system to be used by other event declarations or aspects.

Thus, the idea of defining aspects or events that collect information and are triggered when the collected informa-

FOAL'11, March 21, 2011, Pernambuco, Brazil.

tion satisfies a certain property is natural in systems. We will focus here on showing how to verify the correctness of event declarations, that use other events in their declaration. These definitions are also useful for existing systems already defined using observers and spectators even though the notation presented for events in [3] will be used here.

Events and observer aspects may seem trivial due to their spectative behavior on the base system. However, events incorporate the logic of when they must be triggered, and what information is exposed. They (and observer aspects) collect information from different possible sequences in the base system. This information may be collected from actions on the base system, and even subjected to some internal processing. Given that in the extension presented in [3] general aspects now respond to states in which events are detected, for later aspect verification it will be essential to assume that events are detected in the correct states and that they expose the expected information.

In this work we will focus on model checking techniques. However, present methods are "flat" in that they relate to aspects that directly are woven to a base system without event dependencies defined hierarchically. Here we emphasize the incorporation of assumed properties on used events in the verification of an event. Moreover, standard use of temporal logic assertions is problematic for event specifications. Thus, the framework presented gives enough liberty to choose among different modeling languages for the specification: regular expressions, Kripke models, Moore machines or temporal logic formulas.

The verification of events will be presented using the assume-guarantee model, which allows building a reference library of available events already verified. Hence, for any base system and set of events which satisfy the assumptions of needed events from the library, the library events may be included and their guarantees will hold without further verification.

1.1 Outline

This work is organized as follows: Section 2 presents background on models and simulations. Section 3 presents the definition of events and the necessary assumptions. In section 4 the verification steps are defined. Lastly, in Section 5 the conclusions are presented.

2. BACKGROUND

To be able to give a formal model of the specification and apply formal verification, the definitions of structures, homomorphism, preorder in structures, and Moore Machines presented in [10] will be used. The concepts of LTL, fairness and reductions presented in [7] will be used as well.

2.1 Kripke model for a Moore machine

In [10], a procedure was presented to obtain the corresponding Kripke structure of a given Moore machine. This structure contains the Cartesian product of each state with all the transition labels that may lead to another state. In our case we use a different construction, in which each state contains the transition labels that caused arriving to it. The formal definition is presented in the full version of this summary.

2.2 Model restriction

For a model $M = \langle S, S_0, R, L, \mathcal{F} \rangle$ given by a Kripke model over the set of atomic propositions AP, the operator $M \restriction AP'$ represents the restriction of the model to the atomic propositions in AP'. That is, for all $s \in S$, $L'(s) = L(s) \cap AP'$.

3. EVENTS

Events collect information on the base system, are triggered when an interesting situation to be detected occurs, and may expose certain information.

3.1 Assumptions

The following is assumed:

- Event invocation and execution do not affect any variable external to the event declaration, and they also do not affect the control flow (they must return the execution flow to the base system at the point they were begun).
- Event internal fields are only updated within the event declaration execution.
- There are no cycles in the event dependencies, i.e., an event cannot depend on itself being triggered in the correct places or its own exposed information being correct.
- Fairness restrictions must satisfy that it is always possible to get to a returning state for each event evaluation.

The first two properties may be checked by applying static analysis tools, adapting the tools presented in [2, 5, 16, 18] which work for identifying spectative or observer aspects. The dependency between events define an Event Dependency Graph [3], and cycle dependencies can be checked by analyzing this graph. Event models should guarantee that the only fair paths are those that eventually reach the end of execution.

From now on, the previously mentioned properties are assumed to hold.

3.2 Event Model

Each event contains a set of internal atomic propositions corresponding to the values of the internal fields, a set of external atomic propositions representing the parameters obtained from the lower level events, the initial values for the internal atomic propositions, and which *basic units* form the event.

Each *basic unit* u is a pair of a condition (consisting of other events having been triggered, pointcuts or predicates over the atomic propositions) and an event response that may only change the internal atomic propositions, or may trigger the event.

For the variables, fields, and parameters exposed, standard encoding and abstracting mechanisms are used, for example, range of values, boolean variables, etc.

We will use a high-level-syntax event example. There are well known translations from this language to the model form. The fragment of code presented in Figure 1 serves as an example of an event defined in terms of another event. There are three event declarations: (1) commit, (2) $TwoCommits$ is an event detected every two times *commit* is applied, and (3) $SixCommits$ is an event detected every six times *commit* is applied (but is defined indirectly using $TwoCommits$).

```
event commit(): call(* *.commit(..));
event TwoCommits()
   int counter=0;
   when(): commit()
      counter++;
      if (counter mod 2 == 0)
         trigger();
         counter = 0;

event SixCommits()
   int counter=0;
   when(): TwoCommits()
      counter++;
      if (counter mod 3 == 0)
         trigger();
         counter=0;
```

Figure 1: *SixCommits*

4. VERIFICATION PROCESS

4.1 Event evaluation Semantics

To be able to introduce the event evaluation semantics, the operator $\circledast$ is formally defined in the full version of this article. This operator represents the evaluation of the events in a set of events E in a base system or base system assumption B.

In this semantics, all the events are evaluated immediately, adding which events are detected to the atomic propositions of each possible state in the base system.

The base system regards events as being evaluated all at once, and in parallel due to their spectative nature. This differs from the weaving of aspects into a base system as presented in [9], where every state in the execution of the aspect is added to the woven model. Aspects are not instantly evaluated as their execution is not necessarily spectative and the state of the base system may change.

Note that in Figure 1 the basic units in the events take several steps to execute. However, in the resulting semantics event execution may be seen as immediate relative to the base system because of the locality of the fields, that no event affects the internal propositions of other events, and that there are no dependencies cycles.

Events and observers detect interesting situations in the base system, or collect information for certain paths of execution without affecting the state of the base system. This is one of the more useful advantages in restricting verification to events and observers rather than general aspects. Even though the size of the model grows due to the possible internal states of the events, the execution of all the events involved does not have to be represented at once, but only the resulting internal states and detected events.

From now on, $B \circledast E$ represents the base system with the detection of all events in E. U will also serve to denote a set of event declarations.

4.2 Specification

4.2.1 General idea

To prove a guarantee about an event E, E needs to assume the correctness of the guarantees of events it uses (and the same is true for a general aspect A).

The properties that an event must be proved to satisfy may be categorized as:

1. The event is triggered in the correct places. This requires defining exactly which sequence of situations and contexts in the base system and previously verified events should cause the current event to be triggered. The specification is in terms of event detections, exposed parameters and may as well include auxiliary variables.
2. The parameters exposed by the event satisfy the intended relations with the history of execution.

4.2.2 Specification definition

An event's specification, $\langle Ass, Guar \rangle$ - representing the assume and the guarantee - may be given in different specification languages. If the specification is given as a regular expression, then the equivalent automaton is obtained and it can be understood as a Moore Machine. If any of them is given as a Moore machine, then the equivalent Kripke model is built. If any of them is given as a CTL or LTL formula, then its tableau is built.

In particular the specification of $TwoCommits$ may be expressed as $\langle Ass, Guar \rangle$ where $Ass \equiv \neg commit$ (*commit* is false in the initial state) and $Guar$ is given in Figure 2. This guarantee represents that every two occurrences of *commit*, *twocommits* hold.

The guarantee of the event $TwoCommits$ should be the assumption of $SixCommits$. Then, $SixCommits$' specification is given by $\langle Guar_{TwoCommits}, Guar_{SixCommits} \rangle$.

The guarantee of $SixCommits$ can be expressed as a Moore machine, as in Figure 4 or as its equivalent Kripke model presented in Figure 5.

When the event is intended to occur dependent on the occurrence of other events, either after a sequence of other events or the lack of events, the preferred specification is as a regular expression of events or in state machines, where both the specification and the event are given in that form. Aspect specifications usually refer to what properties each state must satisfy. Events, on the other hand, do not modify the state of the base system and are specified by means of what sequences of triggered event lead to them being detected, and what properties their exposed parameters must satisfy. Hence, aspects usually satisfy properties given in temporal logic, over the atomic propositions that represent the state, and events preserve the state so they refer to sequences of states instead. For every possible sequence it must be described whether it leads (or not) to the event to be detected and which information is to be exposed. Temporal logic expressions can be used to specify events, but become awkward and unreadable very quickly when sequences of lower level events must be expressed. Therefore we prefer regular expressions or state machines.

Specifying a property that the parameters satisfy when the event is detected may be defined by means of any of the languages presented for specification and a similar verification process can be applied.

4.3 Verification

Event verification consists in checking whether: given the assumption Ass of E on the base system and used event detectors, when detecting the event E, the guarantee $Guar$ is satisfied (expressing both when E is detected and what must hold for parameters it then exposes).

In more formal notation, the goal in event verification is to prove that $B \circledast E \vDash Guar$. This is, the base system together with the event detection satisfies its specification.

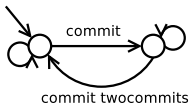

Figure 2: Moore machine: $Guar_{TwoCommits}$

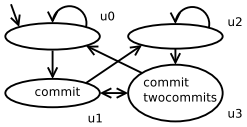

Figure 3: Kripke model: $Guar_{TwoCommits}$

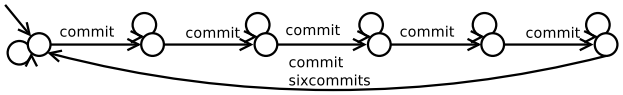

Figure 4: Moore machine: $Guar_{SixCommits}$

However, in order to obtain a modular verification of events, the assume guarantee model is used and the specification will be given by $\langle Ass, Guar\rangle$. The goal will be to prove that if Ass is the assumption about the base system and used event detectors, and $Ass \circledast E \vDash Guar$, then for any base system B and set of used event declarations U such that $B \circledast U \vDash Ass$, it can be inferred that $B \circledast U \circledast E \vDash Guar$, this is, the base system with all the event detectors incorporated satisfies its guarantee. At this step, verification of events is presented for a base system which has no aspects that may affect the event woven into it. Verification of events together with aspects woven will be analyzed in future work.

In particular, for the correctness in the detection of the event, if there exists an assumption Ass_B on the base system such that:

$$\begin{aligned} B &\preceq Ass_B \\ Ass_B \circledast U \upharpoonright AP_{Ass} &\equiv Ass \quad (1) \\ Ass \circledast E \upharpoonright AP_{Guar} &\equiv Guar \quad (2) \end{aligned}$$

Then:

$$\begin{aligned} Ass_B \circledast U \circledast E \upharpoonright AP_{Guar} &\equiv Guar \\ B \circledast U \circledast E &\preceq Guar \end{aligned}$$

The previous inference expresses that:

- Given a base system that satisfies a certain assumption Ass_B
- Given that the composition of this assumption with the set of used event detectors is equivalent to the assumption of the event, and

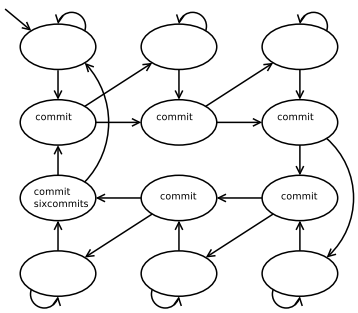

Figure 5: Kripke model: $Guar_{SixCommits}$

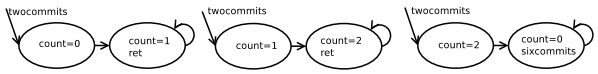

Figure 6: $SixCommits$

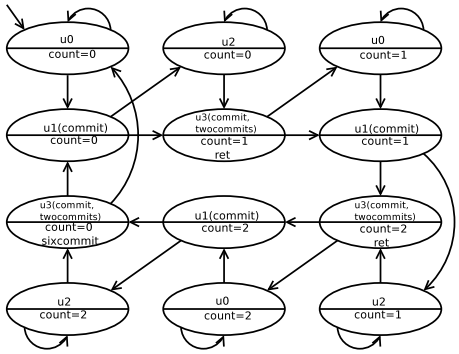

Figure 7: $Guar_{TwoCommits} \circledast SixCommits$

- Given that an event E is proven to be correct with respect to its specification

Then, the composition of the base system, used event detectors and E satisfies the guarantee $Guar$.

In the previous inference, (1) may be proven using the assume-guarantee model as well. In (1) and (2), proving the bisimulation guarantees that there exist paths in the model on the left side of the equation that behave as presented in the guarantee. Ass, $Guar$ and E must be given by the user so as to verify that E satisfies its specification $\langle Ass, Guar\rangle$.

Considering the example, a base system B will be composed with $TwoCommits$ and $SixCommits$. The first step is to prove $TwoCommits$ is correct. When $\neg commit$ is initially true, the tableau of the assumption together with the model of the code of $TwoCommits$ can be proven to satisfy Figure 3. Turning to $SixCommits$, when its assumption holds, the model of the code of $SixCommits$ can be proven to satisfy its guarantee as presented below.

The event $SixCommits$ - with $AP_0 = \{count = 0\}$ contains only one basic unit modeled in Figure 6. The event is evaluated for each occurrence of $twocommits$. Every three occurrences of $twocommits$, $sixcommits$ is triggered.

The event evaluation of $SixCommits$ is given by:

$S = \{u_0, u_1, u_2, u_3\} \times 2^{AP_{SC}}$ where $\{u_0, u_1, u_2, u_3\}$ are the states from Figure 3 and $AP_{SC} = \{count = 0, count = 1, count = 2, ret, sixcommits\}$. $S_0 = \{(u_0, count = 0)\}$

Considering the definition of the relation between the states and the event, the model in Figure 7 is obtained.

Restricting the model of $Guar_{TwoCommits} \circledast SixCommits$ to the atomic propositions of the specification, it is exactly the same model as given by the specification.

Therefore, effectively there is a homomorphism such that $Guar_{TwoCommits} \circledast SixCommits \equiv Guar_{SixCommits}$.

The previous procedure shows the correctness of $SixCommits$ with respect to its specification. Now, for any base system B that satisfies $\neg commit$, $B \circledast TwoCommits$ is correct. Moreover, for any implementation of $TwoCommits$ that satisfies the guarantee presented in Figure 3, incorporating the evaluation of $SixCommits$ will satisfy $Guar_{SixCommits}$. Consequently, it can be inferred that for any base system B

such that *commit* does not occur in the initial state, and for the models of the code presented for $TwoCommits$ and $SixCommits$:

$$B \circledast TwoCommits \circledast SixCommits \vDash Guar_{SixCommits}$$

Note how the guarantee of a simpler event is used as an assumption of one that uses it, and is incorporated into the verification.

The previous verification may include information on what values are exposed by the parameters. For the correctness of the exposed parameters a simulation is enough to prove the correctness and conclude that $B \circledast U \circledast E \preceq Guar$.

The process of finding a homomorphism for $\preceq$ can be done automatically by the algorithm presented in [10]. There are also automatic methods for obtaining the structure equivalent to a Moore machine or the tableau of a formula. The only things remaining to the user to provide are the specification and the model of the event expected to be verified.

5. CONCLUSIONS

Given the need to separate and abstract *when* an aspect is applied from *what* aspects do, *events* were incorporated in [3] aiming to identify *when* things should happen, and being able to collect information, or be detected when particular sequences of other events occurrences. Moreover, observers and spectator aspects [5, 11] are part of current programming practices in aspect-oriented programming. Due to their spectative nature, events and observers may seem trivial to be verified for correctness. However, other events and aspects may use the information and detection of the event, hence events must be verified to be correct when they are triggered, and the information exposed must satisfy requirements that other entities depend on.

Events may be thought of as spectative aspects with the additional *triggering* action, the use of other events as conditions, and restricted not to have output. This guarantees non-interference between the events. No event can affect another event - except by triggering itself, and none of them may modify the state in the base system.

In this work a modular verification method for events is introduced where the user is requested to present the assumptions, the expected guarantees and the event itself to be verified. Without additional intervention of the user the property is verified relative to the specifications of the used events. The use of the guarantees of events as assumptions for other events is shown as well. For all the steps presented there are existing tools that perform the necessary algorithms.

The users may follow the procedures presented in the article to define their own specifications or use their own methods, using for example regular expressions or state machines. Expressing sequences of events in a temporal logic formula gets hard to read very easily.

In the full version of this summary (as noted, available from the authors), a fuller semantic notation is presented, as well as an additional example with *Guar* on the parameters, and detection involving the absence of other events. In future work, the influence of events on aspects will be analyzed, to give a complete hierarchical and compositional formal verification algorithm for systems that include both events and aspects. In this framework, as opposed to [9], the assumptions about other events can be incorporated naturally.

6. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. *SIGPLAN Not.*, 40:345–364, 2005.

[2] Y. Alperin-Tsimerman and S. Katz. Dataflow analysis for properties of aspect systems. In *Proceedings of 5th Haifa Verification Conference, LNCS 6405*, 2009.

[3] C. Bockisch, S. Malakuti, M. Aksit, and S. Katz. Making aspects natural - events and composition. In *AOSD 2011 Modularity Visions Track*, 2011.

[4] E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In *Software Composition*, 2006.

[5] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical report, Iowa State University, Department of Computer Science, 2002.

[6] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD 2004*, 2004.

[7] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[8] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Press, 2010.

[9] M. Goldman, E. Katz, and S. Katz. Maven: modular aspect verification and interference analysis. *Formal Methods in System Design*, 37:61–92, 2010.

[10] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16:843–871, 1994.

[11] S. Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development I*, LNCS 3880:106–134, 2006.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP*, 2001.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

[14] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[15] O. Mishali and S. Katz. The highspectj framework. In *Proc. of the 8th workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2009.

[16] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. of the 12th ACM SIGSOFT Symp. on Foundations of Software Engineering*, 2004.

[17] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. D. Fraine. Stateful aspects in jasco. In *Software Composition*, 2005.

[18] N. Weston, F. Taiani, and A. Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. MeMoT'07, 2007.