Compositional Verification of Events and Aspects

Cynthia Disenfeld *

Department of Computer Science
Technion - Israel Institute of Technology
cdisenfe@cs.technion.ac.il

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]; D.2.13 [*Reusable Software*]: Reuse models

General Terms Languages, Verification

Keywords Aspects, Events, Compositional Verification, Interference, Abstraction, Refinement

1. Introduction

In reactive systems, events are detected and responses (reactions to the detected events) are activated. These events can be given by a user, software, hardware or by a more complex processing of lower-level events. Complex Event Processing (CEP) [13, 18] is a paradigm that focuses on the detection, processing, distribution and consumption of events.

Crosscutting concerns are ubiquitous in software systems. Examples of typical crosscutting concerns are logging, persistence, security, transaction management and others. Aspect-Oriented Programming (AOP) [17] provides a modular representation of crosscutting concerns where the developer indicates where should these concerns be applied and what should be done. This way, scattering and tangling [24] of concerns is avoided. Aspects are applied at several different locations of the underlying system. Identifying these locations with event detectors is a very natural approach to combine events and aspects.

My research deals with understanding the implications of introducing language extensions to handle events and developing practical tools and techniques for compositional verification in this context.

Moreover, we consider some of the semantic choices in combining events and aspects, their motivation and implications. In particular, these choices affect the understanding of the system. For instance, certain combinations of these semantic choices lead to a possible atomic understanding of events which makes reasoning about them much easier.

Furthermore, we provide a compositional verification technique for modular verification of libraries of events and aspects [12]. Using this technique, model checking is applied to much smaller models, and for the user who is unaware of what the aspect assumes about the underlying event detectors, the necessary assumptions are obtained automatically.

* ACM Membership number: 8906706
Supervisor: Prof. Shmuel Katz

2. Background

In this section we provide the background and related work on event and aspect verification in order to later expand on our compositional specification and verification technique of aspects that respond to complex events.

2.1 Aspect-Oriented Programming (AOP)

As mentioned above, in AOP, crosscutting concerns can be represented modularly. Each aspect consists of a set of advice procedures, where each advice indicates where it must be applied and what it must execute. *Pointcuts* are expressions used to indicate where the aspect should be activated. They usually have keywords to represent method calls, fields set, exceptions thrown and others. Each pointcut captures a set of *joinpoints* (locations where an aspect may be woven). Each AOP language defines its own joinpoint model, that is, the set of all the locations where an aspect may be activated.

A very popular AOP language is AspectJ [17] which extends Java to include aspects. In AspectJ, the joinpoint model does not capture loops or assignments, but does capture some other dynamic information such as the current control flow, the actual class of an argument, calls, exceptions, exception handling and others. However, the fragile pointcut problem [23] (an unexpected joinpoint could be added or removed to those captured by the pointcut without noticing) is a well-known issue in AspectJ, since pointcuts are very attached to the syntactic names of methods and fields used.

2.2 Events

Combining events with AOP not only aids in reducing the fragile pointcut problem but also provides a clearer way to represent the locations where an aspect should be woven. The combination of events and aspects has been considered in several related work [1, 14, 20, 22]. However, each of these extensions either has deficient decoupling properties, or is restricted in its expressive power, its compositionality, or its capability to represent crosscutting events in the context of AOP.

The work in [4] considers a more natural integration of CEP and AOP: event detectors look like aspect definitions but they are only allowed to observe the underlying system. Their side-effects are only those given by the possibly activated responses. In addition, event detectors can gather information and be hierarchically composed, thus building more complex event detectors. Aspects can then respond to complex event detectors. In the present work we consider this event definition, however the ideas presented for compositional specification and verification of events and aspects can be applied to different event languages as long as the observer-detector essence of events is preserved.

2.3 Event Specification and Verification

The issue of event specification and verification was addressed as a first step of this work in [10]. In particular, different specification languages such as LTL (linear temporal logic) [21], automata, regular expressions and Kripke structures were analyzed. LTL allows expressing properties about the exposed information of events according to the events that have occurred before. Other specification languages such as state machines allow expressing properties by means of paths or words accepted, and are particularly useful for defining exactly when the event must or must not be detected.

Moreover, the main general properties that need to be verified about events were presented. A complete guarantee of an event *must* include:

When is the event detected?: More complex event detectors and aspects that respond to these events rely on the considered event being detected at the exactly right places.

What information is exposed?: More complex event detectors and aspects may also rely on the information exposed by lower-level events, therefore the guarantee should express the correctness of the exposed information.

Example 1. *The event detector in Fig. 1 represents when there have been insufficient purchases of a product during a certain period. In this definition, whenever there is a relevant purchase (indicated by the lower-level event detector $RelevantPurchase$), the counter in $purchaseInfo$ is increased. Whenever there is a call to $timeDone$ (a period of consideration has ended) and there number of relevant purchases of the current period is not enough, the $LowActivity$ event is triggered, exposing the product for which it is detected.*

```
event LowActivity(P product)
 int UPPER_BOUND = 100;
 Info purchaseInfo = new Info();
 after(Purchase purchase):RelevantPurchase(purchase){
   purchaseInfo.increase(purchase.product());
 }
 when(P product):call(P.timeDone()) && target(product){
  if (purchaseInfo.count(product) <= UPPER_BOUND)
   trigger(product, purchaseInfo.count(product));
  purchaseInfo.reset(product);
 }
```

Figure 1. `LowActivity` event

Example 2. *Continuing with the example of Fig. 1, the guarantee should express that the event is detected if and only if a period without enough relevant purchases has ended, and the product exposed is the one for which the event detected.*

To apply verification we have presented a compositional technique where each event should satisfy a specification given by an assumption about the underlying system and a guarantee about the augmented system. This model - consisting of an assumption and a guarantee - is called *assume-guarantee*. More details are available in [10].

2.4 Aspect Specification and Verification

Aspect specification consists of an assumption about the underlying system and a guarantee about the augmented system expressed in Linear Temporal Logic (LTL). By applying an *assume-guarantee* verification technique, each aspect is verified on its own and on success the aspect can be applied to any system satisfying its assumption [15]. Modular verification for aspects is applied as follows:

1. The tableau of the assumption is built. The tableau of an LTL formula is a state machine including every possible computation path satisfying that formula.
2. The aspect is woven. At every place where the aspect should be woven a transition from the tableau to the aspect is added, and at every place where the aspect finishes its execution a transition is added from the aspect to the tableau.
3. The augmented model is checked against the guarantee using model checking.

Then, given a system S satisfying the aspect assumption, the augmented model including S and the aspect will certainly satisfy the aspect guarantee.

Example 3. *We can consider as an example the $Discount$ aspect in Fig. 2. This aspect applies a discount to the product for which $LowActivity$ has been detected. A possible assumption of $Discount$ would be that $LowActivity$ is detected at the right places, and a possible guarantee would be*

```
aspect Discount
  after(Product p): LowActivity(p)
    applyDiscount(p);
```

Figure 2. `Discount` aspect

that whenever $LowActivity$ is detected, in future purchases including the product, the discount will be applied.

2.4.1 Interference

Systems usually include several aspects, and there may be interference when a group of them is woven to the same system. There is interference among a set of aspects if each aspect on its own is correct with respect to its specification, but when the whole system is considered, at least one guarantee is no longer satisfied. Interference under sequential-weaving (only one aspect is woven at a time) was considered in [15]. However, in that model, if an aspect A is woven into a system at the available joinpoints, and then another aspect B is woven, the joinpoints of A added by B are not recognized (since B was added *only* after A has already been woven).

As part of this research, we have also considered the problem of interference and cooperation among aspects under joint-weaving semantics [11], introducing the specification and verification techniques in order to detect interference or verify the correctness of a set of (possibly collaborative) aspects. However, in that work, aspects are assumed to respond to LTL pointcuts but more complex events gathering information are not considered.

2.5 Counterexample-Guided Abstraction Refinement

In model checking, the state-explosion problem is widely known for making very large model verification unfeasible. Counterexample-Guided Abstraction Refinement (CEGAR) [8] is an iterative technique that addresses this problem by starting verification on an abstract model and applying the necessary refinements only when necessary.

The abstract model considered at each iteration represents an overapproximation of the concrete model. Therefore, if a property expressing what is expected of every path is satisfied in the abstract model, in particular it is satisfied in the concrete model. However, if the property is not satisfied in the abstract model, the counterexample obtained is analyzed to determine whether the error is real (i.e. it belongs to the concrete model) or spurious (belongs only to the abstract model).

The procedure is applied until the error is found real, or all the necessary refinements have been added to prove the property satisfied.

CEGAR has be combined with various techniques. Some examples are:

- The original work [8] uses BDDs [5] for spuriousness checking and refining.
- BLAST [3] obtains a sequence of interpolants [19] and applies lazy abstraction [16].
- SATABS [9] translates the program to a boolean program and interacts with a SAT Solver for verification and refinement.
- MAGIC [6] represents concurrent components with labelled transition systems (LTS) and compositionally verifies and finds refinements (using weak bisimulation).

Although all the related work applies CEGAR ideas, the use of a counterexample-guided abstraction refinement scheme has not yet been considered for events and aspects. In particular, the knowledge about the interaction between events and aspects can be used to build a compositional technique (to be presented in the next sections).

3. Problem formulation and Approach

The combination of event and aspect verification raises new questions such as how the aspect verification technique is affected when aspects respond to events detected. Aspect verification in MAVEN [15] considers pointcuts as temporal logic formulas. Given that events are now more complex and hierarchical entities, event specification should now be used to identify the places where the aspects should be activated.

The method obtained should be sound and compositional, in order to allow considering aspects responding to different events, and reusing the proofs in different systems to which the library is applied.

Considering event guarantees given by LTL formulas, a naive approach where all the event guarantees are considered as part of the assumption of the aspect would yield a very large model. Asking the user to provide a minimal set of assumptions is not feasible since the user may not necessarily know which information about the event detectors is in fact necessary for proving a property.

3.1 CEGAR for Event and Aspect Verification

To address these problems, an abstraction-refinement scheme for verifying events and aspects has been presented [12]. The main algorithm is presented in Fig. 3. In this scheme, we start with the specification of the aspect and underlying events. The aspect assumption about the underlying events should be an overapproximation of the actual event detection. This initial assumption can be obtained, for instance, with static code analysis. In the example of Fig. 1 a possible initial assumption about the events could be that whenever $LowActivity$ is detected, for sure a call to $timeDone$ must have occurred.

Verification is applied as in MAVEN [15] (Section 2.4). There are two possible outcomes: either verification succeeds or a counterexample is found. If verification succeeds with the overapproximation considered, then in particular for the actual event detection definition the aspect guarantee holds, and therefore the aspect is proven correct. Otherwise,

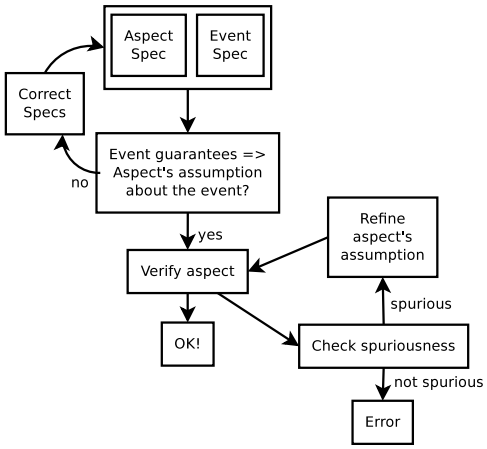

Figure 3. Aspect verification

the counterexample is analyzed to check whether the error is real (the counterexample found is consistent with all event definitions and thus the error is in the aspect or its specification) or whether the error is spurious (some information about the event detectors is missing). This analysis can be done automatically applying formal verification techniques [12] and in case the error is found spurious an appropriate refinement is added.

This abstraction-refinement technique in the context of AOP with events has several advantages:

- The user does not need to know what the aspect assumes about the event detectors for its guarantee to hold.
- Each aspect guarantee does not usually need all the information about the underlying event detectors. Therefore, model checking is applied to much smaller models.
- Checking spuriousness and finding the appropriate refinement is done modularly. Therefore, if the number of event detectors increases, verification should still be feasible.
- By the end of the procedure either a real counterexample is found or the property is verified and the necessary information about the events is obtained.

3.2 Interference

Another question that arises is how to apply compositional interference detection now that aspects may be advised by other aspects because of events detected during their execution. The work already done as part of this research on interference detection [11] has been extended to incorporate event detectors [12]. This implied analyzing interference and cooperation that may arise from the combination of events and aspects and considering the spectative nature of events.

3.3 Event semantics

Different event semantic choices regarding event evaluation may affect the analysis of a library of events and aspects. For example:

1. In which order should event detectors be evaluated? A first analysis of two options was presented in [4]:
   - One possibility is to first evaluate all the event detectors, and then apply all the aspects for which their event detector is matched. In this case, reasoning may seem simpler (the changes applied by an aspect are ignored if another aspect was already decided to be activated because of the current set of events detected). However, this scheme is not consistent with the AspectJ's behavior (in AspectJ, an aspect is activated if and only if at the moment to be activated the dynamic conditions hold as well).
   - Another possibility is to consider at each joinpoint which are the possible aspects that may be activated, and for each of these aspects evaluate the necessary event detectors. In this case the semantics are closer to those of AspectJ and to what AOP developers are used to. However, when applying formal methods the possible changes made by other aspects should be considered.
2. Are events allowed to be detected within events? For example, could an event B be evaluated because of some statement within another event A? If so, what would happen with the aspects that are activated because of B? Restricting events not to be evaluated within the evaluation of other events allows a clearer understanding of the system, and maintains events' spectative essence. This restriction does not affect the expressive power of events greatly, it only implies that the developer should reify and modularize events in order to preserve their modularity and simplicity.
3. How does the underlying joinpoint model affect the evaluation locations? The set of locations where an event is potentially detected is tightly coupled to the underlying joinpoint model.

For the work in [12] we have not restricted the event model besides the natural assumptions of event evaluation being atomic (or reducible to atomic) and free of side-effects. Therefore, different semantic choices can be considered under which these assumptions are maintained.

4. Uniqueness and contributions

The approach to answer these questions involved combining very different formal verification techniques. Static analysis can be used for checking that an event does not affect the underlying system and in [12] was considered to obtain an initial assumption about the underlying events. Model checking is used to verify that an augmented model satisfies a prop-

erty and to analyze whether a counterexample is spurious. SAT solvers are used to find appropriate refinements. Taking advantage of different techniques' strengths aids in building a simple and natural technique for verifying a library of events and aspects.

Although there is several related work on CEGAR, our work is the first one considering CEGAR in the context of events and aspects. In particular, since events are assumed to be spectative, the technique presented is compositional, smaller models are checked and spuriousness checking and refinement is done by considering each event individually, thus improving scalability of AOP verification.

The technique presented also takes advantage of event and aspect modularity, and applies the necessary checks modularly. This is sound since the abstractions considered represent an overapproximation of the actual event detectors and therefore when an LTL property is checked for every path of the abstract model, in particular it is checked for every path of the concrete model.

In particular, CEGAR for events and aspects has been extended to [12] :

- detect interference in a library of events and aspects where aspects may respond to complex events,
- analyze certain cases of reachability (whether there exists some path where the event *may* be activated, and
- complete partial event specifications (user-interaction is required).

Verifying a library modularly prevents dependence on a completely built system in order to apply an early detection of bugs and interference. It also allows reusing the library in any other system that satisfies the necessary assumptions without applying any further checks.

The uniqueness of this research is mainly given by building a compositional verification technique that combines events and aspects, detects interference and is not restricted to sequential weaving semantics, but also accepts aspects that may add and remove events of other aspects as in the joint-weaving model.

The technique presented is also flexible to consider different semantic choices regarding event evaluation. The main assumptions are that event evaluation is considered atomic (or can be reduced to be thought as atomic) and does not affect the underlying system.

A partial implementation of the technique has been done extending MAVEN [15] to include spuriousness checking, finding appropriate refinements and applying verification iteratively by interacting with the model checker NuSMV [7] and an SMT Solver [2].

In future work, we intend to consider more complex case studies to analyze performance and scalability of the approach. Moreover, the user-interaction queries should be refined to find appropriate questions that help the user correct and complete specifications, thus providing a more user-friendly specification and verification tool.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[3] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5), 2007.

[4] Christoph Bockisch, Somayeh Malakuti, Mehmet Akşit, and Shmuel Katz. Making aspects natural: events and composition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 285–300. ACM, 2011.

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8), 1986.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Software Engineering, 2003. Proc. 25th Intl. Conf. on*, 2003.

[7] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 359–364. Springer-Verlag, 2002.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.

[9] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *In TACAS, volume 3440 of LNCS*. Springer, 2005.

[10] Cynthia Disenfeld and Shmuel Katz. Compositional verification of events and observers: (summary). In *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, FOAL '11, pages 1–5. ACM, 2011.

[11] Cynthia Disenfeld and Shmuel Katz. A closer look at aspect interference and cooperation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 107–118. ACM, 2012.

[12] Cynthia Disenfeld and Shmuel Katz. Specification and verification of event detectors and responses. In *Proceedings of the 12th annual international conference on Aspect-oriented Software Development*, AOSD '13. ACM, 2013.

[13] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Press, 2010.

[14] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. Escala: modular event-driven object interactions in scala. In *Proceedings of the tenth international*

conference on Aspect-oriented software development, AOSD '11, pages 227–240. ACM, 2011.

[15] Max Goldman, Emilia Katz, and Shmuel Katz. MAVEN: modular aspect verification and interference analysis. *Formal Methods in System Design*, 37(1):61–92, 2010.

[16] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. POPL '02. ACM, 2002.

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer-Verlag, 2001.

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

[19] K. L. Mcmillan. Interpolation and SAT-Based Model Checking Financial Cryptography. volume 2742 of *Lecture Notes in Computer Science*. Springer, 2003.

[20] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using awed. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 51–62. ACM, 2006.

[21] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57. IEEE Computer Society, 1977.

[22] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 155–179. Springer-Verlag, 2008.

[23] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, 2004.

[24] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119. ACM, 1999.