Specification and Verification of Event Detectors and Responses

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

Abstract

Events and aspects that respond to them can and should be defined, specified, and verified in a modular way, as an aid in understanding and guaranteeing the correctness of each on its own. However, finding the appropriate interfaces and abstractions and expressing them precisely is not an easy task. Moreover, formally verifying large models is often unfeasible for existing model-checking tools.

We present an abstraction-refinement scheme to verify aspects and to define and correct both aspect and event specifications. This allows considering smaller models and learning the needed event guarantees at each step. In addition, this technique can be used to find sound abstractions to check event reachability. Moreover, the technique is applied for detecting interference in systems where there are responses to complex events and aspects may be activated within the execution of other aspects.

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 Aspects, Events, Specification, Verification, Composition

1. Introduction

In reactive systems, events occur and responses are executed when necessary. Aspect-oriented programming (AOP)[17] represents crosscutting concerns with aspects, and responses with advice defined within the aspects. Advice is applied when its pointcut designator matches a joinpoint such as a method call, an exception thrown, and others. Complex Event Processing (CEP)[11, 18], a separate programming paradigm, analyzes the necessary processing of events in order to generate new events or notify event consumers such as services or business processes. Events can be filtered, transformed, aggregated in different time/location windows in order to generate new events. While one approach focuses on crosscutting concerns and the other on event processing, both capture main principles of reactive systems.

In AOP, pointcut designators as presented in AspectJ[16] and other aspect languages are very dependent on how the code was written and the names chosen for the different classes, methods, and fields. Moreover, if an aspect responds to some complex event, the internal fields to record needed information about the event and the processing to detect it are tangled with the response to its detection. Any small change to the underlying system may lead to the fragile pointcut problem [23], where previously captured joinpoints are no longer matched, or new unintended joinpoints are now matched. Moreover, pointcut composition is restricted to boolean operators or knowing if it is in the execution flow of another pointcut.

Some work [2, 4, 21] has considered extending pointcut expressive power to regular expressions or patterns. However pointcuts are still limited to the restricted expressive power of the language presented.

In [3], events were introduced in the context of aspect-oriented programming, distinguishing more clearly between when an aspect should be activated, and what it must do. A joinpoint is then an event occurrence, and pointcuts are low-level event detectors. More complex event detectors can be defined, by writing pieces of code that resemble advice and may trigger new events. Examples of events are: a commit request to a control version system repository without having the tests run and succeed, insufficient purchases of a certain product during a certain period, or identifying a series of messages as a potential denial-of service attack. Event evaluation and detection does not have any external side-effect besides those caused by the event being triggered. This allows applying optimizations and distinguishing clearly between when an event is being detected and what the responses to detection are that may affect the entire system.

Considering events as in [3] in the context of AOP aids in reducing the gap between systems considered in this paradigm and the CEP paradigm[11, 18] where events are announced by several event producers, these events are then processed (filtered, transformed, etc.) and different consumers can be notified so that they can apply the appropriate response. Events can be added with the syntactic sugar presented in [3] or using coding conventions with, e.g., AspectJ, to express event detectors as usual spectative aspects (also called observers [7, 15]). We will call the use of events within aspect-oriented programming, whether explicit or using coding conventions, *Event AOP*.

Previous work has considered how to express formal specification of events[9] and aspects[12, 14]. Such specification provides a better understanding of the system, and moreover, allows applying formal verification techniques based on model checking to guarantee the correctness of each modularly or detect interference among aspects[10, 12].

However, when applying verification to a library containing both events and aspects, new practical issues arise. In particular, aspect and event specifications are difficult to write precisely, as needed for formal verification. Especially, given the complexity of a hierarchy of events, where one depends on numerous others, and considering that model checking suffers from the state explosion problem, a sound, practical, and feasible technique must be used. In this work we present an abstraction-refinement technique, inspired by CEGAR[6] (Counterexample-Guided Abstraction Refinement) which allows checking smaller models, and only refining and considering bigger models when necessary. Basically, an abstraction of the model to be checked is used, and if a counterexample is found, then it is checked automatically to find whether the error is real or spurious (and due to overabstraction of the actual system). If the counterexample found represents a spurious error, the model is automatically refined and a new attempt is made to verify the model relative to its specification.

We adapt this idea to verify smaller models when applying aspect verification, exploiting our knowledge of the relationship between detecting events and responding to them in aspects in order to propose generic abstractions that often are adequate for verification, and provide a basis for refinement when they are not. Using the abstraction-refinement scheme to be presented, the task of writing specifications often becomes much simpler. The abstraction-refinement mechanism can be used for finding appropriate abstractions to write event and aspect specifications, and/or for correcting existing specifications. Moreover, we take advantage of these ideas both to find sound abstractions for verifying event reachability (i.e. determining whether an event may be eventually detected) and to analyze possible *interference* in a library including both events and aspects. Here *interference* means that on their own events and aspects are correct, but weaving them together may cause one or more of them not to achieve their expected behavior.

Therefore, the main contributions of this work are to:

- Introduce to aspect specification the aspects' assumption about underlying events.
- Present an abstraction-refinement technique for verifying a library of events and aspects. This method allows verifying smaller models in most cases, preventing the state explosion problem, thus making model checking feasible in practice for many systems.
- Aid in the difficult task of defining and correcting precise formal specifications. In particular, the necessary abstractions are detected iteratively, through interaction with the user.
- Extend the approach to detect interference now that aspects respond to complex events.
- Show that the approach provides sound abstractions, i.e., when verification succeeds, the property indeed holds in the actual system, and can also be used to check reachability of complex events.

In the next section we provide the necessary background for understanding event and aspect specification and verification. In Section 3 we present our abstraction-refinement technique for aspect verification and in Section 4 the algorithm to define events and correct event specifications is presented. Next, in Section 5 we introduce how these ideas can be used for interference detection among aspects that respond to complex events, including the case of aspects within aspects. Correctness and termination of the algorithms are presented in Section 6. In Section 7, the issue of event reachability is addressed, and finally in Section 8 we conclude.

2. Background

2.1 Specification Language

In this work we consider different specification languages. For aspect specification, LTL[19] is usually used. For event specification, several languages can be used, but we can include all of them in the expressive power of PSL[1] as seen below.

LTL (Linear Temporal Logic) describes temporal properties that must hold along every path. There are certain temporal operators that we will use. Any atomic proposition p is an LTL formula which expresses that in the current state p must be satisfied.

- $G\varphi$: (Globally) expresses that at every state φ must be satisfied.
- $F\varphi$: (Future or Eventually) expresses that there is a state in the future where φ is satisfied.
- $X\varphi$: (Next) expresses that in the next state φ holds.

Past operators can be used too:

- $O\varphi$: (Once) expresses that at some state in the past φ holds.

```
event LowActivity(P product, int purchases)
 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

- $Y\varphi$: (Previous) in the previous state φ is satisfied.

PSL (Property Specification Language) combines LTL with regular expressions. In particular, we will use the operator $\mapsto$ when convenient. A PSL formula of the form $re \mapsto \varphi$ holds if and only if for every sequence starting with a prefix matching the regular expression re, from the state of the last element of the prefix, φ must hold. For example, if the initial state of the system satisfies a, and every successor that satisfies b also satisfies c, then the formula $a; b \mapsto c$ is satisfied.

2.2 Events

Event detectors[3] are aspect-like modules that react to lower-level events by either gathering information in locally defined fields, or processing the gathered information to trigger a detection announcement that can be used either by other event detectors, or by aspects that respond to event detection and may change the underlying system. Evaluation of event detectors only can change local fields and does not otherwise affect the underlying system besides the consequences of being detected. They can be hierarchically composed, thus obtaining complex events and abstracting from syntactic joinpoint matching.

An example of an event detector is presented in Fig. 1, where the event $LowActivity$ is detected whenever a period of examining sales of a product ends and there have not been enough purchases. It is possible to see that this event declaration relies on $RelevantPurchase$, a lower-level event which exposes the purchases being considered. A possible response to this event could be an aspect that applies a discount to the price of the product being considered, in order to encourage future sales.

Given that events are used to define other events, and aspects respond to them, it is important to understand what the event assumes and what it guarantees. In [9] the issue of event specification and verification was addressed. Basically, the specification of an event E consists of what it assumes about the underlying system, and what is guaranteed about the augmented system, that is, the underlying system with the event detection. Event specification can be expressed in LTL, PSL, using automata, Moore machines, Kripke models, regular expressions and others.

For example, the guarantee of $LowActivity$ is given by the state machine which includes a counter capturing the number of relevant purchases in the time period, and if the counter is smaller than `UPPER_BOUND` when $TimeDone$ is true, $LowActivity$ is detected. Otherwise, $LowActivity$ will not be detected even when $TimeDone$ occurs.

However, from now on we will only consider event specification given by a set of PSL formulas, since LTL is included in PSL, and any finite automata, state machine or regular expression can be transformed to an equivalent set of PSL formulas.

For example, if an event guarantee is given by a state machine indicating in which execution states the event should and should not be detected, an equivalent set of PSL formulas can be automatically obtained using standard mechanisms. The main idea is that given a labelled transition system M representing the intended event detection for an event e, where the labels refer to lower-level events, iteratively a different state s is set as final. Then a regular expression re can be obtained respresenting the paths (i.e, the sequences of lower-level events) leading to s. If in M the event e is not detected in s, the PSL formula $re \mapsto \neg e$ represents this behavior. If e is detected in s, $re \mapsto e \wedge exposedInfo\,(s)$ is added to the formulas representing the guarantee of the event e. The expression $exposedInfo\,(s)$ represents the atomic propositions describing the exposed information of the event when reaching the state s.

This way, we have obtained for every possible word in the language derived from the specification in M, whether the event should be detected or not and what should be guaranteed about the exposed information.

2.3 Aspects

In order to apply modular verification, the assume-guarantee method is used for aspects too. The specification of each aspect consists of an assumption and a guarantee, so that each aspect can be verified modularly. In particular, for the verification of an aspect A, it has been shown[10, 12] that the specification needs to include what is assumed about the underlying system (A's *external* assumption), what is assumed about any aspect that may execute during A (A's *internal* assumption) and the guarantee about the augmented system (base system with A woven).

Then, aspect verification can be applied as follows:

1. Build the *tableau* of the assumption. The *tableau* of a formula φ is a state machine which accepts precisely every possible path satisfying φ. There exist automatic mechanisms that, given a formula φ, build the tableau.
2. Weave into the tableau of the assumption the state machine model derived from the aspect advice code. The state machine model can be obtained automatically using existing tools such as Bandera [8].
3. Verify whether the obtained model satisfies the aspect's guarantee.

This technique guarantees that if the verification above succeeds, for any system satisfying the assumption of the aspect, when weaving the aspect on its own to the system, the resulting model satisfies the aspect's guarantee. The main idea is that the tableau - state machine - of the assumption contains every possible path that satisfies the assumption. Therefore, if a system satisfies the aspect's assumption, every path of the system is included in the tableau. And the aspect woven at every necessary place in the tableau in particular includes the paths where the aspect would be woven in any such system.

2.4 Interference detection

Verifying each aspect in a library is usually not enough to guarantee that a system including a set of the aspects in the library satisfies the expected behavior of every aspect. One aspect may affect the behavior of another aspect, for instance when both change the same variables. In previous work on interference detection[10, 12], it has been shown that if in addition to aspect verification we can check that:

- Every aspect A is verified assuming other aspects that may execute as a response (even indirectly) to events detected while A is executing satisfy A's internal assumptions.
- Every aspect preserves the assumption and guarantee of every other aspect. This proof obligation is called pairwise aspect verification. Note that as proven in [13], showing these properties for every pair of aspects is sufficient to guarantee that any subset of the aspects included in a particular system are interference-free.
- Other aspects satisfy the necessary internal assumptions.

Then, there is no interference.

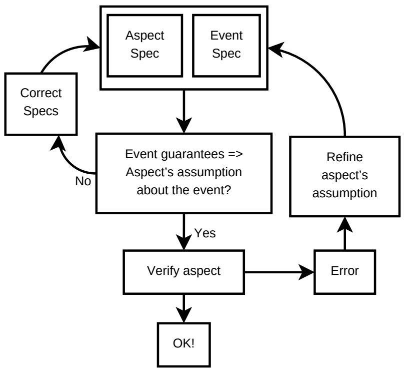
3. Verifying Aspects responding to Complex Events

The problem to address now is aspect specification and verification when aspects respond to complex events.

In previous work it was assumed that aspects could only respond to code-level events. Considering that aspects may respond, add or remove complex events introduces new questions such as: How is aspect verification applied now? What do aspects need to assume about events?

Given that aspects respond to events, we could naively include all relevant event guarantees as part of the assumption of the aspect and then verify the aspect correctness. However, the event hierarchy–where one event detector depends on many simpler event detectors– may include many events, and the model would turn out to be unreasonably large, making model checking unfeasible. Instead, we would like to have the aspect make the necessary assumption about the events on which it depends–an *abstraction* of the full properties of the events, so that the needed guarantee of the aspect can still be shown.

Therefore the aspect assumption should include now the necessary assumptions about the events it relies on. This provides better understanding of what the aspect needs, and



Aspect Spec
Event Spec
Correct Specs
Event guarantees => Aspect's assumption about the event?
No
Refine aspect's assumption
Yes
Verify aspect
Error
OK!

Figure 2. Aspect verification

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

Figure 3. `Discount` aspect

1. $G(LowAct \Rightarrow F\ Disc)$ - always, $LowActivity$ causes the discount to be eventually applied.
2. $G(Disc \Rightarrow O\ LowAct)$ - always, if the discount is applied then once in the past $LowActivity$ was detected.
3. $G(Disc \Rightarrow O\ TmDone)$ - always, if the discount is applied then once in the past $TimeDone$ occurred.

Figure 4. Example: $Discount$ guarantees

if an event definition changes, it can easily be seen which aspects will be affected.

Now, in order to verify aspect correctness, considering their assumptions about events, an abstraction refinement scheme can be used.

In this section, we assume all events have already been specified and verified, with a maximal (strongest possible) guarantee. In the following section we consider the more general case where event guarantees also need to be developed in stages, along with the aspect assumptions and guarantees.

In Fig. 2 the mechanism to find a weak aspect's assumption about the events needed for verification of the aspect guarantees is presented.

Basically, the user provides the aspect guarantee property to be checked, and the specifications of relevant events, and may also provide a first version of the assumption made by the aspect. The first attempt to verify the aspect guarantees should take the most general assumption about the events, e.g., those obtained by using static code analysis [22] of the aspect.

Example 1. We may consider the first overapproximation to be the information obtained by static code analysis. Taking as an example the $LowActivity$ event in Fig. 1 and the $Discount$ aspect in Fig. 3, we can easily see from the code that the event is detected only if $TimeDone$ occurs: $G(LowAct \Rightarrow TmDone)$. Here $LowAct$ is the proposition indicating that the event $LowActivity$ has been detected, while $TmDone$ indicates that $TimeDone$ is detected.

The first step consists of checking that the aspect's assumptions about the underlying events are correct. Afterwards, the aspect is checked for correctness. On success, the procedure ends. However, on fail, the error may be spurious since an overapproximation about the event detection is being used. The refinement scheme is applied until the aspect is proven correct relative to its guarantees, or a non-spurious counterexample is found.

In the next sections we will describe in more detail the steps of the algorithm.

3.1 User input

The user provides a first version of the specification of the aspect to be verified and the specification of all the events involved. As mentioned above, the specification of each aspect ultimately includes what is assumed about the underlying system, what is assumed about other aspects that may be activated within its execution, and a guarantee about the augmented system. Now that aspects respond to complex events it is important to specify what the aspect needs to assume about the underlying events.

3.1.1 Aspect guarantee

The aspect guarantee expresses properties the augmented model (base system + aspect) should satisfy. Examples of guarantees are presented in Fig. 4.

For the aspect guarantee, we will distinguish between two types of formulas: *high-level* and *lower-level*. A high-level guarantee does not involve any lower-level event, only the name of the event causing the aspect to respond. For example, given the event $LowActivity$ and the aspect $Discount$ presented in Fig. 3, a possible high-level guarantee would be: $G\,(LowAct \Rightarrow F\ Disc)$, where $Disc$ represents that a discount has been applied. This is the first example guarantee in Fig. 4.

A lower-level guarantee includes lower-level events. For example, considering the same event and aspect as before, a lower-level guarantee would be: $G\,(Disc \Rightarrow O\ TmDone)$ (the third example guarantee in Fig. 4), since it includes an event not directly responded to by the aspect.

3.2 Overapproximation

The aspect assumption about the underlying events usually represents an overapproximation formula φ of the event detection. This means that every sequence of underlying events where the event is detected in any concrete model is included in the set of sequences of events where the formula φ indicates detection, and the same occurs for every sequence of underlying events where the event is explicitly not detected by φ. In addition, the overapproximation may be satisfied by additional sequences where the event detection does not coincide with the actual behavior. That is, all the actual behaviors are included, but additional behaviors (representing the overapproximation) are also allowed.

It is possible to see the formula in Example 1 represents an overapproximation of the event detection. Every sequence of lower-level events in the guarantee of $LowActivity$ satisfies this formula. However, additional sequences are also included such as those where the event is detected because of $TimeDone$ even if it shouldn't be.

If the guarantee (R) to be proven about the aspect is a high-level guarantee, the overapproximations obtained from static code analysis may be enough. Therefore, the actual event detection is not needed as part of the tableau of the assumption when applying aspect verification.

3.3 Checking the aspect's assumption about the events

When verification is to be applied, the first step is to check that the aspect's assumption about the underlying events is correct. This is done by checking whether the guarantees of the events imply the aspect's assumption. That is, if the aspect's assumption about the underlying events is φ, then this step consists of checking whether φ is implied by the existing event specifications. Provided that the aspect's assumption relies on certain data abstractions (α), these abstractions should be applied to the event guarantees to check that φ is satisfied in the event guarantee under α.

Example 2. The $Discount$ aspect may assume that whenever $TimeDone$ occurs with no relevant purchases, the event $LowActivity$ is detected, represented by the predicate $LowAct$. There is an (implicit) assumption that only one product is being considered, while in the event guarantee N different products are considered, and a predicate $LowActProd_P$ for each product P. Then the abstraction would need to relate to an arbitrary product P_0, and identify $LowActProd_{P_0}$ with the abstract $LowAct$.

If the check in this step fails, the specification should be corrected. For example, the user might have assumed that $TimeDone$ implies $LowActivity$. The actual event guarantee of $LowActivity$ does not imply this fact, as shown by a counterexample including at least UPPER_BOUND relevant purchases. This counterexample provides the user a hint about why the aspect assumption about the event is not correct. If the check in this step succeeds, we can move on to the next step.

3.4 Verifying the aspect

In this step we want to verify whether the aspect guarantee holds under the given assumptions. This can be done applying aspect verification as in [12] taking as part of the assumptions:

1. The external aspect assumption
2. The aspect's assumption about the underlying events.

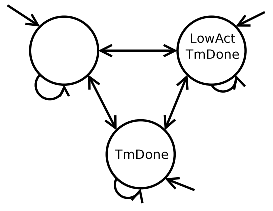


Figure 5. Assumption

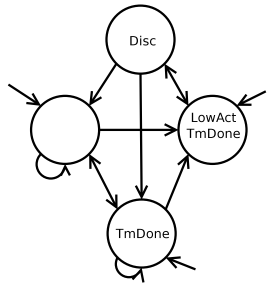


Figure 6. Augmented model

We would build the first tableau of the assumption as all the paths that satisfy

$$G\left(\neg\left(RelPur \wedge TmDone\right) \wedge \left(LowAct \Rightarrow TmDone\right)\right)$$

Here we express that $RelPur$ (representing the detection of $RelevantPurchase$) and $TmDone$ (representing the detection of $TimeDone$) never occur together, and that if the event $LowAct$ is detected then there must be $TmDone$ (from static code-analysis).

Verification is applied as explained in 2.3. If the methodology succeeds, the aspect is correct. If the obtained model does not satisfy the property, it might be the case that the error is caused because of the overapproximation used.

Continuing with the $Discount$ aspect example, in Fig. 5 the tableau of the assumption of $LowActivity$ is presented (always $LowAct$ implies $TimeDone$, as obtained by static code analysis), and in Fig. 6 the augmented model is obtained (whenever $LowAct$ occurs, a $Disc$ state is injected: the discount is applied)

In the obtained model in Fig. 6, all the aspect guarantees in Fig. 4 can be proven. However, we cannot prove that if there are no relevant purchases and $TimeDone$ occurs, the discount is to be applied. More information about the event guarantees is needed in order to prove it. This occurs because the overapproximation indicates only that if there is $LowActivity$ then the aspect is woven, but the only fact known in the tableau of the assumption is that $LowAct \Rightarrow TmDone$. This is reflected by the fact that a key state– where $TmDone$ is true but $LowAct$ is not– is reachable directly from the initial state (because we have made no assumptions so far about the number of relevant purchases).

We obtain the counterexample (1) (moving to the lower state directly from the initial empty state) which contradicts that the discount should be applied when $TimeDone$ occurs and no relevant purchases have been registered in the period.

$$\emptyset, \{TimeDone\}, \ldots \tag{1}$$

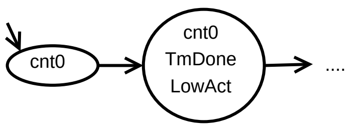


Figure 7. Spurious error detection

3.5 Checking whether a counterexample is spurious

In this section we explain in more detail which event guarantees should be considered and how to detect whether a counterexample is spurious.

Let E be the event that the aspect being considered responds to. The hierarchy tree T_E of the events whose detection may cause E to be triggered can be easily obtained. Similarly, we can consider events and atomic propositions that appear in the guarantee of the aspect, and determine which events influence the value of these propositions.

Now, when checking whether a counterexample π is spurious, we must check whether π satisfies all the formulas representing the guarantees of the events involved.

If there is some event guarantee formula f in the event guarantees (which are assumed correct) not satisfied by π, the error is in fact spurious. Then, f is used to refine the aspect assumption about the events, incorporating it to the tableau used in an attempt to verify the desired aspect guarantee property again.

Note that in the example of Figures 5 and 6 we do not know anything about the calls to $TimeDone$ or the detection of $RelevantPurchase$. Indeed the aspect code itself is sufficient to show that when the event has been detected, the response in the aspect advice is executed. However, we do not know whether the event is detected at the points in execution intended by the aspect. Another reason to include more information about the events is to show that the information provided by the event detector is what is indeed assumed by the aspect.

In the previous section, the counterexample presented in equation (1) is not feasible: in the actual guarantee of $LowActivity$, the corresponding path prefix is presented in Fig. 7, and there the $LowAct$ predicate is true, as required. Therefore, the error is found to be spurious.

In the path, the counter is initially zero, and if $TimeDone$ is detected, then $LowActivity$ must be detected as well. The guarantee of $LowActivity$ contains the PSL formula presented in equation (2) (at any moment where the sequence starts with the counter being zero, and at the next state there is $TimeDone$, then at that next state there must be $LowActivity$).

$$(start \wedge cnt_0)\,; TmDone \mapsto LowAct \tag{2}$$

Refinement is done automatically by adding this assumption to the tableau, thereby eliminating the problematic transition, and then we can check again whether the aspect satisfies its guarantee. We now at least have some basic information about the value of the counter of relevant purchases.

Further refinements will lead to the overall goal of finding the needed assumption: that the counter equals the number of relevant purchases in the time period, and (only) if the counter is smaller than UPPER_BOUND when $TimeDone$ is true, $LowActivity$ is detected.

Note that the refinement is done automatically to eliminate spurious counterexamples, and therefore we obtain only a necessary (weak) assumption about the event. Different aspects may assume different things about events. For example, there may be aspects that do not care about the events' exposed information.

As mentioned before, capturing a weak assumption of an aspect about events is particularly useful for understanding how events affect aspects or which aspects would be affected if some change is made to an event.

3.6 Relation to CEGAR

As mentioned above, CEGAR[6] (Counter-Example Guided Abstraction Refinement) is an already known technique for verifying smaller models and refining only when necessary, automatically.

In our case, we apply CEGAR to the context of Event AOP and take advantage of the facts known. Knowing that we have a modular specification of events, we use these specifications to refine at each step the aspect's assumption about the underlying events. Since events do not affect the state of the underlying system besides the responses caused by them being detected, adding more detail on the event detection is a sound abstraction.

4. Abstraction-Refinement for Events

The next issue to consider is how these abstraction refinement ideas can be used as well to define and correct event specifications and even code definitions. In this case we relax the assumption that we have all events defined and verified, and introduce the algorithm to learn the event definition by interacting with the user.

The steps are presented in the algorithm in Fig. 8. In the next subsections we will describe in detail each of these steps.

4.1 User input

To obtain and refine event specifications, we will consider lower-level event guarantees as input to the algorithm. These lower-level guarantees may have the forms presented in Fig. 9. The first formula expresses that if the regular expression re is matched, then in the state reached the event is detected. The second formula indicates that whenever the regular expression re is matched, then in the state reached, the event is not detected.

These patterns, seen already in the translation of state machine event specifications to PSL, are very natural since the first one captures when the event detection should occur and the second one captures when it should not. Here they are used for partial specification of events and aspect assump-

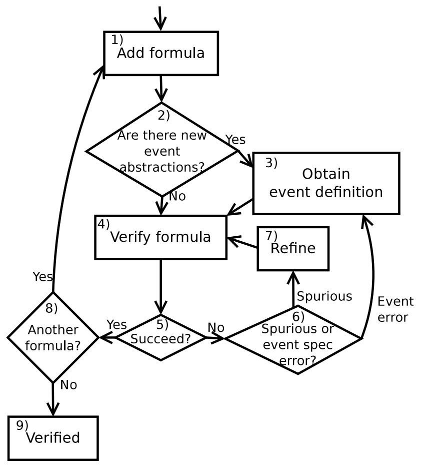


Figure 8. Abstraction-Refinement for Events

1. $re \mapsto eventDetected$
2. $re \mapsto \neg eventDetected$

Figure 9. Patterns of lower-level guarantees

tions. Any other lower-level formula, such as those relating to information gathered or exposed context, is also accepted as input by the algorithm.

The user is also involved in step 6) of the algorithm: using the provided counterexample the user should identify if the error is due to an error in the formula presented, or in the event definition provided in step 3).

4.2 Detecting new abstractions

In the user input, the lower-level events, their exposed information, and related internal fields are represented by means of atomic propositions. We may assume that for method calls and events detected there is an atomic proposition with their name indicating their occurrence. However, when arguments are exposed, these may be defined over unbounded domains, thus abstractions should be applied to obtain a finite model.

The formulas given by the user in the previous step can aid in detecting the necessary abstractions. For example, the formula

$$start; (TmDone) \mapsto (LowAct) \tag{3}$$

expresses that whenever the system starts, and at the immediate following state there is $TimeDone$, then the event is detected for that product. The user should at this step present the assumption that there is only one product (as was already considered in Example 2).

In a next iteration of the algorithm, we may add the case where there is exactly one relevant purchase, learning the atomic propositions and abstractions involved in the exposed information of $RelPur$.

4.3 Obtaining the new event definition

Once we have the necessary abstractions the event's definition can be specified following the synthesis ideas presented in [20]. Basically, the user needs to define for the existing abstractions what the initial state is, and what the transitions are. The work in [20] relies on a state machine description of the underlying system. In our case, we will take the base system to be the tableau of the assumption of the event. Then, the resulting model represents the event detector.

For example, we may assume that $RelPur$ and $TmDone$ never occur together, and we know that somehow we need to count the number of relevant purchases so far, thus the atomic propositions $count_N$ for N between 0 and some chosen constant B are added. B represents the bound such that less than B purchases triggers $LowActivity$. In addition, the atomic proposition $count_{>B}$ represents when there have been more than B relevant purchases. Then, the initial state must satisfy $count_0$, and every occurrence of $RelPur$ or $TimeDone$ should update the value of $count$ as expected.

Alternatively, the user may write the code and the guarantee of the event state machine on his/her own, not using a synthesis algorithm.

4.4 Checking the property

Given an event guarantee property that does not introduce any new abstractions or expose other information, all we need to see is whether the existing event definition causes the event to be detected when expected. If it does and the regular expression looks like 1. of Fig. 9, or if it does not and the formula looks like 2. of the same figure, then the guarantee will hold. Otherwise, there is a problem, either with the event specification, e.g., with the abstractions chosen, or with the definition of the initial state and transitions. For example, consider the case where we are trying to verify property:

$$start; RelPur^B; TmDone \mapsto \neg LowAct \qquad (4)$$

This formula expresses that if there have been at least B relevant purchases since the start of the system, and then there is a call to $TimeDone$, then the event should not be detected.

The user then provides the definition of the initial state and transitions. Let's say that by mistake, the transition caused by TimeDone is expressed as: 1) set the counter to zero, 2) if the counter indicates that there have not been enough relevant purchases of the product trigger the event. That is, the user expresses the transitions on TimeDone in an incorrect order.

In this case, we know that the formula looks like 2. of Fig. 9, so the event should not be detected. However, with the transitions as given above, the event *is* detected. That is, when checking step 5) of the algorithm (Fig. 8) we have not succeeded. In step 6) the user can confirm that the desired guarantee is appropriate, so the event detection code must be corrected. Then, we must go back to step 3) to analyze the initial state and transitions, and continue from there.

4.5 Summary of the algorithm

All the steps presented and explained in the previous sections allow verifying aspects considering event abstractions and finding appropriate abstractions for event specifications.

The algorithm presented in Fig. 2 allows identifying a weak aspect assumption about underlying events, and use these assumptions to verify smaller models.

The algorithm presented in Fig. 8 allows defining and correcting event definitions, interacting with the user and asking appropriate questions. In particular, steps 1), 2) and 3) aid in finding the appropriate abstractions and event guarantees, while steps 4), 5), 6) and 7) constitute the abstraction-refinement scheme for event verification.

By the end of the algorithm, when the user is not interested in adding any other formulas, we obtain an event verified with respect to all the formulas entered, the necessary abstractions for event specification, and the actual event guarantee.

It is important to note that the event should satisfy its guarantee, and adding new aspects or responses to the system, or new properties to the aspect guarantee may involve refining the relevant event guarantees. In this case we need to verify the event detectors for the new guarantees.

4.6 Combination of the algorithms

A useful mechanism is obtained when the algorithms in Figures 2 and 8 are combined, obtaining a technique where the events are defined and corrected and the aspect is verified using the obtained event definitions. Whenever the event guarantees do not imply the aspect assumption about the events (as presented in section 3.3), so the aspect assumption is not a proper overapproximation, the unsatisfied formula is given as input to the algorithm for defining and correcting event definitions (algorithm Fig. 8), and once the assumption can be shown to be an overapproximation of the event guarantees, a new attempt to verify the aspect is made.

For example, if $LowActivity$'s guarantee expresses that on B relevant purchases the event is detected (whereas in the user-provided code strict inequality is used) and the aspect assumes that the event is not detected in that case, the algorithm in Fig. 8 is used, the event guarantee and definition is corrected, and the procedure for aspect verification starts again.

5. Interference detection

5.1 Questions

As mentioned above, there is previous work on interference detection when aspects respond to low-level code events (i.e, traditional joinpoints), including when aspects may execute within other aspects.

The issues we address now are

1. How to check interference because of an aspect activated within another aspect?
2. How to apply pairwise verification (assumption and guarantees must be preserved) now that aspects respond to complex events?

The difficulty arises from the use of complex events, which having a possibly more complex behavior, need to be represented consistently with the aspects' representation (e.g. same atomic propositions for the same method calls, same name for the same variables). Moreover, there is a hierarchy of events to be considered.

Both questions can be solved assisted by the abstraction-refinement scheme presented. For example, we may first check the necessary conditions knowing only the event-detection information obtained from simple static-code analysis. If the checks succeed, since an overapproximation is applied, there indeed is no interference among the actual aspects and events. Otherwise, the error must be analyzed and the model must be possibly refined.

5.2 Aspects within aspects

We first consider question 1. In this section, we will consider two aspects A and B such that B responds to E_B. We first check whether executing the advice of B could violate A's internal assumption. If so, it then is crucial to check whether E_B is in fact detected within A. It should be taken into account that the guarantee of E_B and A may work on different abstractions, therefore the ideas presented before to define events are convenient in this case to detect sound abstractions. Once all the necessary abstractions have been found and the event guarantee built, we check whether E_B is reachable within A. If so, then there is potential interference between A and B. Thus, the steps are:

1. Verify whether B may interfere with A using the event detection information obtained from static-code analysis. This step consists of checking whether B satisfies A's internal assumption.
2. If so, obtain the guarantee of the event being abstracted consistently with the aspect definition, i.e. same atomic propositions representing the same methods, same abstractions for variables of the same type, etc.
3. Restrict the augmented system of A to a precise enough overapproximation of the guarantee of E_B. That is, the resulting model contains only the paths of A for which there is an execution in E_B's guarantee matching the shared variables. If a state triggering within the aspect is reachable, there is potential interference.

The steps presented need not be executed in the order above. An alternative order is first checking whether the event might be detected within A (applying steps 2. and 3.) and only in case it might, check whether B interferes with A. Which order to apply usually depends on how hard it is to prove a property. If the internal assumption can be easily checked then the first order presented should be used (i.e., first checking the internal assumption and if not satisfied, event reachability is checked). If the event detector E_B is very simple then the second option may make verification easier (i.e. first checking event reachability, if reachable then the internal assumption is checked).

```
aspect Auth
before(): doTrans()
  Usr u = requestUsr();
  Pwd p = requestPwd();
  authed = isAllowed?(u, p);

event NotBothEntered
boolen usrSet = false;
boolean pwdSet = false;
after(): requestUsr()
  usrSet = true;
after(): requestPwd()
  pwdSet = true;
when(): isAllowed?(Usr, Pwd)
  if (!usrSet || !pwdSet) trigger();
  usrSet = false;
  pwdSet = false;

aspect ThrowNotBothEntered
before(): NotBothEntered()
  throw new Exception("Both usr and pwd must be entered")
```

Figure 10. Interference example

It is important to note that since the aspect woven may add or remove events, the augmented model is the one restricted to the overapproximation φ of the guarantee of the event.

In step 3, to find a precise enough overapproximation of E_B, the abstraction-refinement scheme can be applied as before.

Soundness of the method: the internal assumption only adds paths. Then, if it is proven for all paths and for any aspect B that *may* execute within another aspect A that B does not interfere with A, then in particular for the actual detection and the actual places where the aspect executes, B will not interfere with A.

Following, we present the method using a running example. In the example in Fig. 10, the aspect $Auth$ authenticates transactions in a website. Before any transaction, the aspect requests a user u and a password p to check whether the user-password is allowed to perform the transaction. The variable $authed$ keeps authentication information. The event $NotBothEntered$ checks whether the method $isAllowed?$ is being called without having entered both a user and password. When $NotBothEntered$ is detected, the aspect $ThrowNotBothEntered$ takes care of it by throwing an exception.

In the example, $Auth$ has an internal assumption that every aspect that executes within $Auth$ returns without throwing exceptions. It is possible to see that if the aspect $ThrowNotBothEntered$ was activated within $Auth$, the internal assumption would not be satisfied (an exception is thrown). However, the event $NotBothEntered$ is not detected within $Auth$ (assuming there are no calls to $isAllowed?$ within other advice that may execute within) and therefore the aspect $ThrowNotBothEntered$ does not execute within $Auth$.

The methodology to verify non-interference among these aspects follows the ideas we have presented.

First, we verify $Auth$ considering its internal assumptions, and next we apply the algorithm presented in Fig. 8 to obtain the appropriate abstractions for the definition of $NotBothEntered$.

Now, using the event definition obtained, we can check whether there may be a joinpoint of $NotBothEntered$ within $Auth$ using some overapproximation of the event detection. Using the overapproximation obtained from static code analysis $G(NotBothEntered \Rightarrow call\_isAllowed?)$, $NotBothEntered$ seems to be reachable within $Auth$. However, when the event detection overapproximation is refined to the actual behavior, we see that the event is not reachable within the aspect because the user and password are in fact entered.

5.3 Pairwise assumption and guarantee preservation

We will now consider question 2. of section 5.1, on how to check whether the assumptions and guarantees are preserved. We recall from the definition of events [3] that events do not have any side-effects besides those of being triggered. We may think that events do not interfere at all since they do not affect the system besides being detected. However, the specification of other events and aspects may influence their detection. For example, some aspect may assume that under certain conditions an event is never detected (to prevent another aspect from executing), or that certain lower-level events are detected in a certain order. Hence, in order to check whether an event may interfere with another event or aspect A, we should first apply check (5).

$$VarsDetection\,(E) \cap Vars\,(\varphi_A) \overset{?}{=} \emptyset \qquad (5)$$

In the equation above, $VarsDetection\,(E)$ returns the set containing all the atomic propositions representing the event detection and parameters. Any other variable is not considered as it is known to be unmodified during the event evaluation. φ_A in $Vars\,(\varphi_A)$ represents either the assumption or the guarantee of A. That is, we check whether the atomic propositions of the event detection appear in the assumption or the guarantee of A. If the intersection in (5) is empty then the event is guaranteed not to interfere with the aspect, otherwise non-interference should be proven as usual [12].

In order to guarantee non-interference we need to check that every aspect A preserves the assumption and the guarantee of every other aspect B. As presented before, the algorithm in Fig. 8 is used to obtain a consistent definition of E_A such that B's assumption and guarantee can be shown to be preserved.

6. Correctness

In this abstraction-refinement scheme, we are checking properties using overapproximations of the event guarantee. We may ask ourselves why this is sound. Following is the lemma proving that assuming the event guarantee we have so far is correct relative to the event detection code, then considering an overapproximation of the event guarantee yields sound results.

Lemma 1. *Checking whether any PSL formula holds in the augmented model of an aspect considering an overapproximation of the event detection is sound.*

Proof. Assuming the procedure is not sound, then there exists a PSL formula φ, and an augmented model M' considering an overapproximation $Guar'_E$ of the guarantee of the event such that $\boldsymbol{\varphi}$ **holds in** $\boldsymbol{M'}$, but does not hold in the augmented model M which detects the event exactly where it should. Because we are checking a PSL formula, there must exist a path π in M where the formula does not hold. However, given that M' includes an overapproximation of the detection of E, in particular it includes the path π (possibly abstracted to the relevant variables). Then $\boldsymbol{\varphi}$ **does not hold in** $\boldsymbol{M'}$ as well, contradicting the assumption. □

Note that every refinement applied in the algorithm in Fig. 2 only restricts existing paths therefore, any property already proven correct continues being satisfied.

We can also consider the termination of the suggested refinement algorithms. The algorithm for aspect verification (Fig. 2) will at most apply a finite number of refinements since there is a finite number of events involved and a finite number of formulas describing their guarantees.

The algorithm for event definition (Fig. 8) continues as long as there are new lower-level formulas the user would like to check. We assume that the user will consider only a bounded number of lower-level formulas. For each formula every step in the algorithm ends, and assuming that eventually the correct event specification is introduced by the user in step 3) the procedure halts.

7. Reachability of Complex Events

Given the whole hierarchy of events it may be hard to see whether there are no contradictions. The problem is by itself undecidable but in some cases certain checks can be applied.

As a first step, we must check that there are not any contradictions among the assumptions of the events, that is, the tableau considering both assumptions is not empty. As a second step, we check that the guarantees do not yield an empty model, and that the events are reachable.

One possibility is to benefit from event modularity and check whether the lower-level events are reachable at some path i.e. in CTL: $EF\,(lowerLevelEventDetected)$ or $EF\,(sequence\;of\;lowerLevelEvents)$. For example, we first verify that $TimeDone$ is reachable with less than `UPPER_BOUND` relevant purchases and assuming that we check whether $LowActivity$ may be reachable.

Another possibility is to check reachability of the conjunction of a set of event detectors (i.e. $E_1 \wedge \cdots \wedge E_n$). In

```
event >200Purchases(P product, int purchases)
 int LOWER_BOUND = 200;
 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) > LOWER_BOUND)
   trigger(product, purchaseInfo.count(product));

 after(P product): call(P.timeDone()) && target(product)
  purchaseInfo.reset(product);

aspect SpecialDiscount
 after(P product): LowActivity(p,int) && >200Purchases(p,int)
  applySpecialDiscount(p);
```

Figure 11. $LowActivity \wedge >200Purchases$

several situations event detectors are expressed as the conjunction of lower-level events, thus it may help checking whether the conjunction may be reachable.

As occurred in previous sections, it is important that the abstractions used to represent the different guarantees involved are consistent among the events being considered. Then, all the guarantees of the events in the conjunction are considered together and the property $\varphi = EF(E_1 \wedge \cdots \wedge E_n)$. If φ is satisfied, the conjunction of the events may occur and the aspect execute. Otherwise, the counterexample may provide the user a hint on why the conjunction never occurs.

In the example of Fig. 11 the aspect $SpecialDiscount$ applies a special discount when $LowActivity$ is detected together with at least 200 purchases. We would like to check whether the aspect might be applied at some place, that is, whether the conjunction of the events $LowActivity$ and $>200Purchases$ is reachable.

Given that both events respond to RelevantPurchase and TimeDone, both event guarantees should have the same atomic propositions to represent lower-level events. In order to obtain consistent abstractions for the guarantee of the events involved in the conjunction, the algorithm in Fig. 8 is used again. The only difference is that the abstractions should be applied simultaneously to both events to guarantee that they are consistent.

Then, it is possible to see that equation (6) is not satisfiable.

$$EF\left(LowActivity \wedge >200Purchases\right) \tag{6}$$

This equation expresses that there exists a computation path where both events are eventually detected at the same time.

This method guarantees that under sound abstractions, if shown unreachable, then the combination of events is in fact unreachable for any computation and provides a hint on the reasons (by means of a counterexample).

8. Conclusions

As is well known, finding the right specification for events and aspects is a difficult task. Nevertheless, providing a specification yields better modular understanding, and avoids the need of reading the code and understanding every internal field and advice every time the event detector or aspect is to be reused. Moreover, the specification, being an abstraction, is less susceptible to change, while the code suffers from more variability (such as applying new naming standards, added information, or others).

The first algorithm allows automatically identifying a weak assumption of an aspect about events. This aids in understanding how a change in the event definition affects the aspects in the system. Moreover, using event detection overapproximations allows applying model checking over smaller models, making it more feasible in practice.

The second algorithm explained can aid in finding appropriate abstractions that allow building the event guarantee and verifying the aspect correctness. The idea is that the user provides formulas the event must satisfy, and the event is synthetized with appropriate abstractions. These ideas are not only relevant for events as defined in [3], but also for code-level events and other pointcut language extensions such as tracematches[2], pointcuts for distributed aspects[21], and others. Moreover, these ideas are relevant for interference analysis and checking event reachability.

Therefore, the algorithms not only provide aspect verification, but present an iterative method based on CEGAR (counter-example guided abstraction refinement) for verifying smaller models, and allowing defining and correcting event specifications.

A partial implementation of the ideas presented above has been achieved, making use of some already existing tools and applying the necessary processing as explained above. The model checker used is NuSMV[5] and modular verification of aspects is achieved using MAVEN[12]. The tool provides the connection between aspect verification and the event guarantee in order to apply refinement when necessary using the information obtained from the model checker.

The main advantage of the tool is that the technical details are hidden from the user, whose only role is to take care of the user input as explained in the sections above. The technical issues the tool takes care on its own include applying verification, determining whether a counterexample is spurious, obtaining automatically the formula representing the counterexample, and refining the assumptions by adding the formula for spurious counterexamples.

As the tool is further developed, applying verification to aspects and defining events should become a much easier task, obtaining a verified library of events and aspects, where every event and aspect is specified modularly.

References

[1] IEEE standard for property specification language (PSL). IEEE Std 1850-2005.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble.

Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40, 2005.

[3] 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*, 2011.

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

[5] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*. Springer-Verlag, 2002.

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*. Springer-Verlag, 2000.

[7] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, 2002.

[8] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[9] 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*, 2011.

[10] 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*, 2012.

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

[12] Max Goldman, Emilia Katz, and Shmuel Katz. MAVEN: modular aspect verification and interference analysis. *Form. Methods Syst. Des.*, 37, November 2010.

[13] Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th Workshop on Foundations of Aspect-Oriented Languages*, 2008.

[14] Emilia Katz and Shmuel Katz. Modular verification of strongly invasive aspects: summary. In *Proceedings of the 2009 Workshop on Foundations of Aspect-Oriented Languages*, 2009.

[15] Shmuel Katz. Aspect categories and classes of temporal properties. *T. Aspect-Oriented Software Development I*, 2006.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[17] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[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] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[20] Shahar Maoz and Yaniv Sa'ar. AspectLTL: an aspect language for LTL specifications. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, 2011.

[21] 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*, 2006.

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[23] M. Störzer and C. Koppen. Pcdiff: Attacking the fragile pointcut problem. Technical report, 2004.