

# Developing and Verifying Response Specifications in Hierarchical Event-Based Systems

Cynthia Disenfeld<sup>(✉)</sup> and Shmuel Katz

Department of Computer Science,  
Technion - Israel Institute of Technology, Haifa, Israel  
{`cdisenfe,katz`}@`cs.technion.ac.il`

**Abstract.** We introduce a CEGAR-based compositional verification technique for verifying response guarantees and finding the necessary assumptions of the response specification about event detectors in hierarchical event-based systems. By taking advantage of the structure of such systems, only the relevant event specifications are considered, and from these only a part of their specifications is learnt as response assumptions. Whenever a spurious counterexample is found (i.e., the abstract counterexample to a response guarantee property is not consistent with the event specifications), our technique modularly finds the necessary refinements that induce state splitting and add fairness constraints to avoid the counterexample automatically. Eventually, either the response guarantee is proved or a real counterexample is found. In addition, new techniques are presented for more feasible spuriousness checking of counterexamples of liveness response guarantees, and to avoid including unnecessary parts of the event detector alphabet in the model of a response.

## 1 Introduction

According to [25], reactive systems are activated by the outside world, and they respond and interact with the environment. These outside world occurrences can be thought of as *primitive events* that are immediately detected. In CEP (Complex Event Processing) [21,32], primitive events may occur at different sources, are processed by *event processing agents/detectors* that may trigger new events, which are finally consumed by different *event consumers*. Event detectors observe the system and environment to identify when an event occurs, and can build more complex event occurrences by detecting sequences, filtering, aggregating information, etc. Events have been combined in other software paradigms such as object-oriented programming (OOP) or aspect-oriented programming (AOP) [30]. In [8], event detectors were introduced in the context of AOP so that they can gather information, be hierarchically composed, and triggered (detect an event occurrence) depending on the lower-level events detected and internal state. Event detectors do not directly influence the underlying system during their evaluation and change only their local variables until the event is detected; then the detection is announced

and parameter values (possibly including gathered information) are exposed to other event detectors *and* responses that can change the system. Here we consider event detectors and responses as in [8] for hierarchical event-based systems.

In this work we propose a reusable compositional verification technique and associated tool called DaVeRS (Developing and Verifying Response Specifications) that under certain assumptions is fully-automated. DaVeRS can verify properties of responses and learn the necessary assumptions about event detectors that allow a successful proof using model checking. The technique takes advantage of the event specifications and their hierarchical structure to check responses modularly using a compositional CEGAR-like (Counterexample Guided Abstraction Refinement) [11] approach combined with an assume-guarantee mechanism. Assuming that the specifications of the events are correct, the system either learns sufficient assumptions about the event detectors to prove the response guarantee being considered, or shows a counterexample sequence of states that violates the desired guarantee and is consistent with all event specifications.

At each step the response and an abstraction of the relevant event detector specifications is considered. Appropriate refinements are obtained when the property is not proven, and we can show that the problem is the current abstraction (and not the actual system). In this case, the counterexample found for the abstract system is called *spurious* relative to the concrete version.

This work encourages modularity on two levels. First, the result of using our abstract-refinement approach yields a minimal collection of events and conjuncts from their specifications that are needed to verify key properties of a response. This allows the isolation of responses and (only) needed event detectors into reusable modules in a library. Second, as will be shown, the techniques applied are themselves modular, involving checks of many small models, rather than an (unfeasible) global model check.

We also introduce two crucial optimizations, that, as seen in the evaluation section, can often make this approach feasible. In order to compare our abstract counterexample to each event detector separately—essential for the modularity described above, it seems necessary to have all of the shared variables among event detectors present in the abstract model to be checked. This would guarantee that any restrictions to those variables that follow from one event detector will be taken into account when we check the counterexample against another detector. We show that this approach (used in related work) is unnecessary overkill, and present a compositional approach that only adds variables and restrictions as absolutely needed. As shown in Sect. 6, this approach leads to significantly improved performance in many cases. We also show new techniques for checking spuriousness and refining liveness properties without the repeated unwinding of the abstract loop used in previous works. These new techniques are also relevant for other compositional CEGAR approaches where the concrete model is finite.

Therefore, the main contributions of this work are:

- Presenting a set of basic assumptions and formalization of reactive systems with hierarchical event specifications as the parallel composition of finite fair discrete systems so that a compositional CEGAR approach can be applied.

- Introducing an alphabet refinement optimization (applicable to other compositional CEGAR approaches as well) to obtain more accurate refinements and avoid redundant iterations.
- Showing an instrumentation-based technique for checking spuriousness of liveness property counterexamples that avoids unfolding an abstract counterexample a very large number of times.
- Including new techniques to find and refine the model with liveness properties.

Note that we consider responses, but the technique is also applicable to a complex event detector that depends on lower-level detectors and primitive events.

We have implemented a tool – DaVeRS – using the tools presented in this paper to evaluate our ideas over different case studies.

Although most of the paper is devoted to the internal operation of the tool, note that this insight is not needed by a typical user. Only the assume-guarantee specifications of the event detectors and the desired guarantees of the responses, and how they react to detected events must be provided. The DaVeRS tool is then completely automatic.

As a running example, we consider the response that adds a helicopter mission in a Car Crash Crisis Management System (CCCMS) [31]. This response depends, among other conditions, on the occurrence of an accident with serious injuries, ambulances not being close enough or being unable to access the location of the crisis, and weather conditions allowing helicopter flight in the area. Applying formal verification allows proving important properties such as “the helicopter mission will always be proposed whenever the necessary conditions hold”, “a helicopter will not be sent whenever an ambulance would arrive sooner”, and others, thus improving system reliability.

Our case studies include several guarantees of the CCCMS system, a Discount response depending on a library of complex event detectors determining different marketing strategies, and a set of responses and event detectors related to security concerns in an email application. These examples illustrate the application of the techniques in different contexts. Input files and iteration examples of the case studies are available at a website<sup>1</sup>.

Section 2 presents the background and basic definitions to understand our model of events and responses, and modular verification and CEGAR techniques. Section 3 presents the basic assumptions and the formalization which allow us to represent hierarchical reactive systems as parallel components and thus use a compositional CEGAR approach. Section 4 explains the methodology, expanding on how each CEGAR-step is applied in our context. In Sects. 5 and 6, some implementation details, the evaluation, results and discussion are presented. Section 7 presents related work and Sect. 8 concludes.

---

<sup>1</sup> <http://www.cs.technion.ac.il/ssdl/research/davers>.

## 2 Background and Basic Definitions

### 2.1 LTL

For specifications describing computations along time, we use Linear Temporal Logic (LTL).

Given a Kripke structure  $M = \langle S, I, R, L \rangle$  over a set of atomic propositions  $AP$  with:

- $S$  is the set of states
- $I \subseteq S$  is the set of initial states
- $R \subseteq S \times S$  is the transition relation
- $L : S \rightarrow 2^{AP}$ , i.e. the atomic propositions that hold at each state

A state  $s$  satisfies an atomic proposition  $p$  in  $AP$  if and only if  $p \in L(s)$ . The semantics of the boolean operators is as expected, for example, a state  $s$  satisfies  $\varphi \wedge \psi$  if and only if  $s$  satisfies  $\varphi$  and also  $s$  satisfies  $\psi$ .

A path in the Kripke structure is a sequence  $\pi = s_0 s_1 \dots$  such that  $s_0 \in I$  ( $s_0$  is an initial state), and for every  $i$ ,  $(s_i, s_{i+1}) \in R$  (there is a transition according to the transition relation).

In addition to boolean operators, formulas can be built with temporal operators. For example,

- $\mathbf{X}\varphi$  (At the next state  $\varphi$ ).
- $\mathbf{G}\varphi$  (From now on, globally  $\varphi$ ).
- $\mathbf{F}\varphi$  (Eventually  $\varphi$ ).

The semantics of these operators is given by the satisfaction relation ( $\models$ ). Given a path  $\pi = \pi_0, \pi_1, \dots$  and  $i$  representing a state in the path ( $\pi_i$ ):

- $(\pi, i) \models \mathbf{X}\varphi$  if and only if  $(\pi, i+1) \models \varphi$  (the path starting from the next state satisfies  $\varphi$ ).
- $(\pi, i) \models \mathbf{G}\varphi$  if and only if for all  $j \geq i$   $(\pi, j) \models \varphi$ .
- $(\pi, i) \models \mathbf{F}\varphi$  if and only if exists  $j \geq i$   $(\pi, j) \models \varphi$ .

Given a model  $M$  and an LTL formula  $\varphi$ ,  $\varphi$  holds in  $M$  if and only if for every path  $\pi$  in  $M$ ,  $(\pi, 0) \models \varphi$ . That is, every path starting from the initial states satisfies the given formula.

In Sect. 4.6, we will also use the path quantifiers  $\mathbf{A}$  (for every path) and  $\mathbf{E}$  (exists path), used in the branching version of temporal logic. Similarly to first order logic quantifiers ( $\forall$  and  $\exists$ ), for any formula  $\varphi$ ,  $\mathbf{A}\varphi$  is equivalent to  $\neg\mathbf{E}\neg\varphi$ . Formulas in LTL must be satisfied by every path, so they can be considered as having an implicit  $\mathbf{A}$  path quantifier at the beginning, and do not include explicit path quantifiers.

Given an LTL formula  $\varphi$ , it is possible to build a state machine containing every possible path satisfying the formula [13]. This state machine is called the *tableau* of  $\varphi$ .

We will consider Kripke structures with fairness constraints ( $\mathcal{F}$ ) which partition the states between those fair and unfair, and the language of the state machine will be given by the *fair* paths (containing infinite fair states) only. For example, the tableau of the formula  $\mathbf{F} p$  is seen in Fig. 1. In the graphic,  $p$  does not hold in the first state ( $s_1$ ),  $p$  holds in the second one ( $s_2$ ), and  $p$  does not hold in the third one ( $s_3$ ).  $s_1$  and  $s_2$  are initial states, and  $s_2$  and  $s_3$  are the fair states (indicated by the double circle). A path is said to be fair if it has infinite fair states. For example, the path  $s_1 s_2 s_3 s_3 s_3 \dots$  is a fair path (infinite times in  $s_3$ ) while the path  $s_1 s_1 s_1 \dots$  is not fair (and in particular it does not satisfy  $\mathbf{F} p$ ). The fairness constraints can be given by explicitly enumerating the fair states, or by a propositional formula  $\varphi$  so that a state is fair if and only if it satisfies  $\varphi$ .

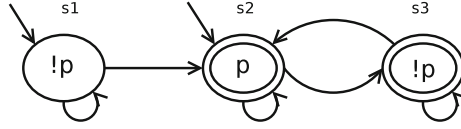


Fig. 1. Fairness example

## 2.2 Events

We distinguish between event occurrences and their detection. Following the definition of [21]: “An *event* is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain”. We call these *event occurrences* to distinguish them from the programming entities or modules, termed *event detectors*, that analyze and announce complex event occurrences. Primitive event occurrences are immediately detected (without a separate detector).

Examples of events detected in a CCCMS are: “a car crash has just been announced”, “an electric storm has begun in the area”, “a fire just started from one of the cars in the accident”, “there are now no helicopters available”, etc. The first three could be considered as primitive input events. A detector of the last event would need to track assignment and release of helicopters from other tasks in order to detect when none are available.

There are several works combining events with existing programming paradigms [3, 8, 20, 22, 28, 34]. In this case, primitive events are given by the base system (or responses). For example, there could be a system with a module **WeatherAnalyzer** responsible for analyzing weather conditions and broadcasting particular situations. The primitive event “an electric storm has begun in the area” in such a system would be given by “the base system has a method call to **broadcast(electric storm)** by **WeatherAnalyzer**”.

Event detectors can update their internal state depending on lower-level event detectors. When an event detector announces detection of an event occurrence (in our version, by executing a *trigger* operation), other detectors and responses can react. This could be implemented by a broadcast mechanism or by having

relevant event modules “listening” for the detection of the event. This implementation detail is not treated here.

Another issue to consider is the event duration. Based on [8], two main approaches are (1) event detectors are reevaluated within each response and between responses (2) given the events detected at a certain point, all responses are applied (no matter if some response may disable the event detection or change the data that affects another response).

The first approach is the one following AspectJ semantics, where an aspect may change the joinpoint matching depending on dynamic information. The second approach may be easier to understand (if the event is detected then no matter which other responses are activated, every response reacting to it will be applied), but does not capture the changes done by other responses. Thus, in this work, we consider the first semantics in the context of reactive systems: at each location of the base system or response where primitive events could occur, the different event detectors are evaluated, thus determining whether a response reacting to an event detector should be applied.

Note that in AspectJ [29], the main language used to express aspects in Java, joinpoints given by a method call or a method execution could last an interval of time (e.g., from the time the method is called until it returns). In [33], a fine grained joinpoint model is presented so that each joinpoint takes an instant of time. That is, there is a joinpoint for the actual call, and a different one for the returning point of a method. Between the two possible semantics of primitive events (region-in-time or point-in-time), we will consider primitive events as those given by the point-in-time joinpoint model [33]. This removes any ambiguity regarding when complex event detectors should be evaluated, while still allowing programs considering the region-in-time semantics to be translated to this model.

Specifications of event detectors (called *event specifications*) [17] include assumptions about the system and underlying events, where exactly the event should be detected, and what is expected about the information exposed by the event detector. Primitive event specifications do not assume anything about lower-level events, but provide the event name and exposed information abstractions to be used by higher-level event specifications. In [17], it was observed that expressing how different lower-level event detector sequences affect the current event detector may be easier with state machines or regular expressions. Thus, to represent event assumptions and guarantees including both state machine definitions and LTL properties, event specifications can be formalized by  $E = \langle X, I, T, P \rangle$  where  $X$  is the set of variables (including those representing the event detection, lower-level events detected, and internal state),  $I$  and  $T$  are the initial and transition relation constraints, and  $P$  is the set of LTL constraints including the event assumptions and guarantees (possibly including safety and liveness formulas). When event detectors are being verified the assumption is used to check the event guarantees. Here we assume that events have already been verified and consider LTL assumptions and guarantees together in  $P$ .

Among the variables in  $X$ , the subset representing event detectors and their exposed information will be called the interface alphabet of  $E$ , since these are the variables possibly affecting or affected by other event specifications.

An abstraction of the event specification  $E_i$  that does not add any constraints is given by  $E' = \langle \emptyset, \text{TRUE}, \text{TRUE}, \emptyset \rangle$ . We denote this as  $\text{TRUE}(E_i)$ .

By considering event specifications, we are abstracting from the implementation. Thus, the results of our technique are sound (to be justified by the correctness proofs), although not complete. That is, if the verification technique succeeds assuming correct event specifications, then the property holds for the actual event implementations. However, a counterexample may seem consistent with all event specifications (really contradicting the desired guarantee), but if more precise event specifications were available, it might be shown spurious. One advantage of using specifications instead of the actual implementation is that the ideas are relevant both while the software is being modeled (because specifications are used rather than implemented code) and when an implementation is already available (where typical model extraction software is used [14, 16]). Other advantages of using event specifications are: (1) proof reusability on any system satisfying the event detectors' and responses' assumptions, (2) abstraction from implementation details, (3) readability of the learnt assumptions, and (4) finer-grained specification dependency understanding: since the learnt assumptions represent a subset of the event specifications needed to prove a property, we can see which event detectors and which parts of their specification may affect that property.

### 2.3 Responses

In reactive systems, not only the event detectors are relevant but also how the different responses (event consumers) affect the system. In this work we consider responses similar to aspects in AOP. Similar to aspects, responses are activated whenever an interesting event is detected. For instance, given the detection of the event representing reaching a call to a method  $m$ , a response can add functionality before or after  $m$  or even override the execution of  $m$  with its own implementation. Differently from event detectors, responses can affect the execution flow and state of the system.

We will consider responses given by:  $A = \langle X_B, X_R, ED, M, P, P_{Ev}, R \rangle$  such that

- $X_B$  is the set of variables of the underlying system.
- $X_R$  is the set of variables local to the response (e.g. response program counter, internal fields).
- $ED$  is a propositional logic formula (on  $X_B$ ) expressing when the response is applied, i.e., to which event detector it reacts.
- $M$  is a finite state machine representing the actual response. It includes initial response states for each activation, a response transition relation, and return states.
- $P$  is an LTL formula (on  $X_B$ ) expressing the base system assumption.

- $P_{Ev}$  is a combination of state machine definitions and LTL formulas (Sect. 2.2) expressing the response assumption about underlying events.
- $R$  is an LTL formula (on  $X_B \cup X_R$ ) expressing the guarantee.

We will note the partial specification of a response  $A$  as  $Spec_A = \langle P, R \rangle$ .

## 2.4 Modular verification

Since reactive systems consist of event detectors and responses, we extend the ideas presented in MAVEN [24] for verifying aspects and adapt and change them to this more general setting while introducing new techniques. In particular the methodology allows the correctness proof of a response guarantee to be reused for different systems by including in its specification an assumption about the underlying system, an assumption about the underlying event detectors, and the guarantee it is expected to satisfy.

As in [24], verification of a response relative to its specification first constructs a model containing the assumption ( $P$ ) about the base system augmented with the response model given by the state machine ( $M$ ). This involves *weaving* the response to the tableau of that assumption at the necessary locations: that is, adding the necessary transitions from the tableau of the assumption ( $T_P$ ) to the response and back at the correct places. The obtained model ( $T_P + M$ ) represents every possible path satisfying the response assumptions augmented with the response behavior and is used to model check temporal logic guarantee properties about the resulting system. If the model check succeeds, the guarantee is true for any system satisfying the response assumptions when the response is woven to it. The given composition does not include  $P_{Ev}$ , that is, it represents the response assumption with the response woven when no assumption about the underlying events is needed. In Sect. 3.2 we describe how the response assumption about the event detectors affects the composition.

## 2.5 CEGAR

CEGAR [11] is an automatic abstraction-refinement technique to verify systems where an overapproximation of the system is considered. The overapproximation represents an abstraction of the concrete system where any path belonging to the concrete system is represented in the abstract one, but more paths may belong as well (the model obtained gets simpler by abstracting from variable values and predicates affecting transitions). When the verification of the abstract system fails, either the counterexample is real or *spurious*, i.e., the counterexample was found because of the overapproximation and not because of the incorrectness of the concrete system. The abstract counterexample is simulated in the concrete model to identify whether the abstract model should be refined. When the counterexample is found spurious, the abstract system is automatically refined by adding information from the concrete system that makes the previous counterexample impossible in the refined version, and a new attempt to verify is initiated.

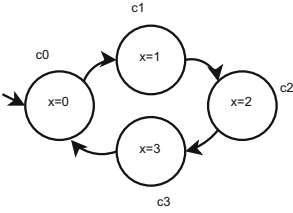


For example, given the concrete system in Fig. 2, and the property given by the LTL formula (1), a possible initial abstraction (Fig. 3) could include the predicates appearing in the formula. In the figure, each abstract state ( $a_i$ ) represents a set of concrete states ( $c_j$ ), e.g. the state  $a_2$  in the abstract model represents both the states  $c_2$  and  $c_3$  of the concrete model. Since there is an edge from  $c_2$  to  $c_3$  in the concrete model, there is a self loop in  $a_2$ .

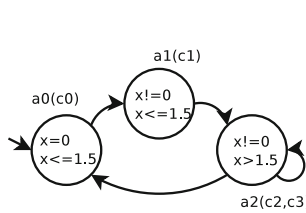
$$\mathbf{G}(x > 1.5 \implies \mathbf{F}(x = 0)) \quad (1)$$

In general, CEGAR techniques calculate the abstract transition relation, so that for every concrete path there exists an abstract path capturing the same states although the abstract model is less precise and contains spurious paths. When the property is checked in the abstract model, a counterexample is found:  $\pi = a_0 a_1 a_2 a_2 \dots$ . This counterexample cannot occur in the concrete model (thus it is spurious) and using CEGAR, a refinement that avoids the counterexample is automatically found. For example, in this case, by splitting the state  $a_2$  with the predicate  $x = 3$ , we obtain the model in Fig. 4 (the transition relation is updated according to the new states). Checking again the property, it is found to be satisfied. Note that in this case the refined model (Fig. 4) contains the same number of states as in the concrete version (Fig. 2). However, in general the number of states required for a CEGAR technique to reach a conclusion is much smaller than in the concrete model.

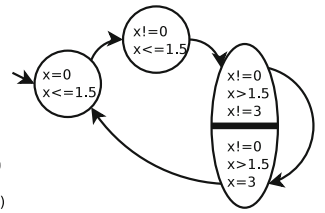
In this work, we will show how we build the abstract model for event and response specifications, how we verify whether a response property holds in the current abstraction, how given a counterexample we analyze spuriousness in new ways, and how we find the necessary refinements to start a new CEGAR cycle.



**Fig. 2.** Concrete model



**Fig. 3.** First abstraction



**Fig. 4.** Refined

Tools implementing CEGAR differ in their program representation for verification, techniques for detecting spuriousness and finding refinements, or the subset of temporal logic considered. Many of these tools interact with SAT or SMT (Satisfiability Modulo Theories) [6] solvers to check spuriousness and to find appropriate refinements. Both SAT and SMT solvers include a tool that obtains the unsat-core of an unsatisfiable set of formulas. The unsat-core is a minimal set of the original set sufficient to prove the model unsatisfiable.

When a counterexample to a safety formula is found in the abstract version, it is sound to consider a finite number of steps ( $n$ ). Thus, it is enough to simulate the counterexample at most  $n$  steps in the concrete version to check whether it is spurious and in case it is, to find the necessary refinements (for example, using the `unsat-core` tool). The work in [11] shows that there is also a finite number of simulation steps necessary to simulate a liveness property counterexample to check spuriousness given by **unfolding** the loop of the abstract counterexample the maximum number of concrete states represented in an abstract state for each state in the loop. Then it is guaranteed that the worst case scenario of the length of the concrete loop matching the abstract one will be covered, but this leads to a bound which is often impractical. Here, we show a more efficient instrumentation approach to check spuriousness for liveness. Using it, we can efficiently detect whether the counterexample is spurious and find the necessary refinements.

## 2.6 Compositional CEGAR

There has been previous work applying CEGAR modularly, i.e. checking spuriousness and finding refinements considering one module (of a generalized alphabet parallel composition [35]) at a time. The generalized alphabet parallel composition of two components  $A$  and  $B$  allows the components to move to their next state asynchronously on non-shared symbols, and requires them to be able to synchronize (both take the same step) on shared symbols that need to be consistent in both components. We will show how hierarchical event-based systems (under certain assumptions) can be reduced to this formal model, allowing almost separate spuriousness checking of an abstract counterexample for each component.

Under this schema, previous work assumed that any alphabet symbol belonging to more than one component should belong to the abstracted component as well (correctness in [35]). However, including any symbol in the alphabet that may affect more than one component may not be necessarily relevant for the guarantee being considered, that may not even use the component with the alphabet symbol. By including all the shared symbols, when an abstract counterexample is found for a particular property, the irrelevant variables receive some random values. Since there is no reason to assume that these values are consistent with the component, abstract counterexamples are likely to be found spurious against components because of these arbitrary values, that will be eliminated by refinement. But such refinements are irrelevant for the guarantee being checked, and just delay finding relevant refinements. We will show how to apply CEGAR with only those shared symbols in the abstract version really necessary to ensure consistency, and evaluate the technique in the case studies.

### 3 Parallel Composition Representation

In this section we present the basic assumptions and formal model to be considered when an assume-guarantee strategy is applied for hierarchical event-based systems.

#### 3.1 Basic Assumptions

The ideas in this paper, though shown for CCCMS, Discount and Security case studies, are applicable to any system where there is a distinction between event detectors and responses. Event detectors observe the system to indicate when interesting things occur. They can be hierarchically composed and have an internal state, but they do not change the state of the underlying system while evaluating. Responses react to an event detected, and may change the state of the underlying system or its control flow. In order to use DaVeRS and apply our approach, correct event specifications (Sect. 2.2), the response state machine and a *partial response specification* (given by the assumption about the base system and desired guarantee) should be available. The response assumption about the events is learnt by our technique.

#### 3.2 Formal Model

Following MAVEN [24], applying verification to a response when only primitive events are considered is done by weaving the response to the tableau of the response assumption and checking whether the response guarantee holds.

Given that complex event evaluations do not affect the underlying system (besides by possibly being detected and causing the response to be applied), the evaluation stage can be modelled as occurring instantaneously, thus having at the same state all the evaluations of event detectors. In this *summarized* version, for each current state of the underlying system and lower-level event detectors, there is one state representing the result of these internal calculations. Given a model of a non-summarized event specification, the summarized version can be obtained by calculating the closure from each possible starting event evaluation point till the end of event evaluation, thus obtaining the summary of changes in one state. When the response guarantee does not have the **X** (next) operator and does not refer to the intermediate states within event evaluation, it can be shown that the summarized and non-summarized version of the event detectors include the same paths (ignoring internal event evaluation states) thus equivalently satisfying LTL formulas with the given conditions.

For example, the event detector indicating that “the crisis location has problematic access”, may require some internal calculations and checks. In the summarized version, every state where the lower-level events detected and system state would cause the event detector to be triggered, includes the atomic proposition representing its detection (and every state that would not cause its detection, does not include the atomic proposition).

Now, since every event evaluation is considered instantaneous, a process algebra notation can be used to justify the techniques. By applying a generalized parallel composition ( $\parallel$ ) over the shared symbols to the event specifications we obtain a model in which at each state – according to the current state of the underlying system, current state of the event detectors, and primitive events detected – we can see which complex events are detected, what information is exposed, and how their internal state is updated.

The generalized parallel composition [35] allows synchronizing on part of the symbols (those shared among components), and interleaved behavior on the remaining ones. If two components share the symbols in  $X$ , to apply a transition influencing a symbol in  $X$  in one component, the other component must also be able to apply the transition on that symbol. For symbols outside  $X$ , the component behaviors are interleaved.

We will note the event specification composition by  $E_1 \parallel \dots \parallel E_N$  where each  $E_i$  is the event specification of the event  $i$ .

We want to consider those paths of the system consistent with the event specifications. To do so, we consider the model  $(T_P + M) \parallel E_1 \parallel \dots \parallel E_N$ , which represents every path satisfying the response assumption with the response woven  $(T_P + M)$  (Sect. 2.4) such that it is also consistent with every event specification  $(\dots \parallel E_1 \parallel \dots \parallel E_N)$ .

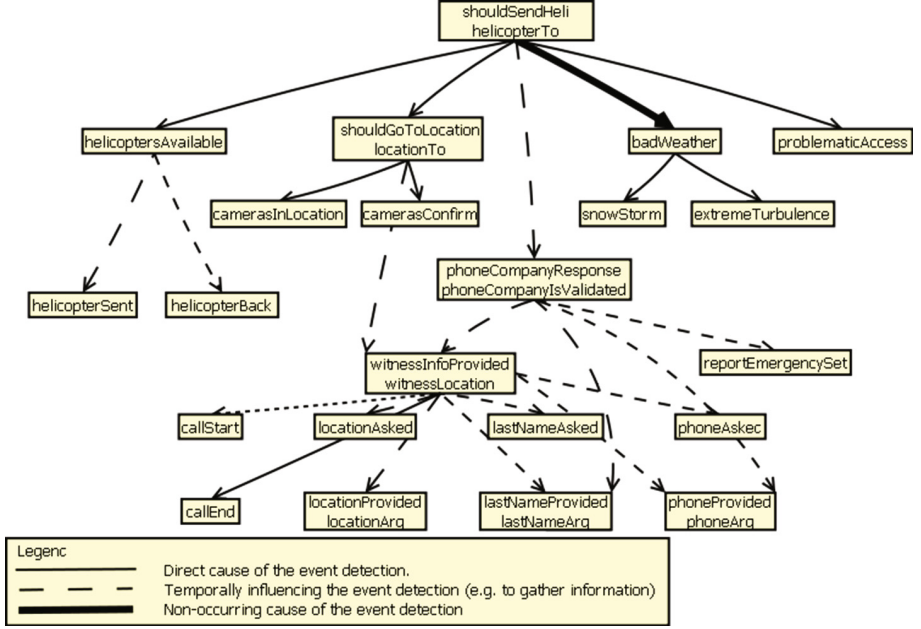
### 3.3 Running Example

Figure 5 shows a fragment of the library of event detectors relevant to the CCCMS example. The arrows represent dependencies including *temporally* (if  $e_i$  has occurred in the past) and *non-occurrence* ones (depending on another event detector not occurring). For instance, the event *shouldSendHeli* depends on the event *problematicAccess* being detected and on *badWeather* **not** being detected at the current state; while *helicoptersAvailable* depends on the history of helicopters that left (*helicopterSent*) and returned (*helicopterBack*). When a box contains multiple names, the first represents the event detection and the rest represent the exposed information. Boxes without exiting arrows represent primitive events.

The response that adds the helicopter mission is activated whenever *shouldSendHeli* is detected (based on a use case of [31]), that is: there is a crisis with serious injuries in a certain location (*shouldGoToLocation*) not easily accessible by normal transportation (*problematicAccess*), the weather conditions do not constrain helicopter flying in the area (*not badWeather*), there are helicopters available (*helicoptersAvailable*) and a response was obtained (*phoneCompanyResponse*) validating the witness information (*phoneCompanyIsValidated*). Each of the complex event detectors has its specification regarding its detection and exposed information. For example, the specification of *badWeather* indicates that this event is detected if and only if there is a snow storm or extreme turbulence.

Following the formal representation of event specifications in Sect. 2.2, the specification of *badWeather* would be given by  $Spec_{badWeather} = \langle X, I, T, P \rangle$  where:

- $X = \{badWeather, snowStorm, extremeTurbulence\}$  (i.e. all the variables representing the detection or the lower-level events and exposed information the specification directly depends on).
- $I$  and  $T$  are TRUE
- $P = \{\mathbf{G}(badWeather \iff (snowStorm \vee extremeTurbulence))\}$



**Fig. 5.** Event dependency graph example

Note that when the event specifications are composed, we obtain at each state the information from all the event detectors, for instance, any state in the composition including *shouldSendHeli* will also include  $\neg badWeather$ ,  $\neg snowStorm$  and  $\neg extremeTurbulence$  (as if the evaluation of *badWeather* were instantaneous).

The event dependency graph illustrates the different components and the hierarchical nature of event-based systems. Each of the boxes will represent a component of a parallel composition, thus getting each component abstracted almost on its own (with only the essential interface from previously checked event detectors); and while in the CEGAR cycle, counterexample spuriousness checking and refinement finding will also be applied considering each component on its own (preserving modularity). The actual input to our technique is the response and the set of event specifications. Since each event specification includes the variables of the lower-level event detectors it depends on, this hierarchy can be inferred automatically.

The following response reacts to *shouldSendHeli* and represents that whenever a helicopter should be sent to a certain location *crisisLocation*, the actual mission of sending an helicopter to that location is added to all the missions to be performed.

```
Response addHelicopterMission
when(Location crisisLocation): shouldSendHeli(crisisLocation)
  allMissions.add(new SendHelicopterMission(crisisLocation));
```

For this example, we consider the following response guarantee:

“If there is a crisis at a certain location (*shouldGoToLocation*), but there is a snow storm (*snowStorm*), the helicopter mission is not added ( $\neg HMAdded$ ).” (2)

We can express this in LTL by

$$\mathbf{G}((\textit{shouldGoToLocation} \wedge \textit{snowStorm}) \rightarrow \neg \textit{HMAdded}) \quad (3)$$

The variable *HMAdded* indicates whether the helicopter mission has already been added to *allMissions*. This variable is not related to the event detectors but to how the response affects the system.

Had we not used a CEGAR approach, model checking would be applied to the model presented in the previous section  $((T_P + M) || E_1 || \dots || E_N)$ , including multiple irrelevant variables and transitions that make calculating the transition relation difficult. In our example, we would have to build the composition of the response assumption and response composed with all the event specifications, when in fact we only need the information about *badWeather*, and from the specification of *badWeather* we do not need to know about *extremeTurbulence*.

## 4 Method

To avoid applying direct verification to the model in the previous section, the abstract model we consider is  $(T_P + M) || E'_1 || \dots || E'_N$  where  $E'_i$  is an overapproximation of  $E_i$  (i.e.  $I \rightarrow I'$ ,  $T \rightarrow T'$ , and  $P' \subseteq P$ ) representing the assumption of the response  $A$  about  $E_i$ , and thus making the composition much simpler (at the first iteration  $E'_i = \text{TRUE}(E_i)$  - as presented in Sect. 2.2). As long as these assumptions are refined (refining some  $E_i$ ), since the  $N + 1$  components are composed, the refinement affects the paths of the augmented response model.

Given a system that satisfies the mentioned assumptions, a CEGAR-like algorithm can be applied (Algorithm 1). The input to the algorithm is the response definition ( $A$ ) and partial response specification  $\langle P, R \rangle$  ( $P$  initially not including any assumption about the event specifications,  $R$  the desired response guarantee), and the event specifications  $S$ .

Initially (line 1), we obtain from all the possible events those from which possible refinements may be obtained (Sect. 4.1), and (line 2) initialize  $E'$  with  $\text{TRUE}$  (every event specification abstraction is  $\text{TRUE}(E_i)$ ), in the first iteration  $(T_P + M) || E' = (T_P + M)$ . At each iteration,  $E'$  includes partial information

obtained from the event specifications necessary to check the response. In line 4 we build the model and in line 5 we check whether it satisfies  $R$  (Sect. 4.2). Since the actual abstraction represents an overapproximation of the actual model to be checked, if it is satisfied with the current refinements, then it is satisfied in the actual model (line 6). Otherwise, in line 6 we check whether the counterexample is due to the abstraction (spurious) or real (Sect. 4.3). If found spurious (line 10), refinements to avoid the current abstract counterexample are obtained (Sect. 4.7). Otherwise, the counterexample is real (line 12) and the CEGAR cycle ends.

---

**Algorithm 1.** Compositional CEGAR for Hierarchical Reactive Systems

---

**input** :  $M, \langle P, R \rangle$ : Response model and response partial specification  
 $S : \text{Set}[E]$ : Event Specification Library  
**output**: *satisfied?*: Indicates whether the response guarantee is guaranteed  
with the given assumptions so far  
 $E'$ : Event specifications' abstraction

```

1  $S' =$  "get subset of relevant events from  $S$ ";
2  $E' = \text{TRUE}$ ;
3 while (True) do
4    $\text{modelToCheck} = (T_P + M) || E'$ ;
5   if  $\text{modelToCheck} \models R$  then
6     satisfied? = True;
7     return
8   else
9     spurious? = "check spuriousness using  $S'$ ";
10    if spurious? then  $E' = E' \cup$  "get spuriousness reasons" ;
11    else
12      satisfied? = False;
13      return
14    end
15  end
16 end

```

---

At each step, the event specification abstractions (response assumption about the events) are refined by adding constraints to  $I$ ,  $T$ , or  $P$  and refining  $X$  accordingly. Since  $P_{E_v}$  is an abstraction of the event specifications, at every step any path in  $E_1 || \dots || E_N$  is a path in  $P_{E_v}$ .

If every call to the model checker or SMT solver terminates (in reasonable time), the technique terminates: every iteration includes at least one refinement (if spurious) and there is a finite number of refinements (obtained from the event specifications). The technique is sound: if verification (after a number of refining iterations) succeeded, then the guarantee indeed holds for the concrete model (every step preserves soundness).

When the CEGAR cycle ends, either the response guarantee holds (and the necessary assumptions about the event detectors have been obtained) or a real counterexample has been obtained.

On success, knowing the fine-grained dependencies (which part of which event specifications are required for a guarantee) allow us to change event detectors or specifications and know exactly which response guarantees are affected. Moreover, the assume-guarantee model used for response and event specifications implies that given any concrete system  $S$ , it is enough to check whether  $S$  satisfies the response assumption about the underlying system, and the learnt event assumptions required to prove the response guarantee ( $R$ ) to assure that  $S$  with the response activated at the correct places will satisfy  $R$ .

On failure, due to the essential-alphabet strategy contribution (Sect. 4.3), in most cases only a few iterations are necessary to find that there is a real counterexample consistent with all the event specifications. This counterexample contains only the variables of the response and those included in the refinements. Then, the counterexample becomes easier to understand (there are fewer variables to be considered).

#### 4.1 Relevant Events

The input contains a library of event specifications. However, not every event may be necessary to check the response guarantee and DaVeRS automatically considers only those potentially relevant. The only event specifications that may include relevant refinements are those sharing some alphabet symbol with the response and those affecting (directly or indirectly) these event detectors. Lower-level event detectors must be considered because the necessary refinements may be in their specifications.

All other event specifications do not share the alphabet symbols with the response nor affect higher-level events sharing some alphabet symbol with the response, and thus do not add any path restriction that would imply a refinement.

In our example, all the event detectors in the fragment of the library presented are relevant (affect the detection of *shouldSendHeli* and are potential sources of refinements to prove the guarantee). However, other events such as “fire started”, “heat wave”, “police at location” are not relevant according to the current definition: they do not affect *shouldSendHeli* or the event detectors relevant to the response guarantee.

#### 4.2 Verification

To apply verification, the model  $(T_P + M) \parallel E'$  is built.  $T_P + M$  is built as in Sect. 2.4. In each step,  $E'$  represents partial information of the event specifications, that is,  $E'_1 \parallel \dots \parallel E'_N$ . If  $E'_i$  contains LTL formulas, the state machine representation of these formulas is considered. Therefore, building the state machine  $(T_P + M) \parallel E'$  is done by including all the constraints of both  $(T_P + M)$  and all the current response assumptions about the events ( $E'$ ).

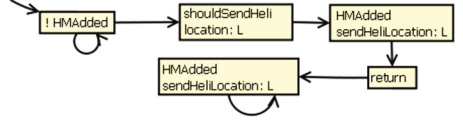
The response guarantee is an LTL formula, thus can be checked for the built model with any LTL model checker.



Figure 6 shows the helicopter response model on its own and Fig. 7 shows the response model after weaving it to its assumption (that the base system does not itself add the helicopter mission). The response model indicates that whenever *shouldSendHeli* is detected, after that state the variable *allMissions* includes the helicopter mission. In the woven model (Fig. 7, as in line 4 of Algorithm 1), the system can remain at the initial state (performing actions irrelevant to our response) until *shouldSendHeli* is detected. At those locations the response is *woven*, and at the return state the execution continues from the base system where it should be with the updated state. Note that this model is very simple (almost trivial): it does not include any information about the remaining events. Any atomic proposition not appearing can have any value.



**Fig. 6.** Response: Helicopter mission



**Fig. 7.** Assumption + Helicopter mission

In the given example, our desired property (Property (3)) is not initially satisfied: there could be a path where *shouldSendHeli* (which activates the response) is detected together with *shouldGoToLocation* and *snowStorm*, causing the response to be activated, even when there is a snow storm. The unexpected behavior is due to the initial overapproximation of the system that does not include (yet) the *indirect* connection where both *shouldSendHeli* and *snowStorm* cannot be detected in the same state.

### 4.3 Checking Spuriousness

As mentioned before (Sect. 2.6), contrary to previous work we allow the abstract version not to automatically include the concrete model alphabet. We will

---

**Algorithm 2.** Checking spuriousness - essential-shared alphabet

---

**input** :  $E_1, \dots, E_n$ : Event specifications

$V_1, \dots, V_n$ : Variables determining the needed abstractions

$\pi$ : Abstract counterexample

**output**: spurious: indicates whether  $\pi$  is spurious with  $E_1 || \dots || E_n$

1 prevAbstractModel =  $M_{True}$ ;

2 **for**  $i$  in  $1..n$  **do**

3     currModel = prevAbstractModel  $\text{---} E_i$ ;

4     spurious = not “ $\pi$  is consistent with currModel”;

5     **if** spurious **then** break;

6     **else** prevAbstractModel = “currModel abstracted to  $\text{COI}(\text{currModel}, V_i \cup \Sigma_\pi)$ ” ;

7 **end**

---

call the strategy of including all the shared symbols of the concrete components in their abstractions as *full-shared alphabet strategy*, and the strategy we present as *essential-shared alphabet strategy*, i.e. we only include the symbols essential to prove the guarantee. When the abstract version does not contain all the shared symbols of the concrete components (the event specifications), and if no further steps are taken, a counterexample could be consistent with every event specification but not with their composition. For example, given  $G(\text{snowStorm} \rightarrow \text{badWeather})$  belonging to  $\text{Spec}_{\text{badWeather}}$  (as in Sect. 3.3) and  $G(\text{shouldSendHeli} \rightarrow \neg \text{badWeather})$  in  $\text{Spec}_{\text{shouldSendHeli}}$  (similarly specified), then there cannot be a state satisfying  $\text{shouldSendHeli} \wedge \text{snowStorm}$ . However, if the abstract version of the response does not include in its alphabet  $\text{badWeather}$ , then the problematic state is consistent with each event specification (for each modular check there is an assignment of  $\text{badWeather}$  making the state possible), but not with their composition. The problem has to do with the shared alphabet among event specifications not being included in the abstract counterexample. Our approach to deal with this situation is to sequentially consider each event specification with a needed subset of the alphabet interface of other event specifications. That way, we can abstract the response alphabet (i.e. not include variables that do not affect the current guarantee).

Given an event specification sequence, we first compute  $\{V_i\}$ :  $V_0$  is empty and  $V_i$  ( $i > 0$ ) contains  $V_{i-1}$  and the variables of  $E_i$  that some event specification appearing later in the sequence includes. The event specification sequence,  $\{V_i\}$  (representing the variables to which each event model should be abstracted), and the abstract counterexample are the input to Algorithm 2 which checks spuriousness for the essential-shared alphabet strategy.

The first event specification of the sequence does not need to be composed with a previous event specification abstracted, thus *prevAbstractModel* is initialized as  $M_{\text{True}}$  (i.e., the model accepting every path). Every other event specification  $E_i$  is composed with the Cone of Influence (COI) [13] reduction of the previous model to the variables that may affect following events in the sequence.

When a model is given by a set of variables  $V$  (i.e. the set of states is every combination of the values of the variables in  $V$ ), an initial constraint (which values are allowed for each variable at the initial states), and a transition relation constraint given by how each variable is affected by the values of the variables at the previous state (i.e.  $v'_i = f_i(V)$ , where  $v'_i$  represents the value of  $v_i$  at the next state), the Cone of Influence  $C$  of a set of variables  $V' \subseteq V$  is the minimal set of variables such that:

- $V' \subseteq C$
- if for some  $v_l \in C$ , its  $f_l$  depends on  $v_j$ , then  $v_j \in C$

Then, the COI reduction of the system is built considering only the variables in  $V'$  and the equations determining their next value. The COI reduction is obtained syntactically from the model definition. Therefore, it does not depend on the size of the model, but on the size of the model description, making it very feasible in practice. Moreover, due to the hierarchical structure of the event

detectors, in general only a small set of variables will be required to be included in the abstraction.

At each step, we abstract *currModel* (last composed with  $E_i$ ) to the COI of  $V_i$  and the alphabet of the counterexample to guarantee that events appearing later in the sequence will be affected by the shared symbols appearing in the current event specification (when composed with the abstraction of the previous model).

If the counterexample is found spurious, then it is inconsistent with *currModel*, which contains the composition of the necessary interfaces with the last event specification considered. This model will be used later to find the appropriate refinements. In the worst case scenario, the COI reduction of the model is the actual model. If this is the case for all the event specifications till step  $i$ , then we are checking spuriousness against the actual composition of these event specifications. However, we are considering hierarchical reactive systems with multiple event detectors, and since not every event specification depends on every other event specification, the obtained model is much smaller than the full composition.

In our running example, *shouldSendHeli* implies that there is not *badWeather* (within the specification of *shouldSendHeli*).

With our algorithm (essential-shared alphabet), *badWeather* does not belong to the initial alphabet. Then, in the first CEGAR cycle the counterexample is:

$$\pi = (\neg HM \text{ Added}), \begin{pmatrix} \text{shouldSendHeli} \\ \text{shouldGoToLoc} \\ \text{snowStorm} \\ \neg HM \text{ Added} \end{pmatrix}, \begin{pmatrix} \text{shouldGoToLoc} \\ \text{snowStorm} \\ HM \text{ Added} \end{pmatrix}$$

That is, the event *shouldSendHeli* is detected in the second state of the abstract counterexample, and in the third state the helicopter mission is added to the set of missions. In the first iteration of Algorithm 2, the counterexample is checked with the specification of *shouldSendHeli* as given (*prevAbstractModel* does not add any restrictions in the first iteration of Algorithm 2). Since the specification of *shouldSendHeli* does not refer directly to *snowStorm*, the abstract counterexample is possible (not spurious so far), and *prevAbstractModel* is updated. Among the variables within the specification of *shouldSendHeli*, there is *badWeather* which appears later in the sequence of events. Since *badWeather* is part of the variables to calculate the COI reduction (it is required by a future event specification in the sequence, the property referring to *badWeather* and *shouldSendHeli* ( $\mathbf{G}(\text{shouldSendHeli} \implies \neg \text{badWeather})$ ) is preserved in *prevAbstractModel*. In the second iteration of the algorithm, the abstraction of the previous model is composed with the specification of *badWeather*. Checking the counterexample with the composition shows the abstract counterexample spurious. From this composition the necessary refinements will be obtained (Sect. 4.7): (*shouldSendHeli* implies not *badWeather*, and *snowStorm* implies *badWeather*) which prevent future counterexamples with *shouldSendHeli* and *snowStorm* in the same state.

Had we not used our optimization, variables shared among components would have been included in the abstract model, and the model checker would have given random values to those variables in the abstract counterexample, thus adding potentially irrelevant refinements. For example, given the following abstract counterexample

$$\pi = (\neg HMAdded), \left( \begin{array}{c} shouldSendHeli \\ shouldGoToLoc \\ \neg \text{helicoptersAvailable} \\ snowStorm \\ \neg HMAdded \end{array} \right), \left( \begin{array}{c} shouldGoToLoc \\ snowStorm \\ HMAdded \end{array} \right)$$

may add the refinement that there must be helicopters available for the event *shouldSendHeli* to be detected, but that refinement is irrelevant for the property being checked.

**Event Ordering.** In both approaches (full-shared, essential-shared alphabet) the order in which the event specifications are considered can significantly affect the performance. Since the goal is to find the refinements as soon as possible, we have observed that a good event ordering is a prioritized search such that starting from the response event detector, we next consider the root of the unexplored subtree in the event dependency graph with the greatest number of atomic predicates in common with the desired response guarantee.

#### 4.4 Modular Approach - Correctness

This section is more technical and proves correctness of Algorithm 2.

Compositional CEGAR approaches in which the abstraction of the components includes the alphabet of the concrete components (without alphabet refinement) has been proven to be correct in [9] based on Lemma 1.

**Lemma 1 (from [35]).** *A path belongs to the parallel composition of a set of components if and only if its projection to the alphabet of each component  $C_i$  is a path in  $C_i$ .*

We have shown in Sect. 3.2 that our model represents hierarchical reactive systems as the parallel composition of the event specifications and the response augmented model, making those ideas applicable.

We now show correctness of our optimization for alphabet refinement, by showing that a path belongs to the alphabet of the composition of two components if it belongs to the COI reduction of the first one composed with the second one. This can then be easily extended to any number of components by induction.

We first present some auxiliary notations, definitions and propositions that we will use to prove the correctness of our approach.

## Notation

- We will use  $M$  to represent any model.
- For any path  $\pi$  and alphabet  $\Sigma$ ,  $\pi \upharpoonright_{\Sigma}$  represents the path projected to include only the symbols appearing in  $\Sigma$ .
- For any model  $M$  and alphabet  $\Sigma$ ,  $M \upharpoonright_{\Sigma}$  represents  $M$  abstracted to the variables appearing in  $\Sigma$ .
- For any model  $M$  and alphabet  $\Sigma$ ,  $COI(M, \Sigma)$  represents the variables of  $M$  that belong to the cone of influence of  $\Sigma$ .
- Given a model  $M$ , path  $\pi$ , alphabet  $\Sigma$ ,  $[M]_{\Sigma}$  represents  $M \upharpoonright_{COI(M, \Sigma)}$  and  $[\pi]_{\Sigma}^M$  represents  $\pi \upharpoonright_{COI(M, \Sigma)}$ . The superscript representing the model will be omitted when clear.

**Definition 1.** A path  $\pi$  over an alphabet  $\Sigma_{\pi}$  is consistent with a model  $M$  over an alphabet  $\Sigma_M$  if and only if there exists a path  $\hat{\pi} \in M$  such that  $\hat{\pi} \upharpoonright_{\Sigma_{\pi}} = \pi \upharpoonright_{\Sigma_M}$ .  $\hat{\pi}$  will be called the witness of  $\pi$  being consistent with  $M$ .

The next two propositions are trivial but will be used to prove the correctness of our approach.

Proposition 1 expresses that for a path  $\pi$  and model  $M$ , if  $\pi \in M$ , taking the COI reduction of the path and of  $M$  with any alphabet, maintains the membership relation.

**Proposition 1.** If  $\pi$  is a path of a model  $M$ , then for an alphabet  $\Sigma$ ,  $[\pi]_{\Sigma}^M$  is a path of  $[M]_{\Sigma}$ .

This is a corollary from the theorem in [13]: Let  $f$  be a CTL\* formula with atomic propositions in  $C$  (where  $C = COI(M, Vars(f))$ ). Then  $M \models f \iff [M]_{Vars(f)} \models f$ .

Proposition 2 expresses that if a path belongs to the cone of influence reduction of a composition, then the path belongs to the cone of influence reduction of each component (restricted to the corresponding alphabet). This can be proven using a (weak) simulation relation among the two models.

**Proposition 2.** Given a path  $\pi$ , two models  $M_1, M_2$ , and alphabet  $\Sigma$ ,  $\pi \in [M_1 || M_2]_{\Sigma} \implies \pi \upharpoonright_{COI(M_i, \Sigma)} \in [M_i]_{\Sigma}$  for  $i = 1, 2$ .

The following lemma explains why it is enough to consider the abstraction of the previous model with the concrete version of the current model to check whether an abstract path is consistent with the composition of two components.

**Lemma 2.** Given  $M_1, M_2$  two models and  $M'_1, M'_2$  their respective overapproximations - not necessarily with the same alphabet, i.e.  $\Sigma_{M'_i} \neq \Sigma_{M_i}$  - then, for any path  $\pi$  in  $M'_1 || M'_2$ ,  $\pi$  is consistent with  $M_1 || M_2$  if and only if  $\pi$  is consistent with  $[M_1]_{\Sigma_{M_2} \cup \Sigma_{\pi}} || M_2$

*Proof.* Let  $M^{abs} = [M_1]_{\Sigma_{M_2} \cup \Sigma_{\pi}}$ . Then,  $M^{abs} || M_2$  represents the composition of two components when using alphabet refinement.

Note that  $\Sigma_{M^{abs}} = COI(M_1, \Sigma_{M_2} \cup \Sigma_{\pi})$ .

$\Rightarrow$ ) Assuming  $\pi$  is consistent with  $M_1||M_2$ , we want to see that  $\pi$  is consistent with  $M^{abs}||M_2$ .

Since  $\pi$  is consistent with  $M_1||M_2$ , then there exists  $\bar{\pi} \in M_1||M_2$  such that  $\bar{\pi} \upharpoonright_{\Sigma_{\pi}} = \pi \upharpoonright_{\Sigma_{M_1}||M_2}$  (from the definition of *being consistent with*).

Since  $\bar{\pi} \in M_1||M_2$ , then by the traces definition of composed components [35],  $\bar{\pi} \upharpoonright_{\Sigma_{M_1}} \in M_1$  and  $\bar{\pi} \upharpoonright_{\Sigma_{M_2}} \in M_2$ . Therefore,  $\bar{\pi} \upharpoonright_{\Sigma_{M_1}}$  is a witness of  $\pi$  being consistent with  $M_1$  and  $\bar{\pi} \upharpoonright_{\Sigma_{M_2}}$  is a witness of  $\pi$  being consistent with  $M_2$ .

We now show that  $\boxed{\pi} = \bar{\pi} \upharpoonright_{\Sigma_{M^{abs}} \cup \Sigma_{M_2}}$  (the witness of  $\pi$  being consistent with  $M_1||M_2$ , restricted to the alphabet of  $M^{abs}||M_2$ ) is a witness for  $\pi$  being consistent with  $M^{abs}||M_2$ .

1. We first show that  $\boxed{\pi}$  is a trace of  $M^{abs}||M_2$ , i.e.  $\boxed{\pi} \upharpoonright_{\Sigma_{M^{abs}}} \in M^{abs}$  and  $\boxed{\pi} \upharpoonright_{\Sigma_{M_2}} \in M_2$ .  
 $\boxed{\pi} \upharpoonright_{\Sigma_{M^{abs}}} \in M^{abs}$ :

- (a)  $\boxed{\pi} \upharpoonright_{\Sigma_{M^{abs}}} = (\bar{\pi} \upharpoonright_{\Sigma_{M^{abs}} \cup \Sigma_{M_2}}) \upharpoonright_{\Sigma_{M^{abs}}} = \bar{\pi} \upharpoonright_{\Sigma_{M^{abs}}}$  (by definition of  $\boxed{\pi}$  and  $\upharpoonright$ ).
- (b) Since  $\bar{\pi} \in M_1||M_2$ ,  $[\bar{\pi}]_{\Sigma_{M_2} \cup \Sigma_{\pi}}$  is a path in  $[(M_1||M_2)]_{\Sigma_{M_2} \cup \Sigma_{\pi}}$  (by Proposition 1).
- (c) Then, by Proposition 2,  $([\bar{\pi}]_{\Sigma_{M_2} \cup \Sigma_{\pi}}) \upharpoonright_{COI(M_1, \Sigma_{M_2} \cup \Sigma_{\pi})}$  is a path in  $[M_1]_{\Sigma_{M_2} \cup \Sigma_{\pi}}$ .
- (d) By definition,  $([\bar{\pi}]_{\Sigma_{M_2} \cup \Sigma_{\pi}}) \upharpoonright_{COI(M_1, \Sigma_{M_2} \cup \Sigma_{\pi})} = \bar{\pi} \upharpoonright_{COI(M_1, \Sigma_{M_2} \cup \Sigma_{\pi})} = \bar{\pi} \upharpoonright_{\Sigma_{M^{abs}}}$ .
- (e) From 1a and d,  $\boxed{\pi} \upharpoonright_{\Sigma_{M^{abs}}} = ([\bar{\pi}]_{\Sigma_{M_2} \cup \Sigma_{\pi}}) \upharpoonright_{COI(M_1, \Sigma_{M_2} \cup \Sigma_{\pi})}$  and from 1c,  $\boxed{\pi} \upharpoonright_{\Sigma_{M^{abs}}}$  is a path in  $[M_1]_{\Sigma_{M_2} \cup \Sigma_{\pi}} = M^{abs}$ .

$\boxed{\pi} \upharpoonright_{\Sigma_{M_2}} \in M_2$ :

$\boxed{\pi} \upharpoonright_{\Sigma_{M_2}} = (\bar{\pi} \upharpoonright_{\Sigma_{M^{abs}} \cup \Sigma_{M_2}}) \upharpoonright_{\Sigma_{M_2}} = \bar{\pi} \upharpoonright_{\Sigma_{M_2}}$ . We already have that  $\bar{\pi} \upharpoonright_{\Sigma_{M_2}}$  is a witness of  $\pi$  being consistent with  $M_2$ . Therefore,  $\boxed{\pi} \upharpoonright_{\Sigma_{M_2}}$  is a witness of  $\pi$  being consistent with  $M_2$ .

2. We now show that  $\boxed{\pi} \upharpoonright_{\Sigma_{\pi}} = \pi \upharpoonright_{\Sigma_{M^{abs}||M_2}}$ :

$$\begin{aligned}
 & - \boxed{\pi} \upharpoonright_{\Sigma_{\pi}} = (\bar{\pi} \upharpoonright_{\Sigma_{M^{abs}} \cup \Sigma_{M_2}}) \upharpoonright_{\Sigma_{\pi}} = (\bar{\pi} \upharpoonright_{\Sigma_{M^{abs}||M_2}}) \upharpoonright_{\Sigma_{\pi}} \text{ by the definition of } \boxed{\pi}. \\
 & - (\bar{\pi} \upharpoonright_{\Sigma_{M^{abs}||M_2}}) \upharpoonright_{\Sigma_{\pi}} = (\bar{\pi} \upharpoonright_{\Sigma_{\pi}}) \upharpoonright_{\Sigma_{M^{abs}||M_2}} \text{ by the definition of } \upharpoonright. \\
 & - (\bar{\pi} \upharpoonright_{\Sigma_{\pi}}) \upharpoonright_{\Sigma_{M^{abs}||M_2}} = \pi \upharpoonright_{\Sigma_{M^{abs}||M_2}} \text{ due to } \bar{\pi} \text{ being the witness of } \pi \text{ being consistent with } M_1||M_2.
 \end{aligned}$$

$\Leftarrow$ ) Assuming that  $\pi$  is consistent with  $M^{abs}||M_2$ , we want to see that  $\pi$  is consistent with  $M_1||M_2$ .

By assumption, there exists  $\bar{\pi} \in M^{abs}||M_2$  witness of  $\pi$  being consistent with  $M^{abs}||M_2$ . Thus,  $\bar{\pi} \upharpoonright_{\Sigma_{M^{abs}}}$  belongs to  $M^{abs}$ . Since  $M^{abs}$  is an abstraction of  $M_1$ , there exists a concrete path  $\bar{\pi}_1^c$  in  $M_1$  such that restricted to

$COI(M_1, \Sigma_{M_2} \cup \Sigma_\pi)$  is equal to  $\bar{\pi} \upharpoonright_{\Sigma_{M^{abs}}}$ , i.e.  $[\bar{\pi}_1^c]_{\Sigma_{M_2} \cup \Sigma_\pi} = \bar{\pi} \upharpoonright_{\Sigma_{M^{abs}}}$ .

Let  $\pi^c = (\bar{\pi}_1^c \times \bar{\pi})$ , that is, the labels at each state are obtained from the current state at  $\bar{\pi}_1^c$  and  $\bar{\pi}$ . From the way  $\bar{\pi}_1^c$  was obtained, any shared symbols between the states of the two paths are consistent.

We will now show that  $\pi^c$  is a witness of  $\pi$  being consistent with  $M_1 || M_2$ .

$\pi^c \in M_1 || M_2$ : To prove this, we just need to prove that restricted to the corresponding alphabets it belongs to both components.

1.  $\pi^c \upharpoonright_{\Sigma_{M_1}}$  is equivalent to restricting to the variables of  $\bar{\pi}_1^c$ . By construction  $\bar{\pi}_1^c \in M_1$ , therefore  $\pi^c \upharpoonright_{\Sigma_{M_1}} \in M_1$ .
2.  $\pi^c \upharpoonright_{\Sigma_{M_2}}$  is equivalent to considering  $\bar{\pi} \upharpoonright_{\Sigma_{M_2}}$ . Since  $\bar{\pi}$  is the witness of  $\pi$  being consistent with  $M^{abs} || M_2$ ,  $\bar{\pi} \upharpoonright_{\Sigma_{M_2}}$  belongs to  $M_2$ .

$\pi^c \upharpoonright_{\Sigma_\pi} = \pi \upharpoonright_{\Sigma_{M_1} || M_2}$ : Both sides of the equation are equal to  $\pi$ , thus the paths obtained by each restriction are equal.  $\square$

**Corollary 1.** *Given  $n$  components  $M_1, \dots, M_n$  and their abstractions  $M'_1, \dots, M'_n$ , for any path  $\pi$  in  $M'_1 || \dots || M'_n$ ,  $\pi$  is consistent with  $M_1 || \dots || M_n$  if and only if:*

1.  $\pi$  is consistent with  $M_1$
2.  $\pi$  is consistent with  $[M_1]_{\Sigma_{M_2} \cup \Sigma_\pi} || M_2$
3.  $\pi$  is consistent with  $\left( ([M_1]_{\Sigma_{M_2} \cup \Sigma_{M_3} \cup \Sigma_\pi} || M_2) \right)_{\Sigma_{M_3} \cup \Sigma_\pi} || M_3$
- ...
- n.  $\pi$  is consistent with  $\left[ \dots \left( [M_1]_{\bigcup_{i=2}^n \Sigma_i \cup \Sigma_\pi} || M_2 \right)_{\bigcup_{i=3}^n \Sigma_i \cup \Sigma_\pi} \dots \right]_{\Sigma_n \cup \Sigma_\pi} || M_n$

The proof is applying the previous lemma inductively on the number of elements in the composition.

The previous proof almost provides the basis of Algorithm 2. We will now show how each iteration is obtained from the composition of  $n$  components.

We use the following auxiliary proposition (provable with the definitions given and showing that there is a simulation relation such that  $[M]_{\Sigma \cup \Sigma'} \leq [M]_\Sigma$ ).

**Proposition 3.** *Given a path  $\pi$ , a component  $M$  and two alphabets  $\Sigma$  and  $\Sigma'$ , if  $\pi$  is consistent with  $[M]_\Sigma$ , then  $\pi$  is consistent with  $[M]_{\Sigma \cup \Sigma'}$ .*

In Corollary 1, we saw that a path  $\pi$  is consistent with the composition of the concrete components by proving the  $n$  items in the list. However, it would seem that each proof item requires calculating a new abstraction of the previous components. For instance in 1. we use  $M_1$  as is. In 2.  $M_1$  is abstracted to the COI of  $\Sigma_{M_2} \cup \Sigma_\pi$ . In 3.  $M_1$  is abstracted to the COI of  $\Sigma_{M_2} \cup \Sigma_{M_3} \cup \Sigma_\pi$ .

We now show that each component can be abstracted only one time.

**Corollary 2.** *Given  $n$  components  $M_1, \dots, M_n$  and their abstractions  $M'_1, \dots, M'_n$ , for any path  $\pi$  in  $M'_1 || \dots || M'_n$ , all the conditions in Corollary 1 hold if and only if all the following are satisfied*

1.  $\pi$  is consistent with  $M_1$
2.  $\pi$  is consistent with  $[M_1] \cup_{i=2}^n \Sigma_i \cup \Sigma_\pi || M_2$
3.  $\pi$  is consistent with  $\left[ ([M_1] \cup_{i=2}^n \Sigma_i \cup \Sigma_\pi || M_2) \right] \cup_{i=3}^n \Sigma_i \cup \Sigma_\pi || M_3$
- ...
- n.  $\pi$  is consistent with  $\left[ \dots \left[ [M_1] \cup_{i=2}^n \Sigma_i \cup \Sigma_\pi || M_2 \right] \cup_{i=3}^n \Sigma_i \cup \Sigma_\pi \dots \right]_{\Sigma_n \cup \Sigma_\pi} || M_n$

*Proof.* Each item in this corollary implies the corresponding item in Corollary 1 because of Proposition 3. Therefore if all the conditions in Corollary 2 are satisfied, then all the conditions in Corollary 1 hold.

For every condition in Corollary 2 proving  $\pi$  consistent with some  $\tilde{M}_1 || \tilde{M}_2$ , there are conditions in Corollary 2 in which each  $\tilde{M}_i$  appears (perhaps abstracted, but due to Proposition 1, it still is consistent). Therefore when all the conditions in Corollary 1 hold, in particular  $\pi$  is consistent with every  $\tilde{M}_i$  appearing in any condition of Corollary 2. Thus, all the conditions in Corollary 2 are satisfied.  $\square$

The last condition of Corollary 2 includes as sub-expressions the previous conditions, from here we infer the algorithm.

$$\begin{array}{c}
 \dots \left[ \begin{array}{c} \underbrace{[M_1]}_{(1)} \quad || M_2 \\ \underbrace{\bigcup_{i=2}^n \Sigma_i \cup \Sigma_\pi}_{(2)} \end{array} \right] \underbrace{\bigcup_{i=3}^n \Sigma_i \cup \Sigma_\pi}_{(4)} \dots \\
 \underbrace{\hspace{10em}}_{(3)}
 \end{array}$$

Part (1) (matching the first condition of Corollary 2), is when the algorithm executes line 3 during the first iteration. The first event specification is composed with  $M_{True}$ , leaving the specification as is.

If not spurious, we abstract the model to the Cone of Influence of  $V_1 \cup \Sigma_\pi$ . Recall that  $V_i$  contains the symbols of  $E_i$  that appear in some event specification later in the sequence. When abstracting the current model to the COI  $\bigcup_{i=2}^n \Sigma_i$ , we are in fact abstracting  $M_1$  to the variables that may appear later in the sequence. This is line 6 of Algorithm 2, first iteration.

Part (3) represents line 3 of the second iteration: composing the previous abstracted model with the current one. The resulting model is checked and if not spurious the algorithm continues by abstracting this resulting model to  $V_2$  (4). This continues until either the abstract counterexample is found spurious (one of the conditions in Corollary 2 does not hold) or it is consistent with every event specification, making the counterexample real.



#### 4.5 Liveness Abstract Counterexample Spuriousness Checking

In this section we describe in more detail line 4 of Algorithm 2.

Depending on whether the property is a safety or liveness guarantee, different techniques are used. Our tool interacts with the model checker NuSMV [1], whose LTL counterexamples are always given by infinite paths, thus we cannot infer from these whether the original property is safety or liveness. To distinguish between safety (a prefix is enough) and liveness properties (counterexample given by an infinite path), in [4] it was shown that given a Büchi automata representing a property  $m$ , the automata represents a safety formula if and only if  $L(m) = L(Cl(m))$  where  $Cl(m)$  is the same as  $m$  but with every state being an accepting state. When the translation of an LTL formula to a state machine does not include fairness constraints, there are no restrictions regarding the states that should occur infinite times thus all states are accepting states (therefore, the state machine represents a safety formula).

Then, when the translation of the formula does not have any fairness constraints we infer that the formula is safety. For safety it is enough to simulate the counterexample (there is a finite path which contains the violation of the safety property) with `currModel`. For liveness, we propose the new instrumentation technique below in place of the usual unfolding seen in Sect. 2.5.

The abstract counterexample (that shows a liveness formula unsatisfied) looks like:  $\underbrace{s_0, \dots, s_i}_{\text{prefix}}, \underbrace{s_{i+1}, \dots, s_j, s_{i+1}, \dots}_{\text{loop}}$ . The prefix can be empty, and the loop may contain one or more abstract states.

When the counterexample is real, the abstract loop might have to be unfolded multiple times to find the concrete counterexample. In order to avoid this unfolding, given the abstract counterexample, we know that each abstract loop iteration contains  $j - i$  states. Each event specification state machine (that is, considering the initial and transition relation constraints together with the state machine translation of the LTL formulas of the specification) is instrumented with a counter that represents the states within the loop. To check spuriousness we check whether there is a concrete path that is consistent with the abstract counterexample.

Given a concrete model  $M = \langle X, I, T, F \rangle$ , where  $X$  is the set of variables and the different values of the variables determine the set of states, the instrumented model is given by  $\tilde{M} = \langle \tilde{X}, \tilde{I}, \tilde{T}, F \rangle$  with:

$$\begin{aligned}
 \tilde{X} &= X \cup \{\text{prefix} : 0..i; \text{cnt} : 0..j - i\} \\
 \tilde{I} &= I \wedge \text{prefix} = 0 \wedge \text{cnt} = 0 \\
 \tilde{T} &= T \wedge (\text{prefix} < i \rightarrow (\text{prefix}' = \text{prefix} + 1 \wedge \text{cnt}' = \text{cnt})) \quad (\text{part 1}) \\
 &\quad \wedge (\text{prefix} = i \rightarrow \text{prefix}' = \text{prefix}) \quad (\text{part 2}) \\
 &\quad \wedge ((\text{prefix} = i \wedge \text{cnt} < j - i) \rightarrow (\text{cnt}' = \text{cnt} + 1)) \quad (\text{part 3}) \\
 &\quad \wedge ((\text{prefix} = i \wedge \text{cnt} = j - i) \rightarrow (\text{cnt}' = 1)) \quad (\text{part 4})
 \end{aligned}$$

That is, we add a variable *prefix* to identify while in the abstract counterexample prefix, and the counter *cnt* to identify the different states of the abstract

loop. Both counters are initialized as 0 and the following transition relation constraints are added:

**part 1.** While in the prefix part of the abstract counterexample, increment *prefix*.

**part 2.** While in the loop part of the abstract counterexample, do not return to the prefix part.

**part 3.** While in the loop part of the abstract counterexample, if it is not yet the end of the abstract loop, increment the counter.

**part 4.** While in the loop part of the abstract counterexample, if it is the end of the abstract loop, reset the counter to represent the beginning of the abstract loop.

Note that if the number of states of the concrete model is  $|S|$ , the number of different states in the instrumented model will be at most  $|S| \cdot (j + 1)$ : in the worst case scenario, every abstract state of the counterexample is matched by every concrete state.

Once the event state machine is instrumented according to the abstract counterexample, the LTL formula to be checked is

$$\neg \left( s_0 \wedge X s_1 \wedge \cdots \wedge \underbrace{X \dots X}_i s_i \wedge \underbrace{X \dots X}_{i+1} G \left( \bigwedge_{k=1}^{j-i} (cnt = k \implies s_{i+k}) \right) \right) \quad (4)$$

The first  $i + 1$  conjuncts characterize each state of the abstract prefix and the remainder what every state within the abstract loop satisfies. Therefore, this property expresses that no path of the concrete model is consistent with the abstract counterexample (the formula within the brackets represents a path consistent with the counterexample). If this formula is satisfied, the counterexample is found spurious with respect to the current event specification; otherwise, a path has been found showing the counterexample consistent with the current event specification and the abstract counterexample is checked against the remaining event specifications. Note that, as before, this formula is checked against each (concrete) event specification modularly, and not the whole concrete system.

In our case study, one possible guarantee could be that whenever the location is provided (*locationProvided*) by a witness of a crisis within a witness call (*inCall*), then that phone call will eventually end (*callEnd*).

$$\mathbf{G} ((locationProvided \wedge inCall) \rightarrow \mathbf{F} callEnd) \quad (5)$$

When we try to verify the guarantee (5) with the abstract model, we may obtain the following abstract counterexample:

$$\begin{array}{lll} \pi = callStart, & locationProvided, & (\neg callEnd)^\omega \\ & \neg callEnd & inCall \\ & & \neg callEnd \end{array}$$

That is, there is a witness call starting in the first state, in the second one the location of the crisis is provided, and then there is no state ending the current call. Then, to check spuriousness with each event specification (and previous abstractions), we build the formula (6) as in the general formula (4). Since there is only one abstract loop state, the counter is not necessary.

$$\neg \left( \begin{array}{l} (callStart \wedge \neg callEnd) \\ \wedge \mathbf{X} (locationProvided \wedge inCall \wedge \neg callEnd) \\ \wedge \mathbf{X} \mathbf{X} \mathbf{G} (\neg callEnd) \end{array} \right) \quad (6)$$

This formula holds whenever the counterexample is spurious (i.e., there is no concrete path matching the abstract counterexample), and does not hold whenever the counterexample is real (i.e., there is a concrete path matching the abstract counterexample).

In Sect. 4.7, we will show how the spuriousness reason of this example is found (and thus used to refine the model).

#### 4.6 Instrumentation Correctness

To see that instrumenting and checking Formula (4), which is  $\neg \left( \underbrace{\overbrace{\dots}^{prefix} \wedge \overbrace{G(\dots)}^{loop}}_{\phi} \right)$ ,

is indeed sound, we prove that the counterexample is consistent with the concrete model if and only if the property is not satisfied in the instrumented model.

Let  $\tilde{M}$  be the instrumented model  $M$  (i.e.  $M$  with the counter instrumentation).  $M \models E\pi$  represents whether  $\pi$  is a path in the model  $M$ .

$$M \models E\pi \underbrace{\iff}_{(a)} \tilde{M} \models E\pi \underbrace{\iff}_{(b)} \tilde{M} \models E\phi \iff \tilde{M} \models \neg \neg E\phi \iff \tilde{M} \not\models A\neg\phi$$

Relation (a) holds since the instrumentation added to the model does not restrict nor add paths to  $M$  (over the variables of  $M$ ), and has no effect in the counterexample. The  $\implies$  part of (b) is easy to see: if  $\pi$  is a path in  $\tilde{M}$ , then there is an assignment to  $cnt$  such that the formula holds in  $\tilde{M}$ . The remaining steps are trivial logic identities. Thus we have obtained that  $M \models E\pi \implies \tilde{M} \not\models A\neg\phi$ , therefore  $\tilde{M} \models A\neg\phi \implies M \not\models E\pi$ . If the instrumented model satisfies formula (4), then the path is not consistent with the model and is found spurious.

We now show the other direction of (b), that  $\tilde{M} \models E\phi \implies \tilde{M} \models E\pi$ :

Let  $\tilde{\pi} = \tilde{s}_0, \tilde{s}_1, \tilde{s}_2, \tilde{s}_3, \tilde{s}_4, \tilde{s}_4, \tilde{s}_5, \dots$  be the witness path of  $\tilde{M} \models E\phi$ . Since  $\tilde{\pi}$  is a path in  $\tilde{M}$ , then  $\tilde{s}_0$  must satisfy the initial conditions defined in  $M$  and for every pair of consecutive states  $\tilde{s}_i$  and  $\tilde{s}_{i+1}$ ,  $M$ 's transition relation constraints must hold. Using this idea and the  $\phi$  definition, we observe that  $\phi$  is also consistent with  $\pi$ : It is clear that any state before the loop of the witness is also at the same position within  $\pi$  (satisfying any initial and transition constraints); and for every

state within the loop, the state has a value of the *cnt* variable, and therefore should be both consistent with the corresponding abstract state and respect the transition relation. Therefore we have found an actual path in  $M$  consistent with  $\pi$ . Note that the formula does not force the concrete loop proving the *globally* part of the  $\phi$  to be of the same length as the abstract counterexample. Instead, it could correspond to  $k$  abstract loop iterations. Once this loop has been found in the concrete instrumented model, every iteration of the concrete loop will be consistent with  $k$  iterations of the abstract one.

Thus, we have obtained that  $M \models E\pi \iff \tilde{M} \not\models \varphi$ , determining whether the error is spurious by checking Formula (4) over the instrumented model.

## 4.7 Refining

Refinement is obtained from an event specification (composed with the previous event abstractions) against which the counterexample has been found spurious. Let *modelFindRefs* be this model from which the refinement will be obtained. The refinement consists of information (initial or transition relation constraints, or LTL properties) and variables appearing in these constraints obtained from the event specifications that the current path (the spurious abstract counterexample) does not satisfy, but any path behaving consistently with the events does.

**Safety Refinement.** For safety guarantees, one SMT unsat-core activation simulating the finite counterexample with *modelFindRefs* is enough to find the necessary refinements. Recall that the unsat-core gives a small subset of the constraints enough to show unsatisfiability. The part from the event specifications in that core should be added to the assumption of the response, in order to prevent obtaining the same abstract counterexample in the future. This part may strengthen the initial states constraints or the relation transition constraints.

Recall that event specifications can include both safety and liveness properties. The state machine representation of a safety formula does not include any fairness constraints (i.e. every path belongs to the language of the state machine), while the state machine representation of liveness formulas must include fairness constraints (restricting the language of the state machine to include only fair paths, c.f. Sect. 2.1). When checking a counterexample with an event specification, we are actually checking it with the state machine representing the event specification. Therefore, for every liveness property  $\varphi$ , the state machine includes the transition relation and fairness constraints representing  $\varphi$ . When translating the liveness formulas into their corresponding state machine, we save what fairness constraints are introduced by each liveness formula within the event specifications.

For example, given our running example, when the abstract counterexample was checked with the COI reduction of *shouldSendHeli* composed with *badWeather*, it was found spurious. This composition included the formula  $G(\text{shouldSendHeli} \implies \neg \text{badWeather})$  translated to a state machine, that is:

$$(\text{shouldSendHeli} \implies \neg \text{badWeather}) \in I$$

$$(shouldSendHeli' \implies \neg badWeather') \in T$$

The SMT model checked includes these assertions, and when simulating the abstract counterexample,  $(shouldSendHeli' \implies \neg badWeather')$  is found as a part of the unsat-core. Since we have saved how each LTL formula is translated to a state machine, we know that this constraint was introduced by the property  $G(shouldSendHeli \implies \neg badWeather)$  within the specification of  $shouldSendHeli$ , and this property is then added to refine the current abstract model.

**Liveness Refinement.** For liveness guarantees, the refinement to avoid a spurious abstract counterexample is also obtained from *modelFindRefs*. This refinement either includes transition relation constraints (refine the model by splitting the abstract states – including the initial states – so that the current path is no longer feasible in the model) or liveness properties (finding the necessary refinement first finds the missing fairness constraints, and then the refinement is given by the liveness formulas within the event specifications introducing those fairness constraints).

This must be handled differently from safety guarantees since finite path simulation does not capture fairness refinements. Previous work either considered only safety formulas (making simulation enough) or considered predicate abstraction or well-founded sets refinement for which the fairness constraints are not part of the refinements.

If the abstract counterexample is consistent when checked against *modelFindRefs* but without any fairness constraints (there is a concrete path  $\tau$  matching the abstract path in the model without fairness), we know that some fairness constraint would avoid the abstract counterexample (and the refinement will be the liveness property whose state machine representation introduced that fairness constraint).

That is, if  $\mathcal{F}$  is the set of fairness constraints, then there exists a path  $\tau$  witness of  $\pi$  being consistent with  $modelFindRefs \setminus \mathcal{F}$ . From  $\tau$ , we can obtain how many times the abstract loop has to be unfolded to represent a concrete loop. Thus, *modelFindRefs* and the counterexample can be translated to SMT to find the unsat-core that makes the counterexample spurious. The counterexample loop is translated in the standard way (the last state of the loop is followed by the first state of the loop) and each fairness constraint  $f$  is translated to “at least one state of the (concrete) loop satisfies  $f$ ”.

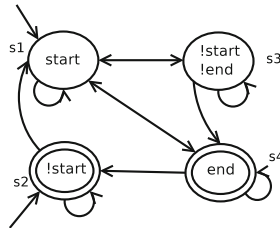
The counterexample was shown consistent with the model without any fairness constraints but spurious otherwise. Thus, the unsat-core of this translation (that includes the fairness constraints) will include at least one of those fairness constraints. Since we have saved for each liveness formula what fairness constraints it introduces, from the fairness constraints in the unsat-core we can easily get the liveness formulas that introduced those constraints and add them as refinements to the abstract assumption of the response, thus avoiding any concrete counterexample with a loop matching  $n$  times the abstract loop. Any abstract counterexample found in a future iteration will not match these concrete paths. This is similar

to other CEGAR work: when an abstract counterexample is found, a predicate splitting a problematic state is added so as to avoid the counterexample. Our fairness refinement “splits” paths when fairness constraints are added.

For instance, let  $\pi$  be the counterexample obtained in the example of Sect. 4.5.

One of the event specifications (*witnessInfoProvided*) assumes that every call that starts eventually ends:  $\mathbf{G}(callStart \rightarrow \mathbf{F}callEnd)$  and that a call does not start and end at the same state  $\mathbf{G}\neg(callStart \wedge callEnd)$ . This can be represented by the state machine in Fig. 8. In the initial state either there is or there is not a call start (states  $s_1$  and  $s_2$  respectively). It is possible to stay in  $s_2$  indefinitely (no matter if there is a call end), but when there is call start it moves to  $s_1$ , which is not a fair state. Therefore, the only way to achieve a fair path is by eventually reaching  $s_4$  (guaranteeing that every call that has started, eventually ends).

To find that this information from the event specifications avoids the current counterexample ( $\pi$ ), our technique first detects that the fairness constraint of this assumption is required.



**Fig. 8.** State machine representing part of WitnessInfoProvided’s specification

When  $\pi$  is checked with this event specification,  $\pi$  is found to be spurious. If we check this same abstract counterexample with the event specification without including any fairness constraints, then the abstract counterexample is consistent (the state machine without any fairness constraints would allow staying infinitely in  $s_1$  or  $s_3$ ). Thus, we can conclude that the refinement required includes fairness constraints.

The translation of the current model to an SMT instance includes the fairness constraint translation that it cannot stay forever in a state where `callStart` has occurred, but `callEnd` does not occur. This constraint will be part of the unsat core, and since it originated from the liveness property  $\mathbf{G}(\text{callStart} \rightarrow \mathbf{F}\text{callEnd})$ , this property is added to the assumptions of the response.

*Transition Relation Refinement.* Now we describe how we find the necessary refinements when we have identified that no fairness constraint avoids the abstract counterexample, i.e. there is some transition (or initial state constraint)

in the abstract model that is not allowed according *modelFindRefs* and the refinement will be a transition relation constraint.

For liveness guarantees not requiring liveness property refinements, instead of simulating the counterexample through repeated SMT activations, we take advantage of the formula representing the counterexample. If the abstract counterexample is spurious, then there must be some state reachable from the initial states but without any actual successor in the concrete model (*modelFindRefs*) consistent with the abstract counterexample. If we consider the product of *modelFindRefs* and the state machine representation of the formula representing the abstract counterexample, then the resulting model will not contain any infinite path. Thus the model checker outputs that the model is empty and returns the diameter needed to reach this conclusion. This diameter serves as the bound for which the state to be split is sure to be found. A single activation of an SMT solver with that bound can then find the needed refinement using the *unsat-core* option.

A possible response guarantee could express that whenever the conditions for *shouldSendHeli* to be detected hold, then the helicopter mission will be added (Property (7)).

$$\mathbf{G}((\text{shouldGoToLoc} \wedge \neg \text{badWeather} \wedge \text{helicoptersAvailable} \wedge \text{problematicAccess}) \rightarrow \mathbf{F} \text{HMAdded}) \quad (7)$$

If we try to verify the guarantee (7), we may obtain the counterexample

$$\pi = \begin{pmatrix} \neg \text{shouldSendHeli} \\ \text{shouldGoToLoc} \\ \neg \text{badWeather} \\ \text{helicoptersAvailable} \\ \text{problematicAccess} \\ \neg \text{HMAdded} \end{pmatrix}^{\omega}$$

That is, all the conditions for the event *shouldSendHeli* are satisfied but the event is not detected with the current abstractions.

In this case, we find the abstract counterexample spurious with the specification of *shouldSendHeli*, even when removing any fairness constraints of the concrete *modelFindRefs*. Therefore we conclude that in this case we need a transition relation refinement.

Composing the model representing the abstract counterexample with *modelFindRefs* shows that there is no infinite path (by considering a diameter of one state the problematic situation is found) and by simulating one step of *modelFindRefs* and the abstract counterexample the necessary refinement (Property (8)) that belongs to the specification of *shouldSendHeli* is found and added to the response assumptions.

$$\mathbf{G}((\text{shouldGoToLoc} \wedge \neg \text{badWeather} \wedge \text{helicoptersAvailable} \wedge \text{problematicAccess}) \iff \text{shouldSendHeli}) \quad (8)$$

## 5 DaVeRS

As noted before, many CEGAR tools use SMT solvers to build the abstract model or find predicate refinements. In this work, DaVeRS also interacts with an SMT solver (SMTInterpol [2]) whenever a bound is known (for example, once the counterexample is known to be spurious because of a missing transition definition). Otherwise DaVeRS interacts with a BDD-based unbounded model checking tool (NuSMV [1]).

Recall that event specifications are given by  $E = \langle X, I, T, P \rangle$  (Sect. 2.2) representing the state machine and temporal logic constraints for the event assumptions and guarantees. Responses are given by  $\langle X_B, X_R, ED, M, P, P_{Ev}, R \rangle$  (Sect. 2.3), representing the variables of the underlying system and response, the event detector to which the response reacts, the state machine definition of the response, and the response partial specification.

The response and event specification are given by text files expressing the contents of each of the categories in their representation, where  $P_{Ev}$  could initially be empty. We use the NuSMV language to express variable domains (boolean, integer range, enumerated types); state machine definitions (initial and transition relations constraints); and LTL formulas (for the temporal logic assumptions and guarantees of event detectors and responses). This is the only input required from the user. DaVeRS automatically applies the techniques presented in this paper.

NuSMV includes a tool (ltl2smv) that allows building the state machine matching a linear temporal logic formula, in particular for liveness properties it introduces the fairness constraints required. We interact with this tool both to build the tableau of the assumption Sect. 4.2, and to obtain the state machine representation of the LTL properties within event specifications. By saving which event specification formula introduced which fairness constraints, following the ideas in Sect. 4.7, we obtain the actual formula that introduced the fairness refinement.

Once we have obtained the LTL to state machine translation of each event specification, we can see every event specification given by a state machine where we can calculate the value of the variables of the next state according to the value of the variables at the previous state, and thus NuSMV can easily calculate the cone of influence of an event specification reduced to a set of variables.

For each response guarantee, the relevant events are obtained and if the full-shared alphabet configuration is used, all the relevant event interface alphabets are added to the response alphabet, otherwise only the symbols directly necessary for building the augmented model and checking the property are added. Then, the application executes the CEGAR cycle as explained in the previous sections until the guarantee has been checked or a real counterexample has been found, and the assumptions learnt about the events are saved.

When DaVeRS terminates, it outputs whether it succeeded or failed, saves the response file with the refinements that led to that conclusion, and in case of failure, outputs the real counterexample.



## 6 Evaluation

We have implemented the DaVeRS tool using these techniques and evaluated it with three extensive case studies: a Car Crash Crisis Management System (CCCMS) [31], a Discount library (as in [18]) and a security concern in an email application. These represent contrasting examples of reusable systems with many options based on event detectors and responses. The goals of the Car Crash Crisis Management System are to receive information about a possible crisis, assess and propose the necessary missions, assign internal/external resources, update the state of the missions, etc. At any moment any of a large number of events occur possibly causing many events to be detected by the event detectors, and appropriate responses (often with guarantees verified in advance) can react and be activated. The Discount case applies discounts according to the events detected (such as buying a product for which sales have not been enough in the last period, discounts for the loyalty program customers, or detecting whenever two of a certain family of products is bought, so that the second one is free). This case study resembles more a library of reusable event detectors and response specifications (and implementations). A user involved in e-commerce can decide which events and responses (that apply discounts of various types) to use over his existing software for handling purchases. The email application includes event detectors triggering when user authorization is required and a response that encrypts any password to be sent. The security concerns in the email application represent the application of a reusable library to a particular domain.

We have considered 34 guarantees for each case study, including assertions about the future and past, identifying when the response should be activated or should not, checking assertions that refer only to the higher-level event or also to lower-level ones, and referring only to the event detection or also to the exposed information.

The CCCMS event dependency graph contains seven complex event specifications relying on 16 primitive events such as a phone call just started, there is a snow storm, etc. The Discount event dependency graph contains six complex event specifications relying on 12 primitive events. The security concern for the email application event dependency graph contains 12 complex event specifications relying on 24 primitive events.

Examples of guarantees of the CCCMS case study include checking (1) that the location parameter with which the helicopter sending mission is created is the one exposed by lower-level event detectors, (2) that if a helicopter mission is added then *shouldSendHeli* must have occurred, and the other way around, or (3) the guarantee considered as a running example.

The Discount response we have considered gives priority to the “buy one, get one free” Discount, and only in case this discount is not applied, the other discounts are considered. Some examples of guarantees of the Discount case study are checking that indeed the indicated discount is prioritized, or that loyalty customers receive the appropriate discounts.

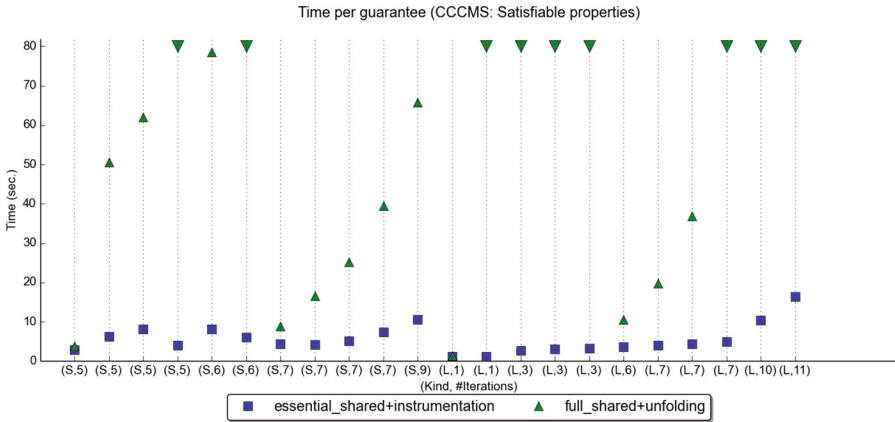
The security concern in the email application requires authentication, for instance, whenever preparing to write an email or accessing the inbox and the

user has not yet authenticated; or when intending to change the account settings. During authentication, a password is sent, and our concern encrypts this password (or any other password sent within the application).

Recall that in classical compositional CEGAR all potentially relevant variables are included in advance (full-shared alphabet strategy), and loop unfolding is used to check liveness. We compared this with our new techniques for adding only needed variables and instrumenting loops with a counter for liveness spuriousness checking.

We compared the different strategies against the different guarantees measuring the time taken by each. All the experiments were carried out on a 2.5 GHz Intel Core i5 (quad-core) with 4 GB RAM running 64-bit Ubuntu 14.04.

For all the case studies, if all the relevant event specifications are included as initial assumptions of the response (CEGAR not applied), the technique takes more than 15 min for each example.



**Fig. 9.** CCCMS - satisfiable properties

Figure 9 shows the time taken by the classical compositional CEGAR techniques `full_shared+unfolding` (triangles) and our `essential_shared+instrumentation` (boxes) for CCCMS satisfiable guarantees. The x axis represents the kind of guarantee (S: Safety, L: Liveness) and the number of iterations required to reach the conclusion when the essential-shared alphabet + instrumentation strategy was used (full-shared alphabet + unfolding always took the same or more iterations). The y axis shows the time in seconds. The inverse triangle on the top of the figure represents when the classical techniques were off the scale of the figure. All such cases took more than 15 min, except for two liveness properties that took 3 and 10 min, respectively. Both for safety and liveness guarantees, our approach works significantly better. In particular, the improvement is even more noticeable for liveness formulas, where our technique took less than 20s for every guarantee considered.

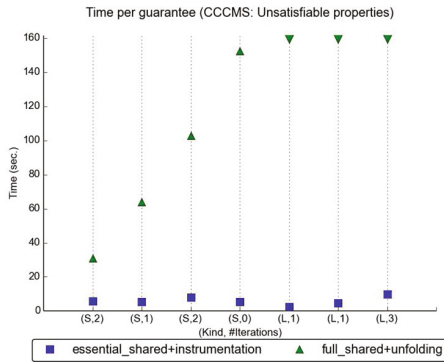
Figure 10 shows the time taken for unsatisfiable CCCMS guarantees (including safety and liveness formulas). Here too, our technique improved over the classical techniques considerably, due to the irrelevant variables needed in classical techniques, and the unfolding strategy used for liveness spuriousness checking. Some guarantees considered giving very similar results to the ones presented in the graphic were not included.

Figure 11 shows the time taken for satisfiable safety guarantees belonging to the Discount (D) and Security (S) case studies. Both techniques show a significant performance improvement when applying our optimizations.

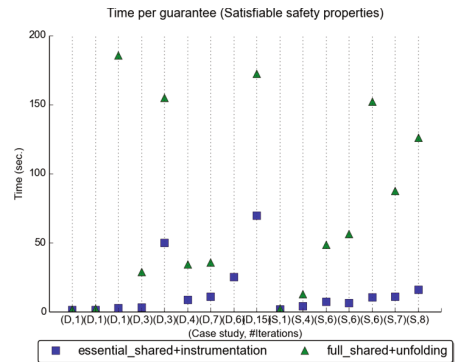
The Discount case study event specifications are more complex than in the other case studies (include information about the customers, products, counters). For this example, liveness and unsatisfiable guarantees reached timeout (15 min) for most guarantees when checking with classical techniques, while our techniques provided results in less than three minutes. For the only two liveness guarantees where the classical techniques terminated, it took around 7 min with the classical techniques and less than two minutes with ours.

The Security case study contains more event detectors (primitive and complex) than the other case studies, making the bound for unfolding liveness guarantee counterexamples significantly larger when using classical techniques. Thus, classical techniques reached timeout for every liveness guarantee in the Security case study (satisfiable and unsatisfiable), while our technique took less than 15 s. For unsatisfiable safety guarantees considered, our technique took less than 20 s, while classical techniques took between 40 s and 2 min (depending on the complexity of the guarantee).

The results in the graphics (and the timeout results) suggest scalability improvements over existing techniques, since now fewer iterations are required to check guarantees, and liveness spuriousness checking can be applied, and terminate in reasonable time. As long as many event specifications are involved, our alphabet refinement optimizations avoids automatically including the interface alphabet of all these, thus considering only the necessary refinements.



**Fig. 10.** CCCMS - unsatisfiable properties



**Fig. 11.** Discount, security - satisfiable safety properties

## 7 Related Work

In [18], the idea of using CEGAR for event systems is proposed, but here we formalize the model and assumptions, show the mechanism and elaborate a tool implementation, including new techniques for handling liveness and reducing the state space that were not presented there.

There are several CEGAR approaches, each with its own way of building the abstract model, verifying, analyzing counterexamples and refining. Among the non-compositional ones, [5, 12, 27] consider only safety formulas, simulate abstract counterexamples against the system's implementation, and learn new predicates that refine the abstract model. The work in [11, 15] present non-compositional CEGAR approaches for also checking liveness formulas. The bound to which liveness counterexamples can be simulated to detect spuriousness in [11] can be very large, and refinements consist of predicates that make the spurious counterexample unreachable, but fairness constraints are not added to the abstract model. As seen in the evaluation section, using instrumentation instead of unfolding the abstract counterexample loop to this bound gives better performance results. In [15], Terminator uses predicate abstraction-refinement for safety formulas and well-founded sets abstraction refinement to prove program fair termination and check liveness specifications. Although we use a symbolic model checker to verify liveness formulas, the fair binary reachability algorithm presented in that work could also be used and then spuriousness checking and refinement finding applied as we have presented. One could argue that from the ranking functions variables and current predicates, one can obtain which of the event specification parts are relevant (that we obtain by translating the event specification to SMT and obtaining the unsat-core). Future work will analyze performance differences between these.

Instrumenting a model to check properties has been considered before [7, 15]. In both works, the model is instrumented to check liveness properties as safety. Though our instrumentation is based on the same principle as theirs (every liveness counterexample consists of a prefix and a loop), the problem addressed and instrumentation itself are different. In those papers, the input is a liveness specification to be checked in a model, in ours it is a path of the abstract model to be checked in the concrete model. Moreover, the instrumentation proposed in [7] is based on non-deterministically *guessing* the initial loop state and checking if it is reachable later in the path (thus checking every possible state). If the original state machine contained  $|S|$  states, the instrumented state machines contains  $|S|^2$  states. Thus, the obtained state machine is much larger than with our instrumentation ( $|S| \cdot j$  - Sect. 4.5). The work in [15] instruments the program to include fairness related assertions as well and reduces the problem to analyzing binary fair reachability (checking that every possible cycle satisfies the fairness constraints). In that paper as well, due to the instrumentation used (guessing the fair state), the instrumented state machines contains more than  $|S|^2$  states.

Other works have considered compositional CEGAR approaches [9, 10, 23, 26] for safety guarantees. The abstract components in [9, 10, 26] have to include every

symbol shared among components, only [23] includes a way to refine the alphabet of the composition of two components.

Our work is mostly useful for event-based systems relying on complex events. That is, our approach is most useful for approaches including hierarchically composed events [3, 8, 19] rather than related work adding events to existing paradigms including limited composition (at most boolean composition) as [22, 34]. For example, tracematches and trace-based aspects [3, 19] trigger the execution of a response depending on a regular pattern of events (detected). These regular patterns include the lower-level events they rely on (described by pointcuts) and the regular expressions to which a method reacts (response). For each regular expression, an event dependency graph can be built. The event dependency graph includes two levels: the one of the joinpoints captured by pointcuts, and the one of the regular expression. We could apply DaVeRS to understand which of the pointcuts are relevant to which guarantees.

## 8 Conclusions

We have presented a practical tool and a CEGAR-based compositional verification technique for verifying response guarantees and finding the necessary assumptions of the response specification about event detectors in hierarchical event-based systems.

The responses and event detectors are specified with state machines and temporal logic formulas, that can either represent a design stage (before implementing in a programming language), or an abstraction of an implemented system.

The basic assumptions about hierarchical event-based systems allow the system to be represented as a parallel composition and thus apply a compositional CEGAR technique. At each step, the response augmented model is built considering only an abstraction of the event specifications, and when a counterexample is found, spuriousness checking and refinement finding is done modularly.

We have presented improvements to state of the art CEGAR techniques for checking spuriousness of liveness property counterexamples, and for including alphabet refinement (even over shared alphabet symbols). The results in the evaluation section validated these as really improving performance with respect to techniques in related work.

## References

1. NuSMV. <http://nusmv.fbk.eu/>
2. SMT. <http://ultimate.informatik.uni-freiburg.de/smtinterpol/>
3. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., De Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. *ACM SIGPLAN Not.* **40**, 345–364 (2005)
4. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. *Distrib. Comput.* **2**(3), 117–126 (1987)

5. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 102–122. Springer, Heidelberg (2001). doi:[10.1007/3-540-45139-0\\_7](https://doi.org/10.1007/3-540-45139-0_7)
6. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2010). [www.SMT-LIB.org](http://www.SMT-LIB.org)
7. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. *Electron. Notes Theoret. Comput. Sci.* **66**, 160–177 (2002)
8. Bockisch, C., Malakuti, S., Akşit, M., Katz, S.: Making aspects natural: events and composition. In: AOSD 2011. ACM (2011)
9. Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., Yorav, K.: Efficient verification of sequential and concurrent C programs. *Formal Meth. Syst. Des.* **25**, 129–166 (2004)
10. Chucuri, F.: Exploiting model structure in CEGAR verification method. Ph.D. thesis, University of Bordeaux I (2012)
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:[10.1007/10722167\\_15](https://doi.org/10.1007/10722167_15)
12. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005). doi:[10.1007/978-3-540-31980-1\\_40](https://doi.org/10.1007/978-3-540-31980-1_40)
13. Clarke, E.M., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (2001)
14. Cobleigh, J.M., Clarke, L.A., Osterweil, L.J.: FLAVERS: a finite state verification technique for software systems. *IBM Syst. J.* **41**, 140–165 (2002)
15. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. *ACM SIGPLAN Not.* **42**, 265–276 (2007)
16. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: ICSE 2000 (2000)
17. Disenfeld, C., Katz, S.: Compositional verification of events and observers (summary). In: FOAL 2011. ACM (2011)
18. Disenfeld, C., Katz, S.: Specification and verification of event detectors and responses. In: AOSD 2013. ACM (2013)
19. Douence, R., Fradet, P., Südholt, M., et al.: Trace-based aspects. *Aspect-Oriented Software Development* (2004)
20. Douence, R., Südholt, M.: A model and a tool for event-based aspect-oriented programming (EAOP). *Techn. Ber., Ecole des Mines de Nantes. TR*, vol. 2(11) (2002)
21. Etzion, O., Niblett, P.: *Event Processing in Action*, 1st edn. Manning Publications Co., Greenwich (2010)
22. Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: Escala: modular event-driven object interactions in scala. In: *Proceedings of the Tenth International Conference On Aspect-Oriented Software Development*. ACM (2011)
23. Bobaru, M.G., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008). doi:[10.1007/978-3-540-70545-1\\_14](https://doi.org/10.1007/978-3-540-70545-1_14)
24. Goldman, M., Katz, E., Katz, S.: MAVEN: modular aspect verification and interference analysis. *Formal Meth. Syst. Des.* **37**, 61–92 (2010)

25. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) *Logics and Models of Concurrent Systems*, pp. 477–498. Springer, New York (1985)
26. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt, W.A., Somenzi, F. (eds.) *CAV 2003*. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003). doi:[10.1007/978-3-540-45069-6\\_27](https://doi.org/10.1007/978-3-540-45069-6_27)
27. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: *POPL 2002*. ACM (2002)
28. Kamina, T., Aotani, T., Masuhara, H.: EventCJ: a context-oriented programming language with declarative event-based context transition. In: *Proceedings of the Tenth International Conference On Aspect-Oriented Software Development*
29. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: *ECOOP (2001)*
30. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) *ECOOP 1997*. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). doi:[10.1007/BFb0053381](https://doi.org/10.1007/BFb0053381)
31. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis management systems: a case study for aspect-oriented modeling. In: Katz, S., Mezini, M., Kienzle, J. (eds.) *Transactions on Aspect-Oriented Software Development VII*. LNCS, vol. 6210, pp. 1–22. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-16086-8\\_1](https://doi.org/10.1007/978-3-642-16086-8_1)
32. Luckham, D.C.: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co. Inc., Boston (2001)
33. Masuhara, H., Endoh, Y., Yonezawa, A.: A fine-grained join point model for more reusable aspects. In: Kobayashi, N. (ed.) *APLAS 2006*. LNCS, vol. 4279, pp. 131–147. Springer, Heidelberg (2006). doi:[10.1007/11924661\\_8](https://doi.org/10.1007/11924661_8)
34. Rajan, H., Leavens, G.T.: Ptolemy: a language with quantified, typed events. In: Vitek, J. (ed.) *ECOOP 2008*. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008). doi:[10.1007/978-3-540-70592-5\\_8](https://doi.org/10.1007/978-3-540-70592-5_8)
35. Roscoe, A.W., Hoare, C.A.R., Bird, R.: *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River (1997)

Transactions on Modularity and Composition I

Chiba, S.; Südholt, M.; Eugster, P.; Ziarek, L.; Leavens,  
G.T. (Eds.)

2016, IX, 269 p. 86 illus., Softcover

ISBN: 978-3-319-46968-3