Compositional Verification of Events and Responses

Cynthia Disenfeld
Compositional Verification of Events and Responses

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Cynthia Disenfeld

Submitted to the Senate of the Technion — Israel Institute of Technology Sh’vat
5775 Haifa February 2015
This research was carried out under the supervision of Prof. Shmuel Katz, in the Faculty of Computer Science.

ACKNOWLEDGEMENTS

I wish to express my most sincere gratitude to my supervisor, Professor Shmuel Katz, for guiding me in my journey as a new researcher, assisting whenever possible, inspiring me and giving invaluable advice, while allowing me to grow and practice my academic independence.

I would also like to thank my parents, Susana and Hector, and my sister, Jesica, for having been always there for me, encouraging me in all of my pursuits and inspiring me to follow my dreams.

Last but not least, special thanks go to my husband, Tal, for his endless love and support.

The generous financial help of the Technion is gratefully acknowledged.
## Contents

### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
</tbody>
</table>

### 1 Introduction

### 2 Preliminaries

- 2.1 LTL
- 2.2 CEGAR
- 2.3 Compositional CEGAR
- 2.4 Aspect-oriented programming
- 2.5 Events
- 2.6 Modular verification
- 2.7 Interference

### 3 Formal Setting

- 3.1 Assumptions
- 3.2 Event specifications
- 3.3 Responses
- 3.4 Modular verification
- 3.5 Related work

### 4 Responses within responses

- 4.1 Running case study
- 4.2 Weaving semantics
- 4.3 Augmented responses
- 4.4 Internal assumptions
- 4.5 Interference
  - 4.5.1 Rule $OK_{PI_{AB}}$
  - 4.5.2 Rule $KW_B(\varphi)$
  - 4.5.3 Partial guarantees
  - 4.5.4 Steps for each response added
  - 4.5.5 Soundness
- 4.6 Cooperation
  - 4.6.1 External assumption cooperation
List of Figures

2.1 Fairness example ................................................................. 10
2.2 LowActivity event ............................................................... 13

4.1 Responses Auth, SaveCookie and EncryptPwd .......................... 22
4.2 Auth model ....................................................................... 25
4.3 Auth+PI model ................................................................. 26
4.4 Internal assumption example ................................................. 27
4.5 Cooperation: Passwords length ............................................. 34
4.6 Removed detections ............................................................. 39

5.1 Event dependency graph example .......................................... 43
5.2 Response: Helicopter mission ............................................... 45
5.3 Assumption + Helicopter mission .......................................... 45
5.4 State machine representing part of WitnessInfoProvided’s specification ......................................................... 55

7.1 CCCMS - Satisfiable properties ............................................. 66
7.2 CCCMS - Unsatisfiable properties ......................................... 68
7.3 Discount,Security - Satisfiable safety properties ........................ 68
Abstract

In reactive systems, responses react to interesting occurrences (events). Event detectors allow detecting complex events by gathering information and hierarchically composing lower-level event detectors. Event detectors are only allowed to observe the system to indicate when something of interest occurs, while responses may change the system state or control flow, or cause other events to be detected.

In this work, we show how under a set of assumptions, the model to verify a response in hierarchical event-based systems is formally represented as the parallel composition of the response and event detectors. We allow responses to be activated within other responses and present the extended specification and necessary proof obligations to verify, detect interference and check cooperation among responses under this scenario. Moreover, we describe how some other possible assumptions affect the formal model and proof obligations.

Counterexample-guided Abstraction Refinement (CEGAR [Clarke et al 2000]) is an iterative algorithm for considering at each verification step an abstraction of the system, and refining the abstraction when a spurious counterexample is obtained (the counterexample is consistent with the abstract model but not with the concrete version of the system).

We have introduced a CEGAR-based compositional verification technique for verifying complex event detectors and response guarantees and finding the necessary assumptions of the response specification about lower-level event detectors in hierarchical event-based systems. This technique makes the proof obligations for modular verification, interference detection and cooperation checking more feasible in practice. Our CEGAR-based technique considers only the relevant event specifications, and learns only a part of their specifications as assumptions.

Whenever a spurious counterexample is found (i.e., the abstract counterexample to a complex event detector guarantee, response guarantee or interference proof obligation is not consistent with the lower-level event specifications), our technique modularly finds the necessary refinements that induce state splitting to avoid the counterexample automatically. Eventually, either verification succeeds 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.

We have implemented these ideas in a tool (DaVeRS) and evaluated the techniques in three extensive case studies.
Publication List


Chapter 1

Introduction

According to [HP85], reactive systems are activated by the outside world, and they respond and interact with the environment. These outside world occurrences can be thought of primitive events that are immediately detected. In CEP (Complex Event Processing) [EN10, Luc01], 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 should be detected, 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) [KLM+97]. In [BMAK11], 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. Responses (event consumers in CEP) can observe or even affect the system's control flow or state. In this work we consider event detectors as in [BMAK11] for hierarchical event-based systems.

In this thesis, we put together and extend the ideas presented in [DK11] (where we have analyzed specification and verification of event detectors), in [DK12] (where we have analyzed interference in the context of aspect oriented programming when aspects can be activated within other aspects) and in [DK13] (where we have presented a compositional abstraction-refinement algorithm, first for aspects and then focused on the techniques for general hierarchical event-based systems).

To be able to apply specification and verification of systems including event detectors and responses, in Chapter 3 we define the formal setting of reactive systems considered. We present a set of basic assumptions which several kinds of systems satisfy, define event detectors, responses and their specifications, and formalize their composition.

Previous work (MAVEN - Modular Aspect Verification [GKK10]) addresses aspect specification and verification by presenting assume-guarantee specifications and applying model checking to check whether the aspect guarantee is satisfied by the state machine representing the aspect assumption augmented with the aspect definition. The augmented model is built by weaving the aspect to the assumption’s state machine, i.e. adding the transitions so that the aspect is activated at the correct places, and returns to the appropriate states. Then, for every system satisfying the aspect assumption, it is guaranteed that the augmented system will satisfy the aspect guarantee.
Verifying each aspect in a library is usually not enough to guarantee that a set of the aspects in the library satisfies its expected behavior. One aspect may affect the behavior of another aspect, and cause interference. In MAVEN, aspect interference was considered under the sequential weaving semantics, which assumes that there are no joinpoints of other aspects within an aspect (and cases that relax this assumption but can still be reduced to the sequential weaving semantics).

In Chapter 4, we have extended and adapted the ideas of MAVEN to the general case in which responses react to complex event detectors, and may cause new responses to be activated within their evaluation (joint-weaving semantics) by considering augmented responses, internal assumptions and distinguishing between global and local guarantees.

The internal assumption of a response \( A \) represents what every other response \( B \) to be executed within the execution of \( A \) must satisfy. Some default internal assumption sorts are presented, e.g. when inserted responses are assumed to satisfy an invariant; however they are not restrictive and any such assumption can be treated. Being aware of the need for internal assumptions when writing response specifications promotes a better understanding of the system, even when not applying formal verification techniques: response interactions, dependencies and cooperation requirements are conveyed by the internal assumptions.

Given a library of responses, they must be checked only once. This analysis either proves non-interference, or provides the guidelines indicating which responses cannot be activated together (and why). Then, for any system which satisfies the necessary response assumptions, the system augmented with the non-interfering responses is already proven to satisfy all response guarantees. Any new response added to the library implies checking this response against all the other responses in the library, but the proofs already done are still valid and used in the new proof of interference-freedom.

Moreover, we have extended the compositional verification and interference detection technique for considering response cooperation. The correctness proof of cooperating responses yields response dependencies, and also allows incrementally verifying as the system is being built - the assumption of a response \( A \) is considered to hold, and then the necessary assistant responses can be developed and verified to satisfy \( A \)'s assumption.

However, the previous ideas suffer one major drawback. To verify a response, we need to know when the response is applied. The model that includes the response and all potentially relevant event detectors can be very large, thus making direct model checking unfeasible.

In this work, we present an abstraction refinement technique (Chapter 5), inspired by CEGAR \([CGJ^{+}00]\) (Counterexample Guided Abstraction Refinement) which allows checking smaller models, and only refining and considering bigger models when necessary. Basically, an abstraction of the model to be checked is used, and if a counterexample is found, then the concrete model is checked automatically to find whether the error is real or spurious (and due to overapproximating the actual system). If the counterexample found represents a spurious error, the abstract model is automatically refined and, with new information from the concrete version, a repeated attempt is made to verify the new abstract model relative to its specification. Appropriate refinements are obtained when the property is not proven, and we can show that the problem is the current abstraction of the system (and not the actual system). In this case, the counterexample found for the abstract system is called spurious relative to the concrete version.

We adapt this idea to verify smaller models when applying response verification (or interference detection), exploiting our knowledge of the relationship between event detectors and responses in order to
abstract from event specifications when verifying a response, and find the missing information from the event specifications when a spurious counterexample is found. Using the abstraction refinement scheme to be presented, the task of writing specifications often becomes much simpler (the assumptions about the event detectors are learned automatically).

Event specifications can include liveness properties required to prove a response guarantee. We show how to obtain liveness properties from the event specifications during the refinement step that avoid instances of the current abstract counterexample.

We also introduce two crucial optimizations, that, as shown in the evaluation section, can often make this approach feasible. Since we want to compare our abstract counterexample to each event detector separately (we use a compositional CEGAR approach), it seems necessary to have all of the potentially important variables of each event detector present in the (abstract) model to be checked, to guarantee that any restrictions 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 with alphabet refinement for any number of components that only adds variables and restrictions as absolutely needed. This approach leads to significantly improved performance in many cases. We also show that by instrumenting an abstract loop with a counter, spuriousness of abstract counterexamples for liveness properties can be shown without repeated unwinding of the abstract loop. Then, our technique automatically finds refinements including predicates or fairness constraints that split states or restrict unfair paths. Both of the new techniques are also relevant for other compositional CEGAR approaches where the concrete model is finite.

These ideas are relevant both while the software is being modeled (because specifications are used rather than implemented code) or when an implementation is already available (where typical model extraction software is used [CDH+00,CCO02]). Learning the needed response assumptions about the event detectors by using event specifications has a number of advantages: 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.

Therefore, the main contributions are to:

- Introduce to response specification the response assumption about underlying events.
- Adapt the specification and interference proof obligations that assume sequential weaving, to be applied when joint-weaving is considered by considering augmented responses, internal assumptions and distinguishing between global and local guarantees. These ideas are further extended for cooperation analysis and for different event evaluation semantics.
- Present an abstraction refinement technique for verifying a library of event detectors and responses. This method allows verifying smaller models in most cases, preventing the state explosion problem, thus making model checking feasible in practice for many systems.
- Show that the approach is sound, i.e., when verification succeeds, the property indeed holds in the actual system.
• Introduce an alphabet refinement optimization (applicable to other compositional CEGAR approaches as well) to obtain more accurate refinements and avoid redundant iterations.

• Show an instrumentation-based technique for checking spuriousness of liveness property counterexamples that avoids unfolding the loop in an abstract counterexample a very large number of times.

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

We have implemented a tool called DaVeRS (Developing and Verifying Response Specifications) that under certain assumptions is fully-automated, to verify responses to complex event detectors and learn the necessary assumptions. The tool (Chapter 6) receives as input the state machine representing the response, the response specification and the libraries of event specifications, and applies our techniques to verify the response and learn the necessary assumptions about the event detectors. Although most of the thesis is devoted to the internal operation of the tool, note that this insight is not needed by a typical user.

We used the tool to evaluate the ideas over three extensive case studies (Chapter 7).

Some of the related work required to understand our contributions is presented in Chapter 2 (Preliminaries), while other related work is presented at the end of each of Chapters 3, 4, and 5. As mentioned before, in Chapter 3 we present the basic assumptions and formal setting, in Chapter 4 we present the extended specifications and proof obligations for interference detection and cooperation analysis under joint-weaving semantics, and in Chapter 5 we present the CEGAR-based technique explaining each step of the methodology. Chapter 6 describes the tool implementing these ideas, and Chapter 7 presents the case studies considered and evaluation results. Finally, in Chapter 8 we conclude.
Chapter 2
Preliminaries

2.1 LTL

For specifications describing computations along time, we use Linear Temporal Logic (LTL) [MP92]. In this work, atomic propositions represent method calls, fields set, variable values and others. In particular, we may assume that for method calls and events detected (c.f. Section 2.5 and Section 3.2) there is an atomic proposition with their name indicating their occurrence.

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

- \( X \varphi \) (At the next state \( \varphi \)). \( X^k \) will be used to denote \( X \ldots X \) \( k \) times.
- \( G \varphi \) (From now on, globally \( \varphi \)).
- \( F \varphi \) (Eventually \( \varphi \)).
- \( \psi U \varphi \) (\( \psi \) until \( \varphi \)).
- \( \psi W \varphi \) (\( \psi \) weak until \( \varphi \)), equivalent to \( \psi U \varphi \lor G \psi \).

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

- \( (\pi, i) \models X \varphi \) if and only if \( (\pi, i+1) \models \varphi \) (the path starting from the next state satisfies \( \varphi \)).
- \( (\pi, i) \models G \varphi \) if and only if for all \( j \geq i \) \( (\pi, j) \models \varphi \).
- \( (\pi, i) \models F \varphi \) if and only if exists \( j \geq i \) \( (\pi, j) \models \varphi \).
- \( (\pi, i) \models \psi U \varphi \) if and only if exists \( j \geq i \) \( (\pi, j) \models \varphi \) and for all \( k : i \leq k < j \) \( (\pi, k) \models \psi \).

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.

Given an LTL formula \( \varphi \), it is possible to build a state machine that accepts every possible path satisfying the formula, and only those paths [CGP01]. This state machine is called the tableau of \( \varphi \). This state machine may include fairness constraints \( (F) \) which partition the possible paths between those fair and unfair, and the language of the state machine will be given by the fair paths only. For example, the tableau of the formula \( F p \) will look as the one in Figure 2.1. In the graphic, \( p \) does not hold in the first
state \((s_1)\), \(p\) holds in the second one \((s_2)\), and in the third one \((s_3)\) it may or may not hold. \(s_1\) 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 \(s_1 s_2 s_3 s_3 \ldots\) is a fair path (infinite times in \(s_3\)) while the path \(s_1 s_1 s_1 \ldots\) is not fair (and in particular it does not satisfy \(\mathcal{F} p\)).

### 2.2 CEGAR

Counterexample-Guided Abstraction Refinement (CEGAR) [CGJ+00] 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. All the steps involved in CEGAR--verifying an abstract model, analyzing if a counterexample is spurious, refining the abstract model--are fully automatic.

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) [BST10] 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 subset of the original set sufficient to prove the model unsatisfiable.

When a counterexample to a safety formula is found in the abstract version, this counterexample contains 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. The work in [CGJ+00] shows that there is also a finite number of simulation steps enough 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. 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. In Chapter 5, 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.
2.3 Compositional CEGAR

There has been previous work applying CEGAR modularly, i.e. checking spuriousness of a counterexample to the abstract system and finding refinements considering one concrete module (of a synchronous composition) at a time. Under this schema, previous work assumed that any alphabet symbol belonging to more than one component (thus possibly affecting their synchronization) should belong to the abstracted component as well (correctness in [RHB97]).

However, including any symbol in the alphabet that may affect synchronization may not be necessarily relevant for the guarantee, and abstract counterexamples are likely to be found spurious against components for reasons that are not related to the guarantee being checked. By including all the additional symbols, when an abstract counterexample is found, the added variables receive some random values and there is no reason to assume that these values are consistent with the component. The counterexample is then found spurious with respect to the arbitrary value given to an irrelevant variable and a refinement excluding that value is added, thus adding refinements irrelevant to the guarantee. In this work we approach this issue by presenting an alphabet refinement strategy for interface alphabets and evaluate it in the case studies.

Although all the related work applies CEGAR ideas, the use of a counterexample guided abstraction refinement scheme has not yet been considered for aspect-oriented programming, or reactive systems. In particular, by taking advantage of the knowledge about the interaction between event detectors and aspects or responses, a compositional technique will be presented.

2.4 Aspect-oriented programming

Aspect oriented programming (AOP) [KLM+97] allows expressing crosscutting concerns to the application in a modular way. Examples of crosscutting concerns are logging, persistence, exception management, and others. AspectJ (a popular AOP extension of Java) [KHH+01] defines a set of possible joinpoints - states where an aspect can be applied such as the states before or after method calls, fields read or set, or when an exception is to be thrown. For each aspect, pointcuts are expressions capturing a set of joinpoints and defining where the aspect should be applied, while each advice defines the actions to be applied at the joinpoints matched by its pointcut. There are three kinds of advice: before, after and around a joinpoint. For example, given the joinpoint of a method call, the advice can be executed before entering the method, after returning (either on success or due to an exception thrown) or instead the method call (around). Within around advice, the original joinpoint can be executed - possibly with different argument values and post-processing resulting states before returning to the base system where the joinpoint was matched.

Note that joinpoints given by a method call or a method execution could last an interval of time (e.g. since the method is called until it returns). In [MEY06], 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. Thus, advice no longer needs to indicate whether it is applied before, after or around a joinpoint, but it is simply activated when the point-in-time joinpoint occurs. The different region-in-time pointcuts and typical aspects written using the region-in-time approach (AspectJ’s approach) can be translated to be expressed in this model.

Aspects can be categorized according to the semantic transformation they make to the underlying system [Kat06]:
Spectative aspects gather information, but do not change control flow or the values of the variables non-local to the aspect.

Regulative aspects may change the control flow, but do not change the values of non-local variables.

Weakly-Invasive aspects may also change the values of the variables, as long as the returning state was already reachable in the underlying system.

Strongly-Invasive aspects are allowed to change the contents of the variables even when returning to states that were not originally reachable in the underlying system.

2.5 Events

In CEP, event producers provide events to an event processing network (EPN), where they are processed by event processing agents (EPAs) to detect more complex events which can eventually be consumed by event consumers. In reactive systems, responses are the event consumers that when an event of interest occurs, they react.

We distinguish between event occurrences and their detection. Following the definition of [EN10]: "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 entity or module that analyzes and announces complex events. We call these programming entities or modules that detect complex event occurrences event detectors. Primitive event occurrences, e.g. those occurring within the event producers, are immediately detected (without a separate detector).

Each complex event detector can be stateless or stateful. For example, event detectors that detect patterns or output aggregated information need to maintain an internal state. We consider event detectors as those in [BMAK11], given by aspect-like modules that react to lower-level events by either gathering information in locally defined fields, or processing the gathered information to trigger a detection announcement that can be used either by other event detectors, or by responses that react and may change the underlying system. Evaluation of event detectors can change local fields only and does not otherwise affect the underlying system besides the consequences of being detected. They can be hierarchically composed, thus obtaining complex events and abstracting primitive events.

Example. LowActivity (Figure 2.2) is an event detector in a Java-like notation that detects whenever a period of examining sales of a product ends and there have not been enough purchases. It is possible to see that this event declaration relies on RelevantPurchase, a lower-level event which exposes the purchases being considered. A possible response to this event could be applying a discount to the price of the product in order to encourage future sales.

Line 1 in the example declares the event detector name (LowActivity) and the exposed information when the event is detected (a product and the number of purchases of that product). Then, other event detectors or responses can react to this event being detected and use the exposed information.

In lines 4 and 7 there are two advice-like pieces of code, that react to lower-level events (RelevantPurchase and call(P.timeDone(...)) in order to aggregate information and trigger the event if necessary. The trigger keyword indicates that the event is being detected and other event detectors or responses can observe it and act accordingly.
In particular, the code between lines 4 and 6 reacts to when a relevant purchase has occurred and aggregates information in `purchaseInfo`. The code within lines 7 and 11 handles the actual detection. The event is triggered if and only if there is a call to `timeDone`, and the `if` guard holds (there have not been enough purchases). In addition, in every period of time (call to `timeDone`), the purchase information is reset (line 10).

Our later analysis is not on this code version, but rather on a state machine representation as explained in Sections 3.1 and 3.2.

2.6 Modular verification

In MAVEN [GKK10] each aspect is specified using the assume-guarantee model. An aspect specification is given by $(P, R)$ where $P$ represents what the aspect assumes about a system $S$ and $R$ represents what is guaranteed of the augmented system ($S$ with the aspect woven into it). Hence, each aspect can be verified on its own: given $P$ and $R$ formulas in linear temporal logic (LTL), the state machine representing all the possible paths that satisfy the assumption (i.e., the tableau) is built, the aspect is woven into it, and the resulting system is checked to satisfy the guarantee.

Weaving the aspect to the tableau of that assumption at the necessary locations implies adding the necessary transitions from the tableau of the assumption ($T_P$) to the aspect $A$ and back at the correct places. The obtained model ($T_P + A$) represents every possible path satisfying the aspect assumptions augmented with the aspect 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 aspect assumptions when the aspect is woven to it.

We will adapt this approach to verify responses.

2.7 Interference

Verifying each aspect in a library is usually not enough to guarantee that a set of the aspects in a library satisfies its expected behavior. One aspect may affect the behavior of another aspect, for instance when both change the same variables.

In MAVEN, a set of aspects is said to be interference-free if and only if whenever all assumptions are satisfied, and the aspects in the set are woven in any order $\{A_1, \ldots, A_n\}$, then all of the guarantees will...
be satisfied:

\[ S \models \bigwedge_{i=1}^{n} P_{A_i} \Rightarrow ((S + A_1) + \ldots) + A_n \models \bigwedge_{i=1}^{n} R_{A_i} \]

In order to satisfy non-interference, every pair of aspects \( A \) and \( B \) is shown to satisfy two rules:

**KP\(_{AB}\):** (Keep satisfying \( P \)) For any system \( S \) that satisfies \( P_A \land P_B \), the assumptions of both \( A \) and \( B \), when \( A \) is woven into \( S \) the obtained system \((S + A)\) must preserve the assumption of \( B \):

\[ S \models P_A \land P_B \Rightarrow S + A \models P_B. \]

**KR\(_{AB}\):** (Keep satisfying \( R \)) For any system \( S \) that satisfies the guarantee of \( A \) (\( R_A \)) and the assumption of \( B \) (\( P_B \)), when \( B \) is woven into \( S \) the obtained system \((S + B)\) must preserve the guarantee of \( A \):

\[ S \models R_A \land P_B \Rightarrow S + B \models R_A. \]

In proving these properties, the state machine of the assumption (the tableau of the LTL formula) is used to represent all base programs satisfying the conjunction on the left hand side of the implication, and the appropriate aspect is woven to the state machine to yield a state machine that should satisfy the formula on the right hand side of the implication.

The rules are sufficient to guarantee correctness and non-interference only if there are no joinpoints of other aspects within an aspect, or if it is reducible to sequential weaving semantics. This conditions do not necessarily hold in our new general setting.

An event-based system including multiple responses may present interference among responses, i.e. each response on its own behaves as expected but when considering all responses together the expected behavior is no longer achieved.

In the CEP literature the issue of possible interference among producers or consumers has not been addressed, yet when there are events produced within an application, the event consumers may have some logic that affects other consumers or producers, which may be of interest.
Chapter 3

Formal Setting

In this chapter we present the basic assumptions of the model considered, formalize the contents of event specifications based on [DK11] and explain how MAVEN [GKK10] is extended to consider event detectors as part of response assumptions.

3.1 Assumptions

There are two well known approaches for dealing with multiple events, CEP and Event Stream Processing (ESP) [Bas07]. ESP differs from CEP in that the emphasis is on handling event streams rapidly but usual ESP tools do not focus on complex events (i.e. do not deal with patterns, aggregations, etc.). In CEP, primitive events may not arrive in order, while in ESP they do. In this work we consider complex events, where the primitive events arrive in order. In particular, this assumption holds for systems where the events are produced by a given sequential system. If this is not the case, we can assume typical CEP preprocessing that buffers the event to handle out-of order arrival [BGAH07].

If there is no underlying system generating the events, in both CEP and ESP, sliding windows are usually used to aggregate information during a period of time. For instance, events can be considered within every window of 5 minutes, or within two event detectors determining start and end. Thus, even given a continuous stream of events, we can use these windows to discretize time and check at which locations events are evaluated and responses possibly activated.

There is literature on sampling events [BBD+02] in order to deal with performance issues. We assume that every event is considered (besides those explicitly removed by some event consumer - c.f. Subsection 4.6.3).

From all the given assumptions, we can observe that our event detection model can be represented by observing and analyzing primitive events at particular points to determine which complex event detectors are evaluated to gather information and to declare detection of the relevant event, and thus which responses should be applied.

We consider event detectors given by a finite state machine. In case the event detector is given by code, the event detector behavior can be extracted (with tools similar to [CCO02,CDH+00]). We consider event detectors as those in [BMAK11], that observe the system, gather information, and can be hierarchically composed.

We will assume the following about event evaluation:
• Event detector evaluation does not affect any variable external to the event declaration, and does
not affect the control flow besides by the responses they may cause to be activated.

• Event internal fields are only updated within the event declaration execution.

• There are no cycles in the event dependencies, i.e., an event cannot depend on itself being triggered
in the correct places or its own exposed information being correct.

• Event evaluation should terminate.

The first two properties may be checked by applying static analysis tools, adapting the tools presented
in [ATK09, CL02, RSB04, WTR07] which work for identifying spectative or observer aspects. The
dependency between events define an Event Dependency Graph [BMAK11], and cycle dependencies can
be checked by analyzing this graph. Termination may be represented by adding fairness constraints. Event
models should guarantee that the only fair paths are those that eventually reach the end of execution.

Another issue to consider is the event duration. 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.

Between the two possible semantics of primitive events (region-in-time or point-in-time - Section 2.4),
we will consider primitive events as those given by the point-in-time joinpoint model [MEY06]. 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.

Following the ideas of CEP, we assume that events are not detected within the evaluation of other
event detectors, or in CEP within the execution of an event processing agent (EPA). The main issue is
when the responses to these event detectors should be activated, since if they are within the original event
evaluation, this would imply that the event evaluation changes the underlying model (contradicting our
basic assumptions that distinguishes between event detectors and responses). This assumption can be
relaxed so that other event detectors can be evaluated within the original evaluation as long as those other
events do not activate any response (so that the event evaluation remains spectative). In such case the
proof obligations to be presented remain as they are.

3.2 Event specifications

The guarantees of a verified event detector can be assumed in order to prove more complex event detectors
or response guarantees. For example, LowActivity (Figure 2.2 in Section 2.5) is assumed to be correct by
a response applying discounts to increase sales of products with low activity.

In [DK11], we have observed that the properties to be satisfied by an event detector are:

1. The event detector is triggered in the correct places. This requires defining exactly which sequence
   of situations and contexts in the underlying system and previously verified events should cause the
current event to be triggered. The specification is in terms of event detections, exposed parameters and may as well include auxiliary variables representing gathered information.

Since the internal fields of an event detector are for internal gathering of information and are not accessible by other entities (they can only be observed when exposed by the triggered event), the specification can abstract and represent gathered information differently from the actual implementation.

2. The parameters exposed by the event detector satisfy the intended relations with the history of execution.

For example, the product exposed by LowActivity is indeed the one for which low activity has been detected.

The specification may refer to local properties (how lower-level events affect the internal state), or global properties such as an invariant of the system when the event is evaluated, or an LTL formula that holds along every path, while the implementation has a local point of view: for each lower-level event detected, how the internal state changes and when the event is triggered. Therefore, the specification may refer to a different set of lower-level event detectors than those appearing in the actual implementation, such as those involved in global guarantees.

Since the specification may differ from the actual implementation (i.e. different abstractions for the gathered information and different levels of lower-level event detectors involved) we also need to specify and verify event detectors. In particular, we will consider specifications and proof obligations that allow proof reuse in any system satisfying the event assumptions.

By observing different event detectors and possible relevant specifications, we have observed that useful specification languages are: 1) LTL formulas for temporal properties that are assumed or should be guaranteed by every path, including invariants or liveness guarantees 2) State machines or regular expressions to represent event detectors depending on sequences of lower-level events detected.

The abstraction of the data gathered by the event (and exposed) can also be represented either by LTL formulas or state machines indicating how lower-level event detectors affect the data.

The state machine part of the specifications can be translated into an LTL property (each path of a given finite state machine can be represented by an LTL formula), thus obtaining a uniform language for the different part of the specifications and allowing the use of traditional model checkers. In addition, we allow specifications to include auxiliary variables, which can aid in representing the current internal state of the event detector.

Formally, the specification of an event detector $ev$ is given by $E = (X_{ev}, X_{lower}, X_{internal}, P, R)$ where

- $X_{ev}$ contains the variables representing the current event detector and exposed information.
- $X_{lower}$ contains the variables representing lower-level event detectors and their exposed information.
- $X_{internal}$ contains variables to represent the internal state within the specification.
- $P$ is an LTL formula representing the assumption about the underlying system and lower-level events.
\( R \) is an LTL formula representing the guarantee when the detection of \( ev \) is considered in the system.

**Example.** To verify the correctness of \( \text{LowActivity} \), the event assumes the correctness of the lower-level events \( \text{timeDone} \) and \( \text{relevantPurchase} \). Among the internal variables we include a counter of the relevant purchases have occurred in the current period (\( cnt \)) and the predicate \( \#\text{RelPur} \leq B \) that holds if and only if the number of relevant purchases (reflected in the value of \( cnt \)) is lower than the upper bound. Then, the guarantees include:

1. \( G(\text{LowAct} \Rightarrow \text{TmDone}) \): If \( \text{LowActivity} \) is detected then \( \text{TmDone} \) must have occurred.

2. \( G((\text{TmDone} \land \#\text{RelPur} \leq B) \Leftrightarrow \text{LowAct}) \): \( \text{LowActivity} \) is detected if and only if \( \text{TmDone} \) has occurred and the predicate \( \#\text{RelPur} \leq B \) holds.

Note that these properties represent abstractions and conclusions about the event implementation.

### 3.3 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 example, we can have responses being activated when reaching a call to a method \( m \), or when returning from \( m \), or reacting to more complex event detectors. Differently from event detectors, responses can affect the execution flow and state of the system.

A response reacting to a complex event detector is in fact activated when the terminating primitive event that triggers the complex event occurs. For example, if there is a complex event maintaining the history of file operations and being detected when there is an attempt to read a previously closed file, then the response that reopens the file before the actual read reacts to the last event that triggered the response (the read attempt).

Our model considers one “thread” of execution, that is, if there are multiple responses that react to an event detector, one is activated at a time. Concurrently activated responses can be represented in our model as well, when their execution is serializable, i.e. there is an equivalent sequential execution of the responses.

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, and includes variables representing the event detectors and their exposed information.
- \( X_R \) 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 set of LTL formulas (Section 3.2) expressing the response assumption about underlying events.
- \( R \) is an LTL formula (on \( X_B \cup X_R \)) expressing the guarantee.

### 3.4 Modular Verification

Given that event evaluation does not affect the underlying system (besides by the event possibly being detected and causing a response to be applied), the evaluation stage can be modeled as occurring instantaneously, thus having at the same state all the evaluations of event detectors.

For example, the event detector indicating that “there is a product with low activity”, requires some internal calculations and checks. In the formal model we represent the evaluation as instantaneous and yet soundness is preserved. The internal calculations and checks are not visible to the underlying system or response and can be represented as stuttering states when abstracting to only variables external to the detector.

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 event evaluation end, thus obtaining the summary of changes in one state.

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 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 [RHB97] allows synchronization on part of the symbols, and interleaved behavior for the remaining symbols. 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 \( E_1 \| \ldots \| E_N \) where each \( E_i \) is the event specification of the event \( i \).

Since reactive systems consist of event detectors and responses, we extend the ideas presented in MAVEN [GKK10] for verifying aspects and adapt and change them to this more general setting while introducing new techniques. In our case, instead of modular verification of aspects, we will consider modular verification of responses.

Recall that considering responses similar to aspects, the notation \( T_P + A \) indicates the system that satisfies the assumption \( P \) with the response \( A \) woven into it (Section 2.6).

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

Our verification technique is based on model checking state machines derived from AspectJ-like code representing the responses. The method presented is for at most weakly-invasive aspects [Kat06], that is...
they may return to a different state of the underlying system as long as the state was already reachable from some other execution. This assumption is in order to make notation simpler, but the same ideas can be applied for the general case, as explained at the beginning of Subsection 4.5.5. We also assume here that there is no recursion, i.e. a response is never activated under its own execution flow, not even indirectly, although handling bounded recursion will be considered in the next chapter (Subsection 4.5.5).

3.5 Related work

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

Our techniques are then applicable to aspect-oriented programs with events, since they satisfy our basic assumptions.

There have been several related works incorporating events to AOP besides [BMAK11]. [AAC+05, VSCF05, BS06] deal with the first problem by using a restriction of the language of aspects to regular expressions, or treating sequences of events but still the composition of lower level events and independence between the joinpoint and the response are not treated. Douence et al. [DFS04] present a solution for these problems by allowing to share variables between crosscuts (pointcuts), preserving the history of execution and defining composition between aspects. However the crosscuts are still tightly related to the inserts (advices), and this restricts reusability. [MK09, Kat06, CL02] have identified the need of defining event aspects, spectators or observers that gather information but do not change the base system, although those aspects are allowed to print values.

Thus, the idea of defining aspects or events that collect information and are triggered when the collected information satisfies a certain property is natural in systems. These definitions are also useful for existing systems already defined using observers and spectators.

Events and observer aspects may seem trivial due to their spectative behavior on the base system. However, events incorporate the logic of when they must be triggered, and what information is exposed. They (and observer aspects) collect information from different possible sequences in the base system. This information may be collected from actions on the base system, and even subjected to some internal processing. Given that in the extension presented in [BMAK11] general aspects now respond to states in which events are detected, for later aspect verification it will be essential to assume that events are detected in the correct states and that they expose the expected information.

There is related work [CCO+04, JK04, LNAH+01] formalizing how events affect the state of the system by using state/event state machines. However, all these consider primitive events (or their boolean combination) and not a hierarchy of complex events. More related to our work are [RERA10, EPBS07] which address specification and verification of complex events. Both approaches allow defining a state machine for each event detector and understanding all the event detectors as their parallel composition. None of these approaches considers responses in the formal model (their focus is on the events).
Chapter 4

Responses within responses

Previous work on aspect modular verification considered aspects woven at certain locations described by formulas on primitive events. It was assumed that no other aspect was activated within an aspect. In our case we are considering the more general case where responses react to complex events detected and we allow responses to be activated within responses.

In Section 4.1 we present the running case study that will serve for several examples in this chapter. In Section 4.2 we explain the difference between sequential weaving and joint-weaving semantics (which allows responses to be activated within responses). In Section 4.3 we define augmented responses that allow applying verification when joint-weaving semantics are considered. In Section 4.4 we improve completeness by introducing internal assumptions which express what each response assumes about other responses that could be activated within its own evaluation and present verification proof obligation considering the new assumptions. Then, in Section 4.5 we address interference among responses, and in Section 4.6 we address cooperation among responses.

4.1 Running case study

The program listing in Figure 4.1 shows responses (using AspectJ-like syntax) to authenticate transactions on a website (Auth), to save a cookie on authentication success (SaveCookie), and to encrypt passwords (EncryptPwd), where the latter two are activated within the first response: usrPwdChecked is triggered whenever returning from a call to usrPwdExist(Usr, Pwd) and the return value is exposed by the variable success, and pwdRequested is triggered whenever returning from a call to requestPwd() and exposes the obtained password. The response EncryptPwd (activated whenever the event pwdRequested is detected), encrypts the returned value by the original call.

Note that SaveCookie does not affect the behavior of Auth. However EncryptPwd may change the (user,password) combinations found by Auth (because the password is now encrypted by the time the check is done). This interference is caused from the execution of EncryptPwd within Auth. In this section, we will consider a precise representation of response specifications which allows using formal verification techniques to detect such interference.

Intuitively, Auth is correct if when woven to any system, every time the event doTrans is detected, eventually Auth returns where the value of authed is true if and only if the user and password entered exist in the system. This field can be used later to allow or not different actions on the transaction. For instance, when not authenticated, the user may read but not write to the database. The response
response Auth
  when ( ) : doTrans( ){
    Usr u = requestUsr( );
    Pwd p = requestPwd( );
    authored = usrPwdExist(u, p);
  }

  event usrPwdChecked(Usr u, Pwd p, boolean success) = returning(usrPwdExist(u, p)) && returnValue(sucess)

  event pwdRequested(Pwd p) = returning(requestPwd( )) && returnValue(p)

response SaveCookie
  when (Usr u, Pwd p, boolean success) : usrPwdChecked(u, p, sucess){
    if (success)
      saveCookie(u, p);
  }

response EncryptPwd
  when (Pwd p) : pwdRequested(p){
    p.encrypt();
  }

Figure 4.1: Responses Auth, SaveCookie and EncryptPwd

SaveCookie is correct if when woven to any system, every time after the method usrPwdExist is completed and the user and password indeed exist, then a cookie is saved. Finally, EncryptPwd guarantees that when woven, the response encrypts the password obtained by requestPwd(). In addition, every response assumes that the event they react to is correct (detected at the correct places with the correct information).

In the temporal logic representations of the specifications, for any response A, PA represents the assumption about the underlying system and RA expresses the guarantee of the augmented system when the response is woven. The formal specifications of the responses are:

Auth:
  PA = true
  RA = G(doTrans ⇒ (F(usrRequested ∧ usr = U0) ∧ F(pwdRequested ∧ pwd = P0) ∧ F(retAuth ∧ authored ⇔ usrPwdInDB(U0, P0))))

SaveCookie:
  PSaveCookie = true
  RSaveCookie = G((usrPwdChecked ∧ Success) ⇒ F cookieSaved)

EncryptPwd:
  PEncryptPwd = true
  REncryptPwd = G(pwdRequested ⇒ F pwdEncrypted)

The atomic propositions used for the responses in Figure 4.1 are: doTrans, usrPwdChecked, pwdRequested for every state which matches the respective response detectors, retAuth to represent the return states of Auth, authored represents the truth value of authored which may be used elsewhere in the system, the atomic proposition usrPwdInDB indicates that the user and password exist in the database, Success represents the returned Boolean value of the call to the method usrPwdExist(), and cookieSaved represents that a cookie has been saved. Finally, pwdEncrypted indicates that the password has been encrypted.
In the guarantee of $\text{Auth}$, $U_0$ and $P_0$ represent the input values, and can be thought of as bound to a universal quantifier. This can be expressed in propositional temporal logic by substituting each user and password pair (the domain is finite).

### 4.2 Weaving semantics

Two weaving semantics can be considered in the context of Aspect-Oriented Programming:

**Sequential Weaving:** One aspect is woven to a base system at a time. Therefore, the first aspect is only activated at the joinpoints within the base system, but not at the joinpoints within other aspects that are woven later.

**Joint-Weaving:** Every aspect is woven at every joinpoint matching its pointcut whether it is within the base system or within some other aspects.

These weaving semantics can also be considered in the context of hierarchical event-based systems, representing whether responses are activated or not when there are events triggered by other responses. In addition, two main kinds of interference have been considered in the literature on aspects:

**Syntactic interference:** Two aspects interfere (syntactically) if they are potentially applied at a shared joinpoint, and there is at least one variable affected by both [WTR07, SFS06].

**Semantic interference:** Two aspects interfere if each satisfies its specification on its own, but this does not necessarily hold any longer when they are woven together. Related work [GKK10] on semantic interference assumed sequential weaving semantics or that one aspect cannot be activated while other aspect is executing.

In [DK12] we have extended the ideas of semantic aspect interference detection within the sequential weaving model to the joint-weaving model. Within hierarchical event-based systems, there are systems where events could be detected within a response, causing other responses to be applied (as with joint-weaving semantics). Therefore, although considering joint-weaving semantics introduces new challenges, it is very relevant to general reactive systems.

Given a response $A$ responding to a detected event $e$ woven to a system $S$ (including the information about the event detectors), $A$ is activated at every place where the event detector of $e$ is triggered within $S$. However, in the sequential weaving $(S + A) + B$, $B$ is woven into $S + A$, but if $B$ causes $e$ to be detected, this detection is not recognized by $A$ and $A$ is not activated at this point. In our example, sequentially weaving first $\text{EncryptPwd}$ and afterwards $\text{Auth}$, would not yield the correct model where the password is encrypted within $\text{Auth}$.

In the sequential weaving model each response is woven to the system in a certain order, e.g. given the responses $A$, $B$ where there is no precedence defined among the aspects, the possible results are $(S + A) + B$ or $(S + B) + A$. Note that in both cases added events may not activate the first woven response. We will use $S + (A, B)$ to denote the joint-weaving model of multiple responses, where any response can be activated within any other response.

We now present the definition of non-interference for joint-weaving, based on that in [GKK10].
**Definition 1.** Let \( \text{Responses} = \{A_1, \ldots, A_n\} \) be a set of responses depending on a set of event detectors \( \{E_1, \ldots, E_n\} \). Let \((P_i, R_i)\) be the specification of each response \( A_i \). Responses is said to be interference-free if and only if \( \text{OK}_{\text{Responses}} \) holds.

\[
\text{OK}_{\text{Responses}} \triangleq S \models \bigwedge_{i=1}^{n} P_i \Rightarrow (S + (A_1, \ldots, A_n))||E_1|| \ldots ||E_n \models \bigwedge_{i=1}^{n} R_i
\]

\( \text{OK}_{\text{Responses}} \) expresses that weaving all responses together with the joint-weaving model into any system \( S \) that satisfies the assumptions, satisfies the expected guarantees. In particular every event detector \( E_i \) is detected as it should (because of the parallel composition).

In the next sections every time the response is woven to a system (or its assumption) it will be assumed that the weaving is consistent with the detection of every \( E_i \) and omit the parallel composition. That is, \( S + A \) will denote \( (S + A)||E_1|| \ldots ||E_n \).

### 4.3 Augmented responses

To verify a response \( A \) when there could be some other response \( B \) possibly activated within \( A \), the model has to represent that \( B \) can change \( A \)’s current state, and that \( B \)’s termination may not be guaranteed. Thus, we build the augmented version of \( A \) \((A^+)\) by adding between every two states of the response, the state machine that allows any change and does not guarantee returning to the next state of \( A \).

Verifying a response \( A \) taking into account any other response that may be activated within \( A \) is achieved by the following proof obligation:

\[
S \models P_A \quad \Rightarrow \quad S + A^+ \models R_A
\]

Notice that considering this augmented model, verification will almost always fail for non-trivial response specifications (ruining the methodology’s completeness). The internal assumptions introduced below deal with this problem.

### 4.4 Internal assumptions

In order to restrict the behavior allowed for other responses, we add the notion of an internal assumption, which allows expressing for each response \( A \) what is expected of responses to be activated within \( A \). The general form of the internal assumption is \((\rho, \varphi)\) where \( \rho \) is a propositional logic formula (possibly) satisfied by a subset of all the states within \( A \)’s evaluation and \( \varphi \) is a temporal logic formula describing restrictions on the behavior of the possible responses activated at the states satisfying \( \rho \).

**Example.** A reasonable internal assumption for the response \( \text{Auth} \) in Figure 4.1 is for any inserted response (anywhere in \( \text{Auth} \), i.e. \( \rho = \text{TRUE} \)) to return without any exception thrown and preserve the values of the variables \( \text{usr}, \text{pwd} \) and \( \text{authed} \) (expressed by an LTL formula \( \varphi \)).

**Example.** A response \( A \) may initiate a transaction and do some actions, and then close the transaction. We want to avoid that during the execution of the transaction any possibly woven response \( B \) may perform commit for that transaction. Then the internal assumption of \( A \) is defined as: any response \( B \) to be
executed at any state within the transaction of $A$ should never perform commit until the return point is reached. Such an internal assumption is given by the pair $(\rho, \varphi)$ where $\rho : \text{inTrans}$ and $\varphi : \text{G-commit}$.

**Definition 2.** A response $A$ is augmented by its internal assumption $PI$ (noted as $A^{+PI}$) if and only if it has the internal assumption model woven into $A$.

The augmented response is built by adding for each state that could be an event detector of another response, a transition to the state machine that represents the internal assumption, and transitions from the final states of the internal assumption state machine to corresponding states of the first response.

From now on, if a responses has an internal assumption $PI$, we will say that its augmented version is the one augmented by $PI$.

Note that an internal assumption at the first state of the response, represents that there can be a response reacting to the same set of event detectors and being activated before the current response. Analogously, an internal assumption at the last state of the response, represents that there can be a response reacting to the returning state of the current response.

**Example.** The response and the augmented version of $Auth$ with the internal assumption preserving the values of $usr$, $pwd$, and $authed$ are presented in Figures 4.2 and 4.3, respectively. Note that * in the figures represents some arbitrary initial value, different for each field.

The states $s_0, \ldots, s_4$ appear both in the response model and in the augmented model of the response.

In the augmented response, there is a transition from the state $s_0$ to the states $t_0$ and then $t_1$. This means that if a response is inserted at this point of $Auth$ then the response can do anything ($t_0$) as long as eventually when reaching a returning state ($t_1$) the values of $usr$, $pwd$ and $authed$ are preserved, and hence it then executes the statement $Usr u = requestUsr()$ of $Auth$ (in state $s_1$).

It is assumed in this example that the fairness constraints are defined in order to avoid paths which stay infinitely in an “anything here” state.

Note that the augmented version of the response captures as well the assumptions on responses which react to the same event detector, hence, our technique will also handle interference when there are multiple responses reacting to the same set of event detectors. In particular in the example, $t_0$ and $t_1$ represent the assumption of any response that may respond to the detection of $doTrans()$ and might be executed before the first statement of the response $Auth$.

**Internal Assumption Defaults** There are default internal assumptions that can be defined. Typical examples are:

$PI_A = \text{NoResponse}$: If the guarantee of the response $A$ is sensitive to next state assertions ($X$) or real time constraints, $PI_A$ may assume that no response is woven during $A$’s execution.
$PI_A = \text{Spectative}$: It may be assumed from the environment that any response to be activated during the execution of $A$ is spectative.

$PI_A = \text{ReturningValuesPreserved} (V)$: Perhaps, any woven response $B$ may change things as long as when returning to the execution flow of $A$ the values of a certain set of variables ($V$) remain as they were before executing $B$. This is the internal assumption needed to preserve the values of $\text{usr}$, $\text{pwd}$ and $\text{authed}$ in $\text{Auth}$.

$PI_A = \text{Invariant} (I)$: Any response to be executed during $A$ may need to satisfy a certain invariant $I$ at every state.

$PI_A = \text{ReturnsOK}$: It can be assumed that every response executed within $A$ terminates without throwing exceptions.

$PI_A = \text{NoMandatoryProceed}$: According to the fine-grained joinpoint model in [MEY06] (Section 2.4), by default, all responses before a call proceed with the original call. However, if the skip statement is added, then the response skips the original call, i.e. the original call is not executed (possibly causing responses that would have been activated not to be activated any longer). $\text{NoMandatoryProceed}$ is an internal assumption that allows responses to skip proceeding with the cause of the event detection. If this internal assumption is absent, the skip statement is not allowed by responses possibly activated within the current response.

Internal assumptions can be combined using boolean operators. Moreover, general internal assumptions can override ($\oplus$) other internal assumptions at the locations matching $\rho$. Thus, combining a $\text{ReturningValuesPreserved}$ assumption with an internal assumption $(\rho, \varphi)$ may look like: $PI = \text{ReturningValuesPreserved} (V) \oplus (\rho, \varphi)$ where $\oplus \in \{\oplus, \land, \lor\}$.

Example. In Figure 4.4, the response LogDB exhibits the use of an internal assumption as explained above. In this case:

$$PI_{\text{LogDB}} = \text{ReturningValuesPreserved} (\text{msg}) \land (\text{inTrans}, G\neg\text{commit})$$

expresses that any other response $B$ to be executed during LogDB while in a transaction should not
commit that transaction. The intersection of assumptions guarantees that as well every response to be woven preserves the value of \( msg \).

Verifying responses including internal assumptions is now achieved with the following proof obligation:

\[
S \models PE_A \implies S + A^{+PI_A} \models RA
\]

\( PE_A \) represents \( A \)'s external assumption (\( A \)'s assumption about the underlying system), and \( PI_A \) represents \( A \)'s internal assumption.

We can now prove that \( Auth \) satisfies its guarantee by weaving the augmented response in Figure 4.3 to the tableau of \( Auth \)'s external assumption (TRUE).

Note that we could check first that the response on its own is correct (Section 3.4), and then check the augmented version of the response with this new proof obligation when considering a subset of responses within a library. However, the new proof obligation correctness is more general: it allows other responses to be woven as well (as long as there is not interference) and implies the proof obligation for correctness of the response on its own. Therefore, from now on we will only consider response correctness by checking whether it is correct with respect to its specification augmented with its internal assumption.

### 4.5 Interference

We now introduce the proof obligations to analyze interference when there could be responses activated within responses.

When several responses are jointly woven into a system \( S \), all of them correct with respect to their specification, we intend to guarantee non-interference. In order to achieve this, a set of rules is presented. If a library of responses satisfies all the rules, then the library is interference-free, otherwise there may be interference. The rules that responses must satisfy in order to guarantee non-interference are now presented. For every pair of responses \( A \) and \( B \):

\[
OK_{PI_{AB}}: \quad S \models R_A \land PE_B \implies S + B^{+PI_B} \models PI_A
\]

In a system where \( A \) has already been woven (therefore, \( R_A \) holds) any other response (\( B \)) either is not activated within \( A \) or satisfies \( A \)'s internal assumption.
KW_{B}(\varphi): \text{(Keep after weaving) For } \varphi \in \{PE_{A}, PI_{A}, RA\}:

\[ S \models PE_{B} \land \varphi \implies S + B^{+PI_{B}} \models \varphi \]

The rule applied to the different values of \( \varphi \) guarantees preserving the different parts of \( A \)'s specification.

We will denote the KW rules applied to the external and internal assumption, and guarantee of \( A \) as \( KW_{B}^{A} \).

In order to guarantee non-interference, all the rules presented must be satisfied by every pair of responses. In the next sections we explain in more detail each rule.

4.5.1 Rule \text{OK}_{PI_{AB}}

As mentioned above, rule \text{OK}_{PI_{AB}} expresses that every response must satisfy the internal assumptions of other responses. The internal assumption of a response \( A \) determines what is expected of responses that execute during \( A \).

In general, an augmented response \( B^{+PI} \) satisfies \( A \)'s internal assumption \((\rho, \varphi)\) if and only if: for every execution \( \pi \) of \( B \) that starts from a state in \( A \) which satisfies \( \rho \) and matches \( B \)'s event detector, \( \pi \) satisfies \( \varphi \).

\text{Example.} In Example (Trans), \( B^{+PI} \models PI_{A} \) with \( PI_{A} = (\rho, \varphi) \) as presented above if and only if for every activation of \( B \) in \( A \) where \( A \) is in a transaction, \( B^{+PI} \) satisfies \( G^{\neg commit} \).

Checking that a response satisfies the internal assumptions of another response may involve model checking or syntactic checks, depending on the internal assumption.

Now, we present the satisfiability conditions of default internal assumptions.

\text{Definition 3.} An augmented response \( B^{+PI} \) satisfies the default internal assumptions of \( A (PI_{A}) \) - noted as \( B^{+PI} \models PI_{A} \) - if one of the following conditions hold:

1. \( B \) is not activated within \( A \).

   Note: to check whether \( B \) could be activated within \( A \), we verify whether \( A \) satisfies the guarantee: \( G^{\neg (in_{A} \land ev_{B})} \). Then, \( B \) might be activated within \( A \) if and only if the formula is not satisfied.

2. If \( PI_{A} = Spectative \) and \( B \) is activated within \( A \), then all the possible augmented executions of \( B \) from activations of \( A \) are spectative. In terms of temporal logic: \( B^{+PI} \models G (V = V_{0}) \) where \( V \) are all the variables that are not local to \( B \) and \( V_{0} \) represents their original values before \( B \) is executed.

3. If \( PI_{A} = ReturningValuesPreserved (V) \) and there is an event detector activating \( B \) in \( A \), then all possible augmented executions of \( B \) from these detections preserve the values of the variables in \( V \) at the returning state. In terms of temporal logic: \( B^{+PI} \models G (ret_{B} \Rightarrow V = V_{0}) \).

4. If \( PI_{A} = Invariant (I) \) and there is a location where \( B \) should be activated within \( A \), then all possible augmented executions of \( B \) from these locations satisfy the invariant at every state. In terms of temporal logic: \( B^{+PI} \models GI \).
5. If \( PI_A = ReturnsOK \) and there is a location where \( B \) should be activated within \( A \), then all possible augmented executions of \( B \) from these locations reach a returning state without throwing any exception. In terms of temporal logic: \( B^{+PI} \models F(ret_B \land \neg例外_被扔) \)

6. If \( NoMandatoryProceed \notin PI_A \) and there is an activation of a response \( B \) activated when a call occurs, or a method starts its execution within the control flow of \( A \), then \( B \) should not have a skip statement for every execution path in the augmented model of \( B \) starting from activations of \( A \).

**Example.** Considering the program listing in Figure 4.1 and the rule \( OK.PI_{AB} \): the augmented version of EncryptPwd should satisfy the internal assumptions of Auth to prove that these responses do not interfere. The specifications of the responses are now extended to include the following internal assumptions:

- **Auth:**
  \[ PI_{Auth} = ReturnsOK \land ReturningValuesPreserved(usr, pwd, authed) \]

- **EncryptPwd:**
  \[ PI_{EncryptPwd} = Spectative \]

  The actual interference will be detected in this example when evaluating \( OK.PI_{Auth,EncryptPwd} \). In this case EncryptPwd does not satisfy the internal assumption of Auth of preserving the value of the password.

### 4.5.2 Rule \( KW_B(\phi) \)

Rules \( KW_B(PE_A) \) and \( KW_B(R_A) \) are the extensions of the rules described in Section 2.7 \( KP_{AB} \) and \( KR_{AB} \), now considering possibly inserted responses.

Rule \( KW_B(PI_A) \) expresses that \( A \)'s internal assumption is preserved after weaving \( B \). This is particularly useful for the case when multiple responses could be woven within another response. Each may satisfy the internal assumption but this may no longer hold when woven together.

Moreover, given that the conditions for checking \( OK.PI_{AB} \) and \( KW_B(R_A) \) are the same, in certain cases both rules can be considered together. However, in several situations the model is smaller when checking both properties separately.

### 4.5.3 Partial guarantees

If we attempt to apply the previous interference rules to the responses in our running case study, other completeness issues arise. For example, although there is in fact no interference between Auth and SaveCookie, when the rule \( KW_{Auth}(R_{SaveCookie}) \) is checked, it will fail because of the new location where SaveCookie should be applied and isn’t.

\[ S \models P_{Auth} \land R_{SaveCookie} \Rightarrow S + Auth^{+PI_{Auth}} \models R_{SaveCookie} \]

To improve completeness when there are event detectors of other responses within responses, we extend the specification of a response to include a distinction between global guarantees and local ones. Recall that in our running case study, for all three responses, the guarantee has the form \( G(evDet \Rightarrow expected_{behavior}) \), which represents a local behavior at each event detector. We will use this to identify
local and global guarantees. Thus, local guarantees (noted $RL$) are those properties that must be satisfied at each detector activating the response, while global guarantees (noted $RG$) are global properties not connected to being at the location where the event is detected.

The local guarantee can express properties both for each response that starts executing because the current detector has been triggered, or properties that should hold each time the response has finished executing. Then, for a response $A$, $RL$ is the conjunction of formulas of the form:

$G(ev_A \Rightarrow \varphi)$: Every time the detector of $A$ is triggered, $\varphi$ should hold. Note that $\varphi$ is not necessarily a state property, but rather a temporal logic formula. $\varphi$ is a formula expressing what $A$’s execution guarantees.

In particular, guarantees of the form $G(ret_A \Rightarrow \phi)$ expressing what is expected at the end of each execution of $A$ can be translated to $G(ev_A \implies (\neg ret_A W (ret_A \land \phi)))$ which has the form presented before of the property to be satisfied at each event detector activating the response.

We now proceed to define a partial guarantee to express what must be satisfied when there are responses not activated at every triggering of their event detector. These do not follow the joint-weaving semantics, but are possible under sequential weaving.

In the model, we assume that weaving a response $A$ to a system $S$ adds a label $act_A$ to those states in $S$ where $A$ is activated because of its event detector.

**Definition 4.** Let $RL_A = \bigwedge \psi_i$. For each $\psi_i = G(ev_A \Rightarrow \varphi)$ we define $\tilde{\psi}_i = G((ev_A \land act_A) \Rightarrow \varphi)$. Then the partial local guarantee of $A$ is given by $\tilde{RL}_A = \bigwedge \tilde{\psi}_i$.

$\tilde{RL}_A$ is based on the local guarantee but including the atomic proposition that identifies whether a response is activated when its event detector is triggered, so it will be satisfied even when there are locations where it is not activated (even if the event detector has been triggered). When all responses are activated at all locations where their event detectors are triggered, the original local guarantee is satisfied.

**Definition 5.** The partial guarantee of $A$ denoted $\tilde{R}_A$ is given by $(\tilde{RL}_A, RG_A)$.

Note that for any system $S$, $S \models R_A \Rightarrow S \models \tilde{R}_A$. Moreover, if $S \models \tilde{R}_A$ and $A$ is activated at all its detectors in $S$, then $S \models R_A$. The interference proof obligations now considering partial guarantees change as follows:

**OK.PI$_{AB}$:** $S \models \tilde{R}_A \land PE_B \Rightarrow S + B^{+PI_B} \models PI_A$

We require now that if the base system satisfies $A$’s partial guarantee and $B$’s external assumption, weaving $B$ satisfies $A$’s internal assumption.

**KW$_B(\varphi)$:** (Keep after weaving) For $\varphi \in \{PE_A, PI_A, \tilde{R}_A\}$: $S \models PE_B \land \varphi \Rightarrow S + B^{+PI_B} \models \varphi$

The rule now requires that the partial guarantee of $A$ is preserved.

Note that even though the rules only imply that the partial guarantee is preserved, eventually all responses will be activated at all the locations where their event detectors are triggered, hence the partial guarantee will imply the response guarantee.

Now, if the responses of Figure 4.1 are checked taking the partial guarantee of SaveCookie as in the following formula partial guarantee, every check is satisfied proving non-interference.
\[
\overline{RL_{\text{SaveCookie}}} = G((\text{usrPwdChecked} \land \text{ret_val} = \text{Success} \land \text{act}_{\text{SaveCookie}}) \Rightarrow F \text{ cookieSaved})
\]

### 4.5.4 Steps for each response added

Our method predicates that responses can be given a generalized assume-guarantee specification, where the underlying system, environment and event detectors are assumed to satisfy the response assumption, and the augmented system with the response should satisfy its guarantee. This form of specification allows building a library of verified responses where possible interference has been analyzed in advance. Then, this library may be used in any system that satisfies the properties that the responses assume, and the guarantees of the responses hold without the need to perform any additional checks.

If a set of responses \( \{A_1, \ldots, A_{n-1}\} \) has been proven to be correct with respect to their specification and without interference, when adding a new (verified) response \( A_n \), then we need to check that \( OK_{PI_{A_n}} \), \( KW_{A_n}^A \), and \( KW_{A_n}^B \) are satisfied for all \( 1 \leq i \leq n-1 \) (using the \( act_A \) atomic proposition for local guarantees).

If some of these checks do not succeed, then we save this information (including which proof obligation failed) so that the user knows which responses cannot be activated together.

When building a library of \( n \) responses we must do: \( n \) checks for the \( OK \) rule (every response satisfies its specification), \( n^2 \) for each of the rules \( OK_{PI}^+, KW_B(\varphi) \). In several cases checking \( KW_B(PI_A) \) does not require model checking but perhaps uses static/syntactic analysis to identify the detection of events, check whether an response \( B \) satisfies an invariant or \( B \) is spectative. All these checks are done as the library is constructed and then a set of interference-free responses can be used for any system that satisfies all response assumptions.

### 4.5.5 Soundness

In this thesis we treat weakly-invasive responses [Kat06], where control is returned after a response activation to a state which existed in some execution of the original system. In [KK09], verification is shown for strongly-invasive aspects, by adding an assumption about the base system states previously unreachable that now can occur in the woven system after the response ends its execution. A relatively complex modular verification technique is given that treats sequential weaving without events being detected within the response. The treatment here can also be applied to that technique, both for each aspect on its own and for the rules to detect interference.

We have assumed that the responses treated are never activated under their own execution flow, i.e. there is no recursion. However, the technique as defined already allows sound verification of responses that include a bounded number of possible activations within their own execution flow. In particular, the internal assumption allows other responses to be activated or not within the current response. So the “not activation” of itself would represent the base case, while the activation is the recursive step.

An example of this is the login response that reacts whenever there is an attempt to apply some transaction or a previous failed login (bounded by a predefined number of possible failed attempts). The login response shows a dialog for the user to enter his/her authentication information, and when clicking login (still under the login response execution flow), the response is activated again (with the incremented number of attempts).
Allowing (unbounded) recursion introduces the problem of analyzing termination and liveness properties would not necessarily hold.

The aspect category (e.g., whether it is weakly-invasive) and recursion absence can often be checked by already existing techniques. In [ATK09], dataflow techniques were presented to detect aspect categories. To guarantee no recursion a dependency graph can be built and analyzed to check that no aspect depends on itself.

We now show the soundness of the rules in order to guarantee non-interference of a set of responses that satisfies the necessary conditions.

First we prove how the augmented versions of the responses satisfy the partial guarantees (Subsection 4.5.3) when their preconditions initially hold. Secondly we show that this also holds for the original responses and the full guarantee when considering joint-weaving semantics and all responses are woven.

We show soundness when including both internal assumptions and partial guarantees.

**Lemma 1.** Let \( \{A_1, \ldots, A_n\} \) be a set of responses such that all the checks in Subsection 4.5.4 have been applied and all assertions have been proven to hold. Then, for any system \( S \) such that \( S \models \bigwedge_{i=1}^{n} PE_i \), \( S \) with all the augmented responses woven satisfies their partial guarantees, i.e.

\[
S + \left( A_1^{+PI}, \ldots, A_n^{+PI} \right) \models \bigwedge_{i=1}^{n} \tilde{R}_i
\]

**Proof.** By induction on the number of responses in the set.

- **Base case:** When adding one response \( A \) to the system \( S \) which satisfies \( PE_A \), from \( OK_A \), \( S + A^{+PI_A} \models R_A \). Then in particular, \( S + A^{+PI_A} \models \tilde{R}_A \).

- **Inductive step:** We assume by inductive hypothesis that for any system \( S \) such that \( S \models \bigwedge_{i=1}^{n-1} PE_i \), then \( S + \left( A_1^{+PI}, \ldots, A_{n-1}^{+PI} \right) \models \bigwedge_{i=1}^{n-1} \tilde{R}_i \) and we want to see that for any system \( S \) such that \( S \models \bigwedge_{i=1}^{n} PE_i \), then \( S + \left( A_1^{+PI}, \ldots, A_n^{+PI} \right) \models \bigwedge_{i=1}^{n} \tilde{R}_i \).

Given a system \( S \) such that \( S \models \bigwedge_{i=1}^{n} PE_i \), then in particular, \( S \models \bigwedge_{i=1}^{n-1} PE_i \) and by the inductive hypothesis

\[
S + \left( A_1^{+PI}, \ldots, A_{n-1}^{+PI} \right) \models \bigwedge_{i=1}^{n-1} \tilde{R}_i
\]

First, we need to see that \( A_n \)'s assumption still holds. From \( KW_{A_i}(PE_i) \) the assumption of \( A_n \) is preserved as other responses are woven to the system. Hence, \( S + \left( A_1^{+PI}, \ldots, A_n^{+PI} \right) \models PE_n \).

Then, when weaving \( A_n^{+PI} \) to \( S + \left( A_1^{+PI}, \ldots, A_{n-1}^{+PI} \right) \), the conjunction \( \bigwedge_{i=1}^{n-1} \tilde{R}_i \) is preserved from \( KW_{A_n}(\tilde{R}_i) \) and for those places where the \( A_n \) is woven in the execution of a response \( A_i \), the correctness is preserved from \( OK_{A_n}^{+PI} \) (the previously considered responses were shown correct), \( OK_{A_n}^{+PI} \) (the new response satisfies the internal assumptions of the previous responses and \( KW_{A_n}(PI_i) \) (the internal assumptions already satisfied keep being satisfied). Therefore,

\[
S + \left( A_1^{+PI}, \ldots, A_n^{+PI} \right) \models \bigwedge_{i=1}^{n-1} \tilde{R}_i
\]
Weaving \( A_n \) may add detectors of already woven responses, but these paths are already considered in the augmented version of \( A_n \), and due to \( OK_{A_n}, OK_{PI_{A_n,A_i}} \), and \( KW_{A_i}(PI_n) \) the guarantee of \( A_n \) is also preserved.

Therefore \( S + (A_1^{PI}, \ldots, A_n^{PI}) \models \bigwedge_{i=1}^{n} \tilde{R}_i \).

The lemma shows that if all the conditions hold then weaving all the augmented versions of the responses is interference-free. The next theorem uses this lemma in order to prove that if we have established that all the augmented versions of the responses are interference-free then, in particular, there is no interference when considering the resulting system with the (not augmented) responses woven.

**Theorem 1.** Let \( \{A_1, \ldots, A_n\} \) be a set of responses such that all the checks in Subsection 4.5.4 have been applied and all assertions have been proven to hold. Then \( \{A_1, \ldots, A_n\} \) is interference-free. That is, for any system \( S \) such that \( S \models \bigwedge_{i=1}^{n} PE_i \), then \( S \) with all the responses woven satisfies their guarantees, i.e.

\[
S + (A_1, \ldots, A_n) \models \bigwedge_{i=1}^{n} R_i
\]

**Proof.** From Lemma 1, for any system \( S \) such that \( S \models \bigwedge_{i=1}^{n} PE_i \) it holds that

\[
S + \big( A_1^{PI}, \ldots, A_n^{PI} \big) \models \bigwedge_{i=1}^{n} \tilde{R}_i
\]

In particular, \( S_{Responses}^+ = S + \big( A_1^{PI}, \ldots, A_n^{PI} \big) \) is an over-approximation of \( S_{Responses} = S + (A_1, \ldots, A_n) \). This is due to \( PI \) being the assumption of the responses that can be woven within a response, and \( S_{Responses} \) including only those that are actually woven within a response. That is, every path in \( S_{Responses} \) is a path in \( S_{Responses}^+ \). Given that all \( R_i \) are formulas in LTL, \( S_{Responses}^+ \models \bigwedge_{i=1}^{n} \tilde{R}_i \Rightarrow S_{Responses} \models \bigwedge_{i=1}^{n} \tilde{R}_i \). Moreover, given that all responses are already woven, then all responses are activated when their event detectors are triggered, and hence: \( S_{Responses} \models \bigwedge_{i=1}^{n} \tilde{R}_i \Rightarrow S_{Responses} \models \bigwedge_{i=1}^{n} R_i \).

Theorem 1 shows that this procedure is sound under the given assumptions. However, it is not complete. In particular, modularity affects completeness: there could be sets of responses which are interference-free but this cannot be shown with the assumptions and guarantees defined. That is, there may be two responses \( A \) and \( B \), both correct with respect to their specification and when woven together there is no interference, but the rules fail because the assumption or guarantee are not preserved in an intermediate state of building the augmented model.

The main advantages of this interference detection process is that it is modular, it provides flexibility to different external and internal assumptions, and is also used to prove the correctness and non-interference of cooperating responses (described in the next section).

### 4.6 Cooperation

Cooperation is tightly related to modularity: for example, a response \( A \) may assume a property to be satisfied to achieve its goal, and when this property is not guaranteed by the underlying system, some
other response $B$ may be the one satisfying that property. Alternatively, a response $A$ may rely on some other response $B$ achieving part of the overall expected functionality of $A$.

The first kind of cooperation refers to satisfying all the external assumptions of the responses and will be analyzed in Subsection 4.6.1. The second kind refers to responses collaborating when activated within other responses (thus satisfying an internal assumption) and will be analyzed in Subsection 4.6.2.

### 4.6.1 External assumption cooperation

We will consider the following example to illustrate this kind of cooperation:

**Example.** The response `EncryptPwd` (Figure 4.5) that encrypts the password being sent from a registration form may assume that passwords provided are of a minimum length. Systems not satisfying this constraint on their own can include the response (`EnsureCorrectLengthPwd`) that guarantees passwords of a minimum length.

EnsureCorrectLengthPwd:

\[
P_{\text{EnsureCorrectLengthPwd}} = \text{true} \\
R_{\text{EnsureCorrectLengthPwd}} = G (to \_ be \_ sent \Rightarrow pwdLength \geq M)
\]

EncryptPwd:

\[
P_{\text{EncryptPwd}} = G (to \_ be \_ sent \Rightarrow pwdLength \geq M) \\
R_{\text{EncryptPwd}} = G (to \_ be \_ sent \Rightarrow F (sent \land encrypted \land pwdLength \geq M))
\]

Note that another way to represent this is by `EncryptPwd`’s assuming (as an internal assumption) that in its first state $to \_ be \_ sent \Rightarrow pwdLength \geq M$. Cooperation related to internal assumptions will be explained in Subsection 4.6.2. However, there are cases in which external assumption cooperation cannot be translated to internal assumption cooperation, for example, when the assumption refers to a property that must hold along the path (and not only at the states within the response).

Let $rs \subseteq RS$ be the minimal set of responses required for the system $S$ where $RS$ is the set of all available responses, and $ps$ the set of response assumptions not satisfied by $S$. Then we want to know if there is a non-interfering subset of $RS$ that includes $rs$ for which all the formulas in $ps$ are satisfied and in which order responses activated at the same location should be applied so that each satisfies its guarantee.

**Example.** In our example, there are systems that may not necessarily satisfy that the password received by `EncryptPwd` is long enough. In such case our methodology learns that `EnsureCorrectLengthPwd`...
is also necessary and given that both react to `loginInfoEntered`, EnsureCorrectLengthPwd should be activated first so that EncryptPwd satisfies its guarantee.

**Methodology**

The input required to analyze cooperation is the set of available responses $RS$, the set of responses $rs$ that the user wants in the system, and the current system $S$. Using this information we can check the interference proof obligations and build the input to an SMT solver that answers whether there is a possible activation ordering of the responses so that all the assumptions are satisfied (and thus all the guarantees are satisfied when all the responses are woven).

The SMT instance will include for every response $r$ in $RS$: a boolean variable $x_r$, a boolean variable $x_r^p$ for each assumption $p$ of the response, and an integer variable $index_r$ representing the index in the sequence in which the current response should be woven. The value of $x_r$ for every $r$ in $rs$ is true. We also add the constraint: $x_r \rightarrow \bigwedge_{p \in PE_r} x_r^p$, if a response is included all its assumptions should be satisfied.

The first step is to use our techniques to verify each response on its own. Then, the response assumptions are partitioned according to whether they are satisfied by the given system. That is, $PE_A = PE_{A_{sat(S)}} \cup PE_{A_{unsat(S)}}$. In the SMT instance, the value of $x_r^p$ for every $p$ in $PE_{A_{sat(S)}}$ is true.

New proof obligations are required when there is cooperation, since we now allow other responses to satisfy part of the other response assumptions (instead of the base system). For every pair of responses $A$ and $B$, instead of checking $KW_B(PE_A)$ we check whether weaving $B$ preserves the assumption satisfied by the underlying system, i.e. $KW(PE_{A_{sat(S)}})$. The second step is to apply all the interference checks. From these, if there is some pair of responses $(A,B)$ that interferes according to any of the proof obligations given, then we add the constraint: $\neg(x_A \land x_B)$.

In the third step, for each unsatisfied assumption $(p \in PE_{A_{unsat(S)}})$, we can easily check whether there is some other response in $RS$ satisfying $p$. That is, we check for every other response $B$ in $RS$ whether $S \models P_B \Rightarrow S + B \models p$. Let $rp$ be the set of responses that satisfy $p$. Then, we add the following constraint to the SMT instance:

$$x_p^A \rightarrow \left( \bigvee_{B \in rp} x_B \land (activated\_together(A,B) \rightarrow index_B < index_A) \right) \quad (4.1)$$

The predicate $activated\_together(A, B)$ is true if there exists a path in which the event detectors of $A$ and $B$ could be detected together. This can be checked by checking the property $G \neg(evDet_A \land evDet_B)$ with the event detectors, and it is used to handle ordering with responses reacting to a same set of events. If both responses could be activated together, then the order may affect whether their assumptions still hold.

The last step is to check that any other response $C$ woven does not interfere with $B$ guaranteeing $A$’s assumption $p$, i.e.

$$S \models P_C \land p \quad \Rightarrow \quad S + C \models p \quad (4.2)$$

This proof obligation is independent from which is the response satisfying $A$’s assumption. In case it fails, we add the constraint $\neg(x_A \land x_C)$. 

35
Example. Given the responses in Figure 4.5, and a system wanting to include but not satisfying EncryptPwd’s assumption, we would build the SMT instance including the following constraint (the index part of the constraint is added since they may be activated at the same locations):

\[ x_{\text{EncryptPwd}} \rightarrow x_{\text{EnsureCorrectLengthPwd}} \land index_{\text{EnsureCorrectLengthPwd}} < index_{\text{EncryptPwd}} \]

Then, from the SMT solver we would obtain that the instance is satisfiable and a possible model is including both EncryptPwd and EnsureCorrectLengthPwd, and some example indexes satisfying \( index_{\text{EnsureCorrectLengthPwd}} < index_{\text{EncryptPwd}} \).

Note: To keep a feasible number of checks, in the third step we have checked whether there is some other response satisfying the assumptions not satisfied by the base system. However, there could be cases where one response on its own (and with its specification) does not satisfy it, but considering other responses it does. In future work different mechanisms can be analyzed to improve completeness of the methodology.

Soundness

Lemma 2. Given a system \( S \) and two responses \( A \) and \( B \), if

1. \( A \) and \( B \) satisfy their specification.
2. \( A \) and \( B \) do not interfere (including the cooperation proof obligation \( KW(P_E^{sat(S)}) \)).
3. \( S \) satisfies \( P_E^B \).
4. \( S \) satisfies \( P_E^A \setminus p \).
5. Weaving \( B \) to a system satisfying \( B \)'s assumptions satisfies \( p \).

Then, activating \( B \) before \( A \) at a location where both could be activated will cause the resulting model to guarantee both \( R_A \) and \( R_B \).

Proof. Let \( P_E^{sat(S)}_A \) be the subset of \( A \)'s assumptions satisfied by \( S \), i.e. \( P_E^{sat(S)}_A = P_A \setminus p \). From this and from items 2 and 5 of the lemma definition, we obtain that \( S \models P_E^{sat(S)}_A \land P_E^B \implies S + B \models P_E^A \).

From the correctness of \( A \), when \( A \) is woven to any system satisfying its assumption the augmented system satisfies \( A \)'s guarantee. Then, \( (S + B) + A \models R_A \). This expression represents that if both responses could be activated at a same location, then \( B \) would be activated first. Notice that every response includes (implicitly) its internal assumptions so there are no problems if new event detectors are activated or cease to be detected because of \( B \). Moreover, from non-interference, weaving \( A \) after \( B \) preserves \( B \)'s guarantee. Therefore,

\[ S \models P_E^{sat(S)}_A \land P_E^B \implies (S + B) + A \models R_A \land R_B \]

Using the non-interference of the proof obligation 4.2, we know that any other response woven will also preserve those assumptions and therefore the same ideas are applicable when multiple responses are involved.
Output

On success, the SMT solver provides a model satisfying the given constraints. From this model we can automatically build precedence rules to give to the compiler or weaver.

For every pair \((x^A_p, x^B_p)\) that both are true in the obtained model and \(A\) and \(B\) could be activated together, we add the precedence rule “\(B\) is to be applied before \(A\)”.

On failure, we can observe the constraints in the unsat core and understand whether interference, ordering, or missing cooperations affected the result.

4.6.2 Internal assumption cooperation

We will consider the following example to illustrate this kind of cooperation:

**Example.** A response (Copy) saves the objects of a certain class when necessary, trying initially to save them to a database and cooperating with another response (CopyToFile) when copying to the database fails. CopyToFile copies objects to an xml file. Either way, the objects are guaranteed to be saved.

**Copy:**

\[
\begin{align*}
PE_{Copy} &= true \\
PI_{Copy} &= EXISTS\_RESPONSE \\
G((call(DB\_saveObject) \land DBerror) \Rightarrow F\_savedObjectToFile) \\
R_{Copy} &= G(object\_Changed \Rightarrow F\_savedObjectToDB \lor savedObjectToFile))
\end{align*}
\]

**CopyToFile:**

\[
\begin{align*}
PE_{CopyToFile} &= true \\
PI_{CopyToFile} &= Spectative \\
R_{CopyToFile} &= G((object\_Changed \land \\
F((call(DB\_saveObject) \land DBerror)) \Rightarrow F\_savedObjectToFile))
\end{align*}
\]

The specification of Copy guarantees that when an object is changed, it is eventually copied, either to the database or to a file. Copy assumes \((PI_{Copy})\) the existence of a response that saves the object to a file if there is an error when trying to save an object to the database.

The specification of CopyToFile does in fact guarantee this.

\textit{EXISTS\_RESPONSE} represents the assumption that there must be a response satisfying the internal assumption. In the example, it cannot be an assumption of the base system since the call to \textit{DB\_saveObject} is within the response.

Recall that the internal assumptions are added at each location where the event detector of another response might be activated. To build the augmented model including an \textit{EXISTS\_RESPONSE} internal assumption, the original transitions are removed since the auxiliary response is assumed to exist.

Verification is applied similarly to Section 4.6.1, but more constraints are added. For every response \(A\) with an \textit{EXISTS\_RESPONSE} internal assumption, let \(rp\) be the responses satisfying that internal assumption. If \(rp\) is empty, then there is no subset of \(RS\) for which \(A\) can be included. Otherwise, the constraint to be added depends on the location of the internal assumption:
• if the EXISTS RESPONSE internal assumption is applied when within an internal state of the response, then the constraint is:

\[ x_A \rightarrow \bigvee_{B \in rp} x_B \]

(similar to the one in Section 4.6.1, but do not need to represent when activated at the same location.

• if the EXISTS RESPONSE internal assumption is applied at the initial state of the response, then the constraint is:

\[ x_A \rightarrow \bigvee_{B \in rp} (x_B \land index_B < index_A) \]

Being the internal assumption at the initial state represents that they could be activated together.

• if the EXISTS RESPONSE internal assumption is applied at the returning state of the response, then the constraint is:

\[ x_A \rightarrow \bigvee_{B \in rp} (x_B \land index_B > index_A) \]

This will guarantee that if they can be activated together, \( A \) will be activated before \( B \).

The last step is to add constraints representing those responses interfering with some \( B \) satisfying \( A \)'s EXISTS RESPONSE internal assumption. To do this we apply the constraints obtained from Equation 4.2, where \( p \) is the internal assumption.

Note that these same ideas could be applied when the specifications of the responses of Subsection 4.6.1 are as follows:

EnsureCorrectLengthPwd:

\[ P_{\text{EnsureCorrectLengthPwd}} = \text{true} \]
\[ R_{\text{EnsureCorrectLengthPwd}} = G(\text{to \_ be \_ sent} \Rightarrow pwdLength \geq M) \]

EncryptPwd:

\[ P_{\text{EncryptPwd}} = \text{TRUE} \]
\[ P_{\text{EncryptPwd}} = \text{EXIST}S_{\text{RESPONSE}}(\rho = \text{to \_ be \_ sent}, \varphi = pwdLength \geq M) \]
\[ R_{\text{EncryptPwd}} = G(\text{to \_ be \_ sent} \Rightarrow F(\text{sent} \land \text{encrypted} \land pwdLength \geq M)) \]

In this case we would find that whenever we want to include the response EncryptPwd and the base system does not satisfy the necessary assumption when reaching to be sent, then the response EnsureCorrectLengthPwd should be included as well.

### 4.6.3 Removing event detections

In this section, we show that the extended specification and verification also handle responses that eliminate the detection of an event activating another response. That is, the event is detected when the response is absent, but is not detected (and does not occur) when the response is added.

**Example.** In Figure 4.6 we show an example with removed detections. It is easy to see that the response AESEncr removes the detection activating DESSave. DES and AES are encryption algorithms.

In a system in which initially the DES encryption algorithm was used, the specification of the responses could be given by:
response Req&EncrPwd
  when( ) : usrEntered( ){
    enterPwd();
    encryptDES();
  }

response DESSave
  when( ) : DESEncrypted(){
    savePwd();
  }

response AESEncr
  when( ) : aboutToEncryptDES(){
    encryptAES();
    skip; //does not continue with the original call to encrypt DES
  }

Figure 4.6: Removed detections

Req&EncrPwd:

\[ PE_{Req&EncrPwd} = true \]
\[ PI_{Req&EncrPwd} = EXISTS\_RESPONSE \]
\[ G(\text{call}_\text{encryptedDES} \Rightarrow F\text{savePwd}) \]
\[ R_{Req&EncrPwd} = G(\text{usrEntered} \Rightarrow F(\text{enterPwd} \land F(\text{encryptedPwd} \land F\text{savePwd}))) \]

DESSave:

\[ PE_{DESSave} = true \]
\[ PI_{DESSave} = Spectative \]
\[ R_{DESSave} = G(\text{encryptedDES} \Rightarrow F\text{savePwd}) \]

AESEncr:

\[ PE_{AESEncr} = true \]
\[ PI_{AESEncr} = Spectative \]
\[ R_{AESEncr} = G(\text{call}_\text{encryptedDES} \Rightarrow X((G \neg \text{encryptedDES}) \land F(\text{encryptedAES}))) \]

That is, Req&EncrPwd takes care of requesting a password and calling the encryption algorithm and assumes the existence of DESSave, a response that guarantees that eventually the encrypted password is saved. It is possible to see the cooperation in the assertion \( PI_{Req&EncrPwd} \). For now, we concentrate on the guarantee that the password must be saved (not necessarily encrypted).

However, due to a security problem, it is decided to create a response such that around every call to the DES encryption algorithm it uses now the AES encryption algorithm. The guarantee of AESEncr indicates that every time encryptDES( ) is called, it guarantees that no password is encrypted using DES (\( G \neg \text{encryptedDES} \)), but now every call to encrypt the password is replaced by encryptAES(). Then, when the response AESEncr is checked with other responses to detect interference, given that AESEncr has no proceed and \( NoMandatoryProceed \notin PI_{Req&EncrPwd} \) the rule \( OK_{PI_{Req&EncrPwd},AESEncr} \) is not satisfied.

This interference removes the call to encryptDES( ), and hence the password is no longer saved.

Note that even if \( NoMandatoryProceed \) did belong to the definition of \( PI_{Req&EncrPwd} \), then we would detect the problem by applying the cooperation techniques: If we want to include both
Req\&EncrPwd and AESEncr, the SMT instance would include $x_{\text{Req}\&\text{EncrPwd}}$ and $x_{\text{AESEncr}}$. From the internal assumption of Req\&EncrPwd, we add the constraint $x_{\text{Req}\&\text{EncrPwd}} \rightarrow x_{\text{DESSave}}$ and from checking the proof obligation 4.2, the constraint $\neg(x_{\text{AESEncr}} \land x_{\text{DESSave}})$ is added (the response AESEncr interferes with DESSave satisfying Req\&EncrPwd’s internal assumption). The conjunction of all these constraints is clearly unsatisfiable and the problem is exposed. Thus, in both cases there is interference, and the interference-freedom checks, as expected, do not succeed.

4.7 Related work

There are several related works considering aspect interactions under joint-weaving semantics such as [MW10, WTR07, TBB04, KDS08, KF07].

The work in [MW10] does not provide proof obligations for verification or interference detection under this semantics, but extends the joinpoint model to better express aspect interactions. For example, by considering named advice, some other aspect can indicate that it is to be activated as long as a particular advice is not being currently activated.

In [WTR07], aspect dependencies are found using as a base an interprocedural Reaching Definitions Data-flow Analysis [NNH99]. These dependencies do not necessarily lead to semantic interference, possibly yielding false positives (i.e., it only detects cases of suspected interference), and the summary transfer functions imply analyzing a particular underlying system, instead of considering the aspects as an independent library.

In [TBB04], the issue of concerns affecting other concerns due to “Change of functionality, Inconsistent behavior and Composition anomalies” is mentioned. The models are analyzed to check direct conflicts, i.e. when two aspects have a shared joinpoint or one aspect can be activated within another aspect.

In [KDS08] the idea of internal assumptions is represented by Hoare-logic assertions that cross-cutting concerns must satisfy. This approach describes the acceptable state changes. In our approach we show that a general temporal logic formula or some syntactic check is satisfied instead of considering only a Hoare logic assertion where advice is woven and returns.

The work in [KF07] works with interfaces in temporal logic (CTL in their case), covers removed joinpoints due to the absence of proceed, but assumes that any advice restores the stack to the same state it had before the advice execution, not covering weakly-invasive aspects in general. To capture advice within advice, the states at which advice might apply must have an accurate interface. Knowing which are the states and what advice might apply affects obliviousness. This might also be a problem in our approach, especially for cooperation.

There is not much emphasis regarding aspect or response cooperation in the literature, although how the aspect ordering affects the resulting system has been considered in several related works [ARS09, SFS06, DSBA05, Kni09].
Chapter 5

Abstraction-Refinement Verification

In this section we present the abstraction-refinement technique that allows more feasible checking of the proof obligations presented in the previous chapter.

All the proof obligations presented are of the following form:

\[ S \models \varphi \implies (S + A^{+PI}) || E_1 || \ldots || E_n \models \psi \]

That is, we could build the tableau of the assumption, weave the augmented response, compose it with all relevant event specifications and check whether \( \psi \) is satisfied. However, the event hierarchy—where one event detector depends on many simpler event detectors—may include many events, and the model would turn out to be unreasonably large, making model checking unfeasible. Instead, we would like to have the response make only the necessary assumption about the event detectors on which it depends—an abstraction of the full properties of the events, so that the needed guarantee of the response can still be shown.

Therefore, the response assumption should include only necessary assumptions about the events it relies on to show a potential guarantee. This provides better understanding of what the response needs for each guarantee, and if an event definition changes, the affected responses can be easily identified.

Now, considering their assumptions about events, we present an abstraction refinement scheme to verify responses in hierarchical event-based systems.

5.1 Basic assumptions

In order to apply our approach, correct lower-level event specifications (Section 3.2), 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 event detectors is learnt by our technique.

Recall that event specifications are given by \( E = \langle X_{ev}, X_{lower}, X_{internal}, P, R \rangle \) (Section 3.2).

An initial abstraction of the event specification \( E \) that does not add any constraints is given by \( E' = E = \langle \emptyset, \emptyset, \emptyset, \text{TRUE}, \text{TRUE} \rangle \). We denote this as \( \text{TRUE}(E) \). That is, we do not add any of the vocabulary symbols of the event detector, nor any constraints obtained from the event assumption or guarantee.

By considering event specifications, we are abstracting from the implementation. Thus, the results of
our technique are sound (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. Still, there are advantages to using specifications instead of the actual implementation, as described in the introduction.

The technique to be described now, though explained for responses is also applicable for event detectors, since they can be viewed as observer responses.

### 5.2 Running example

As a running example, we consider the response that adds the helicopter mission in a Car Crash Crisis Management System (CCCMS) [KGM10]. This mission 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.

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.

Figure 5.1 shows a fragment of the library of event detectors relevant to the CCCMS example. The square boxes represent event detectors and the ellipse boxes represent system variables affected by event detectors. The main difference between these is that changes to a system variable may not trigger a response immediately, but further in the computation the response may check what was the last assignment of that variable. For instance at any given moment we can check whether there are helicopters available, depending on the history of helicopters that were sent and returned. The arrows represent dependencies including temporally (if \(e_i\) has occurred in the past) and non-occurrence (depending on another event detector not occurring or a state variable with value \(false\)). For instance, the event \(shouldSendHeli\) depends on the event \(problematicAccess\) being detected and on \(not\ \badWeather\) 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 the use case of [KGM10]), 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 (phoneCompanyValidated). Each of the boxes in the event dependency graph (complex event detector or system variable affected by detectors), has its specification regarding its detection and exposed
information. For example, the specification of \textit{badWeather} indicates that this variable is set whenever a snow storm starts or extreme turbulence is detected, and remains such until there is a weather update without bad weather causes.

The response to \textit{shouldSendHeli} is given by the following response expressing that when detecting that a helicopter should be sent to a certain location \textit{crisisLocation}, the actual mission of sending an helicopter to that location is added to all the missions to be performed.

\begin{verbatim}
response addHelicopterMission
when (Location crisisLocation) : shouldSendHeli(crisisLocation)
  allMissions.add(new SendHelicopterMission(crisisLocation));
\end{verbatim}

For this example, we consider the following response guarantee:

\begin{quote}
"If there is a crisis at a certain location (\textit{shouldGoToLocation}), but there is a snow storm (\textit{snowStorm}), the helicopter mission is not added (\textit{\neg HMAdded})." (5.1)
\end{quote}

We can express this in LTL by

\[ G((\textit{shouldGoToLocation} \land \textit{snowStorm}) \rightarrow \neg \textit{HMAdded}) \] (5.2)

Had we not used a CEGAR approach, model checking would be applied to the full model \((TP + M) || E_1 || \ldots || E_N\), including multiple irrelevant variables and transitions that make the transition relation difficult. We will ignore in this chapter the notation representing the augmented response, but it will be
assumed to be always included. 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.

5.3 Method

To avoid applying direct verification to the model in the previous section, the abstract model we consider is 
\((T_P + M) || E'_1 || \ldots || E'_N\) where \(E'_i\) is an abstraction of \(E_i\) - the assumption of the response \(A\) about \(E_i\), making the composition much simpler (at the first iteration \(E'_i = \text{TRUE}(E_i)\)). 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. In comparison to related CEGAR work, this kind of abstraction does not require an SMT solver or theorem prover to calculate the abstract transition relation, thus the model can be expressed simply as a finite state machine and checked by existing model checkers.

Given a system that satisfies the mentioned assumptions, a CEGAR-like algorithm can be applied (algorithm 5.1). 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 (Section 5.4), and (line 2) initialize \(E'\) with the empty set (every event specification abstraction is \(\text{TRUE}(E_i)\)). \(E'\) 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 \(R\) (Sec. 5.5). 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 9 we check whether the counterexample is due to the abstraction (spurious) or real (Sec. 5.6). If found spurious (line 10), refinements to avoid the current abstract counterexample are obtained (Section 5.10). Otherwise, the counterexample is real (line 12) and the CEGAR cycle ends.

Algorithm 5.1 Compositional CEGAR for Hierarchical Reactive Systems

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M, (P, R)): Response model and response partial specification</td>
<td>(\text{satisfied}?:) Indicates whether the response guarantee is guaranteed with the given assumptions so far</td>
</tr>
<tr>
<td>(S): Set ({E}): Event Specification Library</td>
<td>(E'): Event specifications’ abstraction</td>
</tr>
</tbody>
</table>

\[ S' = \text{“get subset of relevant events from } S\text{”} \]
\[ E' = \emptyset \]

\[ \text{while (True) do} \]
\[ \quad \text{modelToCheck} = (T_P + M) || E' \]
\[ \quad \text{if modelToCheck} \models R \text{ then} \]
\[ \quad \quad \text{satisfied}? = \text{True} \]
\[ \quad \quad \text{return} \]
\[ \quad \quad \text{else} \]
\[ \quad \quad \quad \text{spurious}? = \text{“check spuriousness using } S'\text{”} \]
\[ \quad \quad \text{if spurious}? \text{ then} \]
\[ \quad \quad \quad E' = E' \cup \text{“get spuriousness reasons”} \]
\[ \quad \quad \quad \text{else} \]
\[ \quad \quad \quad \quad \text{satisfied}? = \text{False} \]
\[ \quad \quad \quad \text{return} \]
\[ \quad \quad \text{end} \]
\[ \quad \text{end} \]
\[ \text{end} \]
At each step, the event specification abstractions (response assumption about the events) are refined by adding constraints to $P$ or $R$, and refining $X_{ev}$, $X_{lower}$, and $X_{internal}$ accordingly. Since $P_{Ev}$ is an abstraction of the event specifications, at every step any path in $E_1 \| \ldots \| E_N$ is a path in $P_{Ev}$.

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).

### 5.4 Relevant events

The input contains a library of event specifications. However, not every event may be necessary to check the response guarantee and we can automatically restrict the set to 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.

### 5.5 Verification

To apply verification, the model $(T_P + M) \| E'$ is built. $T_P + M$ is built as in Section 2.6. In each step, $E'$ represents partial information of the event specifications, that is, $E'_1 \| \ldots \| E'_N$. Therefore, building the state machine $(T_P + M) \| E'$ is done by including all the constraints of both $(T_P + M)$ and 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 5.2 shows the helicopter response model on its own and Figure 5.3 shows the response model after weaving it to its assumption (that the base system does not itself add the helicopter mission). The
variable \texttt{HMAdded} indicates whether the helicopter mission has already been added to \texttt{allMissions}. The response model indicates that whenever \texttt{shouldSendHeli} is detected, after that state the variable \texttt{allMissions} includes the helicopter mission. In the woven model (Figure 5.3), the system can remain at the initial state (performing actions irrelevant to our response) until \texttt{shouldSendHeli} is detected. At those locations the response is \textit{woven}, and at the return state the execution continues from the base system where it should 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.

In the given example, Property (5.2) is not initially satisfied: there could be a path where \texttt{shouldSendHeli} (which activates the response) is detected together with \texttt{shouldGoToLocation} and \texttt{snowStorm}, causing the response to be activated. The unexpected behavior is due to the initial overapproximation of the system that does not include (yet) the \textit{indirect} connection where both \texttt{shouldSendHeli} and \texttt{snowStorm} cannot be detected in the same state.

## 5.6 Checking spuriousness

As mentioned before (Section 2.3), contrary to previous work we allow the abstract version not to automatically include the concrete model alphabet. However, in this case 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 \texttt{Spec\_badWeather} and \( G (\text{shouldSendHeli} \rightarrow \neg \text{badWeather}) \) in \texttt{Spec\_shouldSendHeli}, then there cannot be a state satisfying \texttt{shouldSendHeli} \& \texttt{snowStorm}. However, if the abstract version of the response does not include in its alphabet \texttt{badWeather}, then the problematic state is consistent with each event specification (for each modular check there is an assignment of \texttt{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 5.2 which checks spuriousness for the non-included alphabet strategy (with alphabet refinement).

The first event specification of the sequence does not need to be composed with a previous event specification abstracted, thus \texttt{prevAbstractModel} is initialized as \texttt{True} (i.e., the model accepting every event specification).
path). Every other event specification \( E_i \) is composed with the Cone of Influence (COI) [CGP01] reduction of the previous model to the variables that may affect following events in the sequence. The Cone of Influence of a model consists of the set of variables that affect (directly or indirectly) \( \{ V_i \} \). At each step, we abstract the \( currModel \) (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 \( shouldSendHeli \)). Using the included alphabet strategy, if both events appear together in the counterexample, it is immediately found spurious, and only in a subsequent CEGAR cycle we obtain the connection between \( badWeather \) and \( snowStorm \). With the non-included alphabet strategy, \( badWeather \) does not belong to the initial alphabet. Then, in the first CEGAR cycle the counterexample is found spurious with the COI of \( shouldSendHeli \) and the specification of \( badWeather \). From this composition the necessary refinements will be obtained (Section 5.10): \( (shouldSendHeli \ implies \ not \ badWeather, \ and \ snowStorm \ implies \ badWeather) \) which prevent future counterexamples with \( shouldSendHeli \) and \( snowStorm \) in the same state.

**Event Ordering** In both approaches (included, non-included 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.

### 5.7 Modular approach - Correctness

This section is more technical and proves correctness of algorithm 5.2.

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 [CCG+04] based on the lemma [RHB97] in which 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 Section 3.4 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
We first present some auxiliary notations, definitions and propositions that we will use to prove the correctness of our approach.

**Notation.**

- For any path $\pi$ and alphabet $\Sigma$, $\pi \mid |_\Sigma$ represents the path projected to include only the symbols appearing in $\Sigma$.
- For any model $M$ and alphabet $\Sigma$, $M \mid |_\Sigma$ represents $M$ abstracted to the variables appearing in $\Sigma$.
- For any model $M$ and alphabet $\Sigma$, $\text{COI}(M, \Sigma)$ 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 $M \mid \text{COI}(M, \Sigma)$ and $[\pi]^M_\Sigma$ represents $\pi \mid \text{COI}(M, \Sigma)$. The superscript representing the model will be omitted when clear.

**Definition 6.** A path $\pi$ over an alphabet $\Sigma$ is consistent with a model $M$ over an alphabet $\Sigma$ if and only if there exists a path $\hat{\pi} \in M$ such that $\hat{\pi} \mid |_{\Sigma \cap \Sigma_M} = \pi \mid |_{\Sigma \cap \Sigma_M}$. $\hat{\pi}$ 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 5.7.1.** If $\pi$ is a path of a model $M$, then for an alphabet $\Sigma$, $[\pi]^M_\Sigma$ is a path of $[M]_\Sigma$.

Proposition 5.7.2 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 5.7.2.** Given a path $\pi$, two models $M_1$, $M_2$, and alphabet $\Sigma$, $\pi \in [M_1 || M_2]_\Sigma$ if and only if $\pi \mid |_{\Sigma \cap \Sigma_M} \in [M_i]_{\Sigma_M}$ 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 3.** Given $M_1$, $M_2$ two models and $M'_1$, $M'_2$ their respective overapproximations - not necessarily with the same alphabet, i.e. $\Sigma_{M'_1} \neq \Sigma_{M_1}$ - 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 $[M_1]_{\Sigma_{M_1} \cup \Sigma_\pi} || M_2$

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

48
Assuming $\pi$ is consistent with $M_1 \parallel M_2$, we want to see that $\pi$ is consistent with $M^{\text{abs}} \parallel M_2$.

The idea is to build the witness path $(\overline{\pi})$ that proves that $\pi$ is consistent with $M^{\text{abs}} \parallel M_2$ from the witness path $(\pi)$ of $\pi$ being consistent with $M_1 \parallel M_2$.

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

Since $\pi \in M_1 \parallel M_2$, then by the traces definition of composed components [RHB97], $\pi \upharpoonright \Sigma_{M_1} \in M_1$ and $\pi \upharpoonright \Sigma_{M_2} \in M_2$.

Since the alphabet $\Sigma_\pi \cap \Sigma_{M_1} \cap M_2 = \Sigma_\pi \cap (\Sigma_{M_1} \cup \Sigma_{M_2})$ contains the alphabet $\Sigma_\pi \cap \Sigma_{M_2}$, and $\pi \upharpoonright \Sigma_\pi \cap \Sigma_{M_2} = \pi \upharpoonright \Sigma_\pi \cap \Sigma_{M_2}$, then $\pi \upharpoonright \Sigma_\pi \cap \Sigma_{M_2} = \pi \upharpoonright \Sigma_\pi \cap \Sigma_{M_2}$.

Therefore, $\pi \upharpoonright \Sigma_{M_2}$ is a witness of $\pi$ being consistent with $M_2$.

We now show that $\overline{\pi} = \pi \upharpoonright \Sigma_{M_{\text{abs}}} \cup \Sigma_{M_2}$ (the witness of $\pi$ being consistent with $M_1 \parallel M_2$, restricted to the alphabet of $M^{\text{abs}} \parallel M_2$) is a witness for $\pi$ being consistent with $M^{\text{abs}} \parallel M_2$.

1. We first show that $\overline{\pi}$ is a trace of $M^{\text{abs}} \parallel M_2$, i.e. $\overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}} \in M^{\text{abs}}$ and $\overline{\pi} \upharpoonright \Sigma_{M_2} \in M_2$.

   a. $\overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}} = (\pi \upharpoonright \Sigma_{M_{\text{abs}}} \cup \Sigma_{M_2}) \upharpoonright \Sigma_{M_{\text{abs}}} = \overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}}$ (by definition of $\overline{\pi}$ and $\upharpoonright$).

   b. Since $\overline{\pi} \in M_1 \parallel M_2$, $\overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}} \cup \Sigma_{M_2}$ is a path in $([M_1 \parallel M_2]) \Sigma_{M_2} \cup \Sigma_{M_2}$ (by Proposition 5.7.1).

   c. Then, by Proposition 5.7.2, $(\overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}} \cup \Sigma_{M_2}) \cap COI(M_1, \Sigma_{M_2} \cup \Sigma_{M_2})$ is a path in $[M_1 \parallel M_2] \Sigma_{M_2} \cup \Sigma_{M_2}$ (recall that $[M_1 \parallel M_2] \Sigma_{M_2} \cup \Sigma_{M_2} = M^{\text{abs}}$).

   d. By definition, $(\overline{\pi} \upharpoonright \Sigma_{M_2} \cup \Sigma_{M_2}) \cap COI(M_1, \Sigma_{M_2} \cup \Sigma_{M_2}) = \overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}} \cup \Sigma_{M_2}$.

   e. From item a. and item d., $\overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}} = (\overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}} \cup \Sigma_{M_2}) \cap COI(M_1, \Sigma_{M_2} \cup \Sigma_{M_2})$.

   f. From the previous item and from item c., $\overline{\pi} \upharpoonright \Sigma_{M_{\text{abs}}}$ is a path in $M^{\text{abs}}$.

2. We now show that $\overline{\pi} \upharpoonright \Sigma_{n \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}} = \pi \upharpoonright \Sigma_{n \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}}$.

   a. $\overline{\pi} \upharpoonright \Sigma_{n \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}} = \overline{\pi} \upharpoonright \Sigma_{n}$ since the alphabet of $\overline{\pi}$ is $\Sigma_{M_{\text{abs}}} \parallel M_2$.

   b. $\overline{\pi} \upharpoonright \Sigma_{n} = (\pi \upharpoonright \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}) \upharpoonright \Sigma_{n} = (\pi \upharpoonright \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2})$ by the definition of $\upharpoonright$.

   c. $(\pi \upharpoonright \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}) \upharpoonright \Sigma_{n} = (\pi \upharpoonright \Sigma_{n}) \upharpoonright \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}$ by the definition of $\upharpoonright$.

   d. $\pi \upharpoonright \Sigma_{n} = \pi \upharpoonright \Sigma_{n} \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}$ since $\pi \in M_1 \parallel M_2$.

   e. $(\pi \upharpoonright \Sigma_{n} \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}) \upharpoonright \Sigma_{n} = (\pi \upharpoonright \Sigma_{n} \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2})$ from the previous item and $\pi$ being a witness of $\pi$ being consistent with $M_1 \parallel M_2$.

   f. $(\pi \upharpoonright \Sigma_{n} \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}) \upharpoonright \Sigma_{n} = \pi \upharpoonright \Sigma_{n} \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}$ since $\pi \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2} = \pi \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}$ (definition of $\cap$ and $M_{\text{abs}} \parallel M_2 \subseteq M_1 \parallel M_2$).

   g. From all the above, $\overline{\pi} \upharpoonright \Sigma_{n} \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2} = \pi \upharpoonright \Sigma_{n} \cap \Sigma_{M_{\text{abs}}} \cap \Sigma_{M_2}$.
The proof is applying the previous lemma inductively on the number of elements in the composition.

\[ \pi \] is a simulation relation such that \( \pi \) is obtained from the composition of \( M_1, \Sigma_{M_{abs}} \cup \Sigma_n \).

**Proposition 5.7.3.**

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 \( [M]_{\Sigma \cup \Sigma'} \).

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

1. \( \pi \in \Sigma_{M_1} \) is equivalent to restricting to the variables of \( M_1 \). By construction \( \pi_1 \in M_1 \), therefore \( \pi \in \Sigma_{M_1} \in M_1 \).
2. \( \pi \in \Sigma_{M_2} \) is equivalent to considering \( \pi \in \Sigma_{M_2} \). Since \( \pi \) is the witness of \( \pi \) being consistent with \( M_{abs} \| M_2 \), \( \pi \in \Sigma_{M_2} \) belongs to \( M_2 \).

\( \pi \in \Sigma_n \cap \Sigma_n \): Both sides of the equation are equal to \( \pi \), thus the paths obtained by each restriction are equal.

**Corollary 5.1.**

Given \( n \) components \( M_1, \ldots, M_n \) and their abstractions \( M'_1, \ldots, M'_n \), for any path \( \pi \) in \( M'_1 \| \ldots \| M'_n \), \( \pi \) is consistent with \( M_1 \| \ldots \| M_n \) if and only if:

1. \( \pi \) is consistent with \( M_1 \)
2. \( \pi \) is consistent with \( [M_1]_{\Sigma_{M_2} \cup \Sigma_n} \| \Sigma_2 \\| M_2 \)
3. \( \pi \) is consistent with \( [ \ldots [M_1]_{\Sigma_{M_2} \cup \Sigma_n} \| \Sigma_2 \\| M_2 ]_{\Sigma_3} \| M_3 \) \ldots \)
4. \( \pi \) is consistent with \( \ldots [ \ldots [M_1]_{\Sigma_n} \| \Sigma_2 \\| M_2 ]_{\Sigma_3} \| \Sigma_4 \ldots ]_{\Sigma_n} \| M_n \)

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 5.2. 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 \( [M]_{\Sigma} \leq [M]_{\Sigma'} \)).

**Proposition 5.7.3.**

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 5.1, 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 $\Sigma_{M_1} \cup \Sigma_{\pi}$. 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 5.2.** Given $n$ components $M_1, \ldots, M_n$ and their abstractions $M'_1, \ldots, M'_n$, for any path $\pi$ in $M'_1 \ldots || M'_n$, all the conditions in Corollary 5.1 hold if and only if all the following are satisfied

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

**Proof.** Each item in this corollary implies the corresponding item in Corollary 5.1 because of Proposition 5.7.3. Therefore if all the conditions in Corollary 5.2 are satisfied, then all the conditions in Corollary 5.1 hold.

For every condition in Corollary 5.2 proving $\pi$ consistent with some $M_i || \tilde{M}_i$, there are conditions in Corollary 5.2 in which each $M_i$ appears (perhaps abstracted, but due to Proposition 1, it still is consistent). Therefore when all the conditions in Corollary 5.1 hold, in particular $\pi$ is consistent with every $M_i$ appearing in any condition of Corollary 5.2. Thus, all the conditions in Corollary 5.2 are satisfied.

The last condition of Corollary 5.2 includes as sub-expressions the previous conditions, from here we infer the algorithm.

![Diagram](image)

Part (1) (matching the first condition of Corollary 5.2), 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 5.2, 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 $V_2$ (Part (4)). This continues until either the abstract counterexample is found spurious (one of the conditions in Corollary 5.2 does not hold) or it is consistent with every event specification, making the counterexample real.

5.8 Liveness abstract counterexample spuriousness checking

In this section we describe in more detail line 4 of algorithm 5.2, i.e. how to check whether “$\pi$ is consistent with currModel” where currModel is the current event specification composed with the abstraction of the previous ones.

Recall that the concrete model is given by $(T_P + M)||E_1||\ldots||E_N$ where each $E_i$ is an event specification, while the abstract one is given by $(T_P + M)||E'_1||\ldots||E'_N$ where each $E'_i$ is an abstraction of $E_i$.

For safety it is enough to simulate the abstract counterexample (a finite path) with currModel. For liveness, we propose the new instrumentation technique below in place of the usual unfolding seen in Section 2.2.

The abstract counterexample (that shows a liveness formula unsatisfied) looks like:

$$s_0, s_1, \ldots, s_i, s_{i+1}, \ldots, s_j$$

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 instance of currModel is instrumented with a counter ($cnt$ belonging to the range $1..j - i$) that represents the states within the loop. To check spuriousness we check whether there is a concrete path (in currModel) that is consistent with the abstract counterexample. Note that we check with each event specification (composed with previous events’ abstractions) separately. Once currModel is instrumented with the counter ($cnt$) obtained from the abstract counterexample, the LTL formula to be checked is

$$\neg \left( s_0 \land X s_1 \land \cdots \land X s_i \land X s_{i+1} \ldots X G \left( \bigwedge_{k=1}^{j-i} \left( cnt = k \implies s_{i+k} \right) \right) \right)$$

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 (being modularly checked considering currModel) 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 (and previous abstractions); otherwise, a path has been found showing the counterexample consistent with the current currModel and the abstract counterexample is checked against the remaining ones. Note that this formula is checked against each (concrete) event specification and previous abstractions, and not the...
whole concrete system.

5.9 Instrumentation correctness

To see that instrumenting and checking Formula (5.3), which is
\[
\neg \left( \text{prefix} \land \text{loop} \phi \right),
\]
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 \( M \).

\[
M \models E\pi \iff \tilde{M} \models E\phi \iff \tilde{M} \not\models \neg E\phi \iff \tilde{M} \not\models A\neg\phi
\]

Relation (a) holds since the instrumentation added to the model does not restrict nor add paths to \( M \), and has no effect in the counterexample. The \( \iff \) part of (b) is easy to see: if \( \pi \) is a path in \( \tilde{M} \), then there is an assignment to \( \text{cnt} \) such that the formula holds in \( \tilde{M} \). The remaining steps are trivial logic identities. Thus we have obtained that \( M \models E\pi \iff \tilde{M} \not\models A\neg\phi \), therefore \( \tilde{M} \equiv A\neg\phi \iff M \not\models E\pi \). If the instrumented model satisfies formula (5.3), then the path is not consistent with the model and is found spurious.

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

Let \( \tilde{\pi} = \tilde{s}_0, \tilde{s}_1, \tilde{s}_2, \tilde{s}_3, \tilde{s}_4, \tilde{s}_5, \ldots \) 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 \( \tilde{M} \) and for every pair of consecutive states \( \tilde{\pi}_i \) and \( \tilde{\pi}_{i+1} \), \( \tilde{M} \)'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 \( \text{cnt} \) 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 \( \tilde{M} \) consistent with \( \pi \). Note that the formula does not force the concrete loop proving the \emph{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 (5.3) over the instrumented model.

5.10 Refining

Refinement is obtained from an event specification (composed with the previous event abstractions) against which the counterexample has been found spurious. Let \text{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.
For safety guarantees, one SMT unsat-core activation simulating the finite counterexample with \textit{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.

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 may include fairness constraints (restricting the language of the state machine to include only fair paths, c.f. Section 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 \( \phi \), the state machine includes the transition relation and fairness constraints representing \( \phi \). When translating the liveness formulas into their corresponding state machine, we save what fairness constraints each liveness formula introduces.

For liveness guarantees, the refinement to avoid a spurious abstract counterexample is also obtained from \textit{modelFindRefs}. This refinement either includes transition relation constraints (refine the model by splitting the abstract states so that the current path is no longer feasible in the model) or liveness properties (obtained from the event specifications).

If the abstract counterexample is consistent when checked against \textit{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. Otherwise, there is some transition in the abstract model that is not allowed according \textit{modelFindRefs} and the refinement will be a transition relation constraint.

\textit{Example: Liveness Property Refinement.} The following guarantee expresses that whenever the location is provided (\textit{locationProvided}) by a witness of a crisis within a witness call (\textit{inCall}), then that phone call will eventually end (\textit{callEnd}).

\[
G((\text{locationProvided} \land \text{inCall}) \rightarrow F \text{callEnd})
\]  

(5.4)

Let’s assume we are trying to verify the guarantee (5.4) and we have obtained the counterexample

\[
\pi = \text{callStart} , \text{locationProvided} , (\neg \text{callEnd})^\omega , \text{inCall} , \neg \text{callEnd}
\]

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.

One of the event specifications (\textit{witnessInfoProvided}) assumes that every call that starts eventually ends: \( G(\text{callStart} \rightarrow F \text{callEnd}) \) and that a call does not start and end at the same state \( G(\neg(\text{callStart} \land \text{callEnd})) \). This can be represented by the state machine in Figure 5.4. 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 \( s_1 \). \( s_1 \) is not a fair
state, so the only way to achieve a fair path is by eventually reaching $s_4$ (guaranteeing that every call that has started, eventually ends).

![State machine representing part of WitnessInfoProvided's specification](image)

**Figure 5.4: State machine representing part of WitnessInfoProvided's specification**

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

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 (from some liveness properties).

**Example: Transition Relation Refinement Required.** Let’s assume we are trying to verify the guarantee:

$$\mathbf{G}((\text{shouldGoToLoc} \land \neg \text{badWeather} \land \text{helicoptersAvailable} \land \text{problematicAccess}) \rightarrow \mathbf{F} \text{HMAdded}),$$

and we have obtained the counterexample $\pi = \left\langle \begin{array}{c}
\neg \text{shouldSendHeli} \\
\text{shouldGoToLoc} \\
\neg \text{badWeather} \\
\text{helicoptersAvailable} \\
\text{problematicAccess} \\
\neg \text{HMAdded}
\end{array} \right\rangle^\omega$

That is, all the conditions for the event $\text{shouldSendHeli}$ 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 $\text{shouldSendHeli}$, even when removing any fairness constraints of the concrete $\text{modelFindRefs}$. Therefore we conclude that in this case we need a transition relation refinement.

In the next paragraphs we will describe how to find which are the constraints that avoid the counterexample and, in case they are required, which are the liveness properties introducing those constraints.

**Transition Relation Refinement.** 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 ($\text{modelFindRefs}$) consistent with the abstract counterexample. If we consider the product of $\text{modelFindRefs}$ 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.

**Liveness Property Refinement.** This must be handled differently 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.

Recall that we have identified that a liveness property refinement is required by observing that the 
abstract counterexample $\pi$ is spurious with `modelFindRefs`, but it is consistent when checked against 
`modelFindRefs` without including any fairness constraints. That is, if $\mathcal{F}$ is the set of fairness constraints, 
then there exists a path $\tau$ witness of $\pi$ being consistent with `modelFindRefs` \ $\mathcal{F}$. From $\tau$, we can obtain 
how many times the abstract loop has to be unfolded to represent a concrete loop. Thus, the current 
event specification (with previous abstractions) and 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 $f$”.

If an abstract counterexample has been found spurious and the needed refinement includes fairness 
constraints, the translation to SMT of the current model (current event specification and abstraction of the 
previous ones), fairness constraints and counterexample generates an unsatisfiable model for which the 
unsat-core includes the information necessary to avoid the counterexample.

The counterexample was shown consistent with the model without any fairness constraints but spurious 
otherwise. Thus, the unsat-core 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 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.

### 5.11 Implications

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 alphabet refinement contribution, 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 counterexamples becomes easier to understand (there are fewer variables to be considered).

### 5.12 Related Work

In [DK13], the idea of using CEGAR for event systems is proposed, but here we show the mechanism and elaborate a tool implementation, including new techniques for handling liveness and reducing the state space.

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, [HJMS02, CKSY05, BR01] consider only safety formulas, simulate abstract counterexamples against the system’s implementation, and learn new predicates that refine the abstract model. The work in [CGJ+00, CGP+07] present non-compositional CEGAR approaches for also checking liveness formulas. The bound to which liveness counterexamples can be simulated to detect spuriousness in [CGJ+00] 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 [CGP+07], 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. In [BAS02], 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 is different. In that work, 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. It is known that every LTL formula can be expressed by a Büchi automata (representing \(\omega\)-regular languages - infinite paths). However there are \(\omega\)-regular languages not expressible in LTL. For the particular \(\omega\)-regular languages representing an abstract counterexample, we have shown that they can be checked as an LTL formula by the additional instrumentation of the model. The work in [CGP+07] instruments the program to include fairness related assertions and reduces the problem to analyzing binary fair reachability (checking that every possible cycle satisfies the fairness constraints). As above, we use instrumentation for a different purpose. Theirs is checking liveness properties in a program, while ours is checking whether a given abstract path representing a counterexample to a liveness formula is spurious.

Other works have considered compositional CEGAR approaches [CCG+04, Chu12, HJMQ03, GBPG08] for safety guarantees. The abstract components in [CCG+04, Chu12, HJMQ03] have to include every symbol shared among components, only [GBPG08] 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 [AAC+05, DFS04, BMAK11] rather than related work adding events to existing paradigms including limited composition (at most boolean composition) as [RL08, GSM+11]. For example, tracematches and trace-based aspects [AAC+05, DFS04] 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. Applying our techniques allows understanding which of the pointcuts are relevant to which guarantees.
Chapter 6

Tool implementation

As noted before, many CEGAR tools use SMT solvers to build the abstract model or find predicate refinements. In this work, we have implemented a tool DaVeRS in Java. It includes a full reimplementation of MAVEN [GKK10] (which was previously implemented in Ruby and built the augmented model of the tableau of the assumption and the aspect being considered), and has been extended to include the optimizations, augmented specifications, and interference analysis as described in this thesis. DaVeRS interacts with an SMT solver (SMTInterpol [SMT]) and with a BDD-based model checking tool (NuSMV [NuS]), which assumes finite models and can check LTL properties. In addition we use the ltl2smv tool included in NuSMV to translate LTL formulas into SMV models. Excluding the unit tests and lexers and parsing classes, the software has 7 KLOC. The tool and example case studies are available at http://www.cs.technion.ac.il/~cdisenfe/phdthesis/.

In the next two sections we will describe DaVeRS input and output and describe the system arguments in Section 6.3. In Section 6.4, we describe in more detail the implementation of each step of the CEGAR algorithm (based on Chapter 5).

6.1 Input

Recall that event specifications are given by $E = \langle X_{ev}, X_{lower}, X_{internal}, P, R \rangle$ (Section 3.2) representing the temporal logic constraints for the event assumptions and guarantees. Responses are given by $A = \langle X_B, X_R, ED, M, P, P_{Ev}, R \rangle$ (Section 3.3), 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 thesis.

The different libraries of event specifications should be available within a folder where the response is located, and DaVeRS searches for the relevant event specification among all the possible ones.
6.2 Output

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. The counterexample is given as in NuSMV’s output, indicating the value of each variable at every state and where the loop of liveness property counterexamples starts.

6.3 Arguments

The tool can be used for event or response verification, or for checking interference among responses. The possible arguments are:

nc (No CEGAR) disables the CEGAR algorithms, that is the first check includes all possible events to check whether the property is satisfied.

ns (No Shared Alphabet) uses our optimization that does not include the shared alphabet in the first abstract version of the system.

ins (Instrumentation) uses our optimization that instruments the concrete model to check spuriousness of liveness property counterexamples.

ii (Instrumentation improvement) if the counterexample of a liveness property is spurious due to missing safety constraints, use the diameter obtained to find the necessary refinements.

check responseFile checks the guarantee of the response provided in responseFile.

interference_assumption responseFile1:responseFile2 checks whether the assumption of responseFile2 is preserved by responseFile1.

interference_guarantee responseFile1:responseFile2 checks whether the guarantee of responseFile2 is preserved by responseFile1.

interference_internal responseFile1:responseFile2 checks whether responseFile1 could be activated within responseFile2, and in that case whether responseFile1 satisfies the internal assumption of responseFile2.

6.4 CEGAR implementation

In the following sections we describe the implementation of the different steps involved in our CEGAR algorithm.

6.4.1 Abstraction

The first abstraction of the composition depends on whether we are using the alphabet refinement optimization Section 5.6. When this optimization is not used, the initial abstraction must include any shared symbol among the events. Thus, in such case before starting the first iteration of the CEGAR cycle, the set of relevant events is obtained from all the available libraries (to find the set of shared symbols).
This is done by finding all those affecting (directly or indirectly) any event detector appearing in the response definition. From the set of relevant events the shared alphabet among these events is added to the response. When the alphabet refinement optimization is used, finding the set of relevant events can be postponed to after the property has been found to be unsatisfied by the abstract model and spuriousness checking against each relevant event specification is to be applied.

The parallel composition of several SMV models is given by considering every constraint of each model. That way, shared symbols among models must have consistent transitions to be able to go to the next state.

In subsequent steps of the CEGAR algorithm in which the abstraction includes information from the event specifications, if the refinements are LTL formulas, then they are first translated to SMV (using ltl2smv) and considered in the composition.

The first component represents the response woven to its assumption about the underlying system. This is represented in a SMV model by adding some auxiliary variables representing whether the current state belongs to the underlying assumption or response, or whether it is a state within the internal assumption of the response. The necessary transitions are added from any state where the event detector occurs to the response, and from the returning states of the response to the underlying assumption.

### 6.4.2 Verification

Verification is applied using the symbolic model checker NuSMV. The tool expects a complete model (where every state has at least one successor) to provide sound results. If the tool finds the formula false, then it is indeed false (the counterexample belongs to the model). The problem is that NuSMV may say that a formula is satisfied by a model even when it is not (when the model is incomplete, i.e. has deadlocks). Thus before applying verification, we have to remove any deadlocks. To deal with this issue in [Kat10], states without successors were identified by repeatedly calling the reachability analysis tool of NuSMV. Then, additional constraints avoiding those states were added, by forbidding the next state variables to have the exact values of the state without successors. This constraint includes the value of every variable of the model, even those not related to reaching deadlock. Therefore, if \( V \) is the set of variables of the model, and \( V' \subset V \) is the set of variables actually causing deadlock in the next state, the given technique has to find every possible combination of the variables in \( V \setminus V' \). DaVeRS also takes advantage of the reachability analysis results, but in order to find the actual information that causes deadlocks and build the complete model faster, it interacts with an SMT solver. In particular, we first check whether the state \( (s) \) without successors is an initial state or not. This is by invariant checking \( \neg s \). Invariant checking does not suffer from soundness problems when the model has deadlocks. The counterexample to the invariant provides the path reaching the state without successors. Now, knowing whether it is an initial state or not, we can use the SMT solver simulating the initial step or a transition and from the unsat core get the information of the counterexample causing the deadlock. This is again done as long as the model has deadlocks, but every time it constraints over the set of variables really causing the deadlock.

Since we have observed that when the formula is shown false we do not need to remove deadlocks, we have implemented that the first reachability check (to analyze deadlocks) and checking the property are done in parallel. If the property is found false, the reachability check is no longer necessary and is interrupted. Otherwise, deadlock removing continues as explained above.
The necessary constraints to remove deadlocks are saved so that if there are future iterations with the original model and some refinements originating from the event specifications, these refinements only add constraints, so we still want to avoid the original deadlocks.

6.4.3 Counterexample analysis

We have implemented the different strategies for analyzing spuriousness of the counterexample. For the shared alphabet strategy, it is enough to check whether the counterexample is consistent with each event specification. For the non-shared alphabet strategy, we have mentioned that we take the COI reduction of each model to compose with the next event specification of the sequence. NuSMV allows obtaining the COI reduction of a model to a set of variables. We then simply consider this model along with that of the next event detector.

Depending on whether the property is a safety or liveness guarantee, different techniques are used. NuSMV’s 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 and liveness properties, in [AS87] 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 can occur infinite times thus all states are accepting states (therefore, the state machine represents a safety formula).

To check whether a safety property counterexample is consistent with a model, we use the SMT solver to simulate every step of the counterexample and find the unsat core.

For liveness formulas, we check check in NuSMV whether the property representing the counterexample is feasible in the model.

6.4.4 Refinement

According to the ideas explained in Section 5.10, both for transition relation or fairness refinements the information is obtained from the unsat core of the SMT solver.

In Section 5.10 we have distinguished between needing a liveness or a safety related refinement by checking the spuriousness of the abstract counterexample with the current model without any fairness constraints (syntactically removing them from the SMV definition).

When a transition relation refinement is needed, we compose the model with which the counterexample was found spurious with the model representing the counterexample (obtained using ltl2smv). Since there is no path consistent with both, analyzing reachability shows the model empty (there are no infinite paths) and the model checker outputs the diameter analyzed to reach such conclusion. This diameter is then used to know how many steps have to be simulated when calling the SMT solver.

When a fairness constraint refinement is needed, the translation is straightforward according to Section 5.10.

When our tool builds the input of the SMT solver, it names the assertions ([BST10]) to distinguish between the constraints representing the counterexample and those part of the model (initial and transition relation constraints). Given the unsat core, we obtain only those that are part from the model. In addition, we know that the tool that translates LTL formula into SMV adds variables to name the formula subexpressions(LTL_{i SPECF_j} where i is the id of the formula being translated and j is a different number
for each subexpression). Thus, given a variable of this form we can know what the original LTL formula is and add it as a refinement.

6.4.5 Translation

Given that DaVeRS interacts with different tools, some translations are required. We have used ANTLR to generate the lexers and parsers of the subset of NuSMV and SMTLib languages used.

6.5 Interference

The tool allows checking interference among responses and allows in particular expressing and verifying responses with internal assumptions.

For each of the internal assumptions represented, DaVeRS builds the augmented system when using them and allows checking if another response could be activated and thus would require the internal assumption to be satisfied.
Chapter 7

Case Studies

We have evaluated our techniques with three extensive case studies: a Car Crash Crisis Management System (CCCMS) [KGM10], a Discount library (as in [DK13]) and a security concern in an email application. The examples can be found in http://www.cs.technion.ac.il/~cdisenfe/phdthesis/. 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 response encrypts this password (or any other password sent within the application).

In addition, we have implemented the transaction authentication case study of [DK12] (seen in Section 4.1) to analyze interference among responses.

### 7.1 Evaluation

Recall that in classical compositional CEGAR all potentially relevant variables are included in advance, 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 2.5GHz 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 minutes for each example.

Figure 7.1 shows the time taken by the classical compositional CEGAR techniques Included+unfolding (triangles) and our Non-included+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 non-included+instrumentation strategy was used (included+unfolding always took the same or more iterations). The y axis shows the time in seconds. When the marker for the classical techniques applied to safety formulas does not appear, it means that the evaluation took more than 15 minutes.
minutes. Among the liveness formulas, when the marker does not appear, there are two cases in which applying the classical techniques took around 3 minutes, another one in which it took 10 minutes, and the remaining cases in which it took more than 15 minutes. 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 20 seconds for every guarantee considered.

Figure 7.2 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 7.3 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 minutes) 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 minutes 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 significant larger when using classical techniques. Thus, classical techniques reached timeout for every liveness guarantee in the Security case study (satisfiable and unsatisfiable). For unsatisfiable safety guarantees considered, our technique took less than 20 seconds, while classical techniques took between 40 seconds and 2 minutes (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 more event specifications are involved, our alphabet refinement optimizations avoids automatically including the interface alphabet of all these, thus considering only the necessary refinements.
Figure 7.2: CCCMS - Unsatisfiable properties

Figure 7.3: Discount, Security - Satisfiable safety properties
Chapter 8

Conclusions

We have improved state-of-the-art techniques to specify and verify responses in hierarchical event-based systems. To do so, we have analyzed the basic assumptions about hierarchical event-based systems, and observed that the system can be represented as a parallel composition. This composition can be used to apply similar assume-guarantee verification techniques as those in MAVEN [GKK10]. 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.

Moreover, we now allow responses to be activated within other responses, by describing the new version of response specifications and proof obligations to verify, check interference and analyze cooperation. Regarding the specifications, we now distinguish between local and global guarantees, that is, those expressing what should happen at each location where the response is activated against those properties referring to the whole computation path. In addition, responses now include an internal assumption expressing what is expected about any other response that may be activated within the current evaluation.

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. 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).

DaVeRS implements the ideas presented so that the presented techniques are transparent to the user. Three case studies were considered, but the ideas are applicable to any other system satisfying the basic assumptions. The results in the evaluation section validated these as really improving performance with respect to techniques in related work. In addition these ideas are applied to the response definition and event specification, and thus can be applied while the system is being built or when the whole implementation is available.

The tool design and new optimizations are crucial in making the theoretical approach practical and automatic. A straightforward model-check or use of CEGAR would have resulted in a tool that is too expensive or time-consuming for routine use, or requires too much knowledge of model-checking.
Bibliography


[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. SIGPLAN Not., 2005.


[NuS] NuSMV. "http://nusmv.fbk.eu/".


[SMT] SMT. http://ultimate.informatik.uni-freiburg.de/smtinterpol/.


Besides

The presented optimizations are shown as an efficient tool for verification in the example used to verify the opposite (whether it is a real or a fake)

For formulas or living (Liveness)

In order to avoid adding all the events within the model.

These ideas are also useful before implementing the system (since the method is based on details, not code) or when there is an implementation (then you can use all the state machines that describe the implementation).

For learning assumptions on modules for events through their details: 1) Reused use for different implementations of these modules, 2) Simplify details implementation 3) Calls the assumptions learned (since they are taken from existing details), and 4) More detailed dependence between reactions (by checking the new general theorems).

The tool accepts as input a state machine that describes each reaction together with its details, and libraries of details for modules for events, and applies our techniques for checking the reaction (or finding a fault or cooperation). At the end of the algorithm, the tool prints whether it is the solution, keeps the assumptions learned during the algorithm (CEGAR), and in case of failure it prints the counterpart and the true opposite sample (including the learned ab).

We examined three examples of libraries that include events and reactions.

These examples demonstrate the use of techniques in this thesis and are also specific to their libraries and also in the user libraries.

Quantitative results show a significant improvement in performance using the optimizations.
A computer science Ph.D. student at Technion, in the Computer Science Department. He is working on the development of algorithms for verification and synthesis of reactive systems. His research focuses on the use of counterexample-guided abstraction refinement (CEGAR) techniques to automatically generate proofs of correctness for complex systems.

In his thesis, the student presents a novel approach to verification that combines CEGAR with satisfiability modulo theories (SMT) solvers. This integration allows for the automatic generation of proofs of correctness for software systems, even those with complex interactions and concurrent behaviors.

The student's work demonstrates the effectiveness of his approach through detailed case studies and comparisons with existing techniques. His research has significant implications for the field of software engineering, as it offers a more efficient and scalable method for verifying complex systems.

Technion - Computer Science Department - Ph.D. Thesis PHD-2015-03 - 2015
In systems that respond (react to events), events are stimuli to events that matter (events). Modules for event detection (event detectors) can be enabled to detect complex events by using and combining information and hierarchically hierarchical events. However, modules for event detection are only allowed to observe the system in order to note an event, unlike the reactions that can change the system's state, its control flow, or lead to other events.

The reactions we are referring to are similar to aspects (aspects) in AOP (Aspect-Oriented Programming) [Kiczales et al. 1997]. AOP allows the expression of functional requirements influencing scattered, modular places in a unit of code that specifies where to add functionality and what needs to be done. Examples of aspects are Logging, Persistence, Exception Management, and so on. In the same way that aspects are operated when a primitive event occurs (a method call, an exception is thrown), reactions are operated when an interesting event is revealed (including complex events).

In previous work related to aspect validation (MAVEN [Goldman et al. 2010]), we verify every aspect by a set of A-O-P (Assume-Guarantee) formulas (P, R). For each aspect given, we modify the system and verify the output. We use Linear Temporal Logic (LTL) to express the properties that must be satisfied after a time for every path in the model. For example: every request will eventually be handled. Therefore, we can verify every aspect individually: given P and R, we build a model of the system that represents all the paths that provide the assumption (P) and only those. We expand the model with the aspect (weaving) by adding transitions from the model of P to the aspect model at the right places, and back considering the actions performed by the aspect. If the resulting model satisfies the aspect validation (R), we have a formal model of the system, the unit of code that expresses the aspect, and the expanded model that satisfies the aspect expressions. We can use this expanded model to simulate the execution of the system.

In this work, we present the basic assumptions, define precisely the contents of the modules for detecting events, and expand the MAVEN ideas to verify that the reactions resulting from events that the reaction depends on.

Verifying each reaction individually is not sufficient to ensure that the whole set of reactions will satisfy all requirements. One reaction can affect other reactions (and interfere), for example when a different variable is changed by two reactions. Additionally, reactions can cooperate to provide their specific details. Under the basic assumptions, we can model event-based asynchronous systems (asynchronous integration).
המחקר בוצע בהנחייתו של פרופסור שמואל כ"א, בנוכחותו למידויה המחשב.

אני מודד לתכינו על התמ וכיום הכספים המודדים בברשתלאות.
אימוט פורמלי לручבה של מאורעות והנובות

הباح על מחקר

לשם מיולי חלבוק של הדורייה ליקבלת התואר
דוקטור לפילוסופיה

סינטיה דיסנפלד

רונן לסקט הנסכין --- מון טכנולוגי לישראל
שבט החול'ו'ף מווניירפ 2015
אמות פורמליות לריכבון של מאורעות ותנובות

סנטיה 디נספלד